Creating a Shell Script

```
#!/bin/bash
# This is simple.sh, a simple bash script
echo "Hello, I am $LOGNAME"
echo Today is `date`
echo Bye-bye
echo "the PID of this is " $$
```

- A bash script is a file containing a list of commands to be executed by the bash shell
- First Line of a shell script (`#!/bin/bash`)
  - Specifies which shell program (here the bash) will be used to interpret the script
- Second line is a comment starting with "#" (not #!)
- The rest are shell commands

Execute a Bash Script

- **`source simple.sh`**
  - not commonly used, not recommended
- Method One
  - Add execution permission to the script

    If you run the script: `script.sh`, you will get error: `command not found` if the current directory is not included in PATH, two ways to solve this
    - Add the current dir to PATH : `PATH=$PATH:./`
    - Provide the path to run the script as: `./simple.sh`
      - Means to run the script from current directory
- Method Two
  - Type: **`sh (or bash) script.sh`**
    - Here sh refers to bash (a symbolic link to bash) in the CS lab systems

Debugging in Bash

- With **`-n`** option when running the script
  - Check syntax error in the script
  - Will not execute the script
- **`bash -x scriptname`**

  Turn on echo option, display each line in script (with variable expansion/substation) before execution
- **`bash -v scriptname`**
  - Verbose option, display each line (without variable expansion/substation) before execution
- Turn on/off the above options in the current shell with “set”
  - Turn ON with **`set -x`** or **`set -v`**
  - Turn OFF: **`set +x`** or **`set +v`**

Read from User Input (stdin)

- The **`read`** command

```
echo –n "please enter student name & grade:"
read name grade
echo name: $name , grade: $grade
```

- Read for multiple variables

```
read var1 var2 var3 var4
```

  - No commas to separate the variables
  - Values are read from standard input and assigned to each variable
    - If more words are typed in, then the excess get assigned to the very last variable
    - If more variables are assigned than the variables given, the excess variables are empty

Command Line Arguments

- The command line arguments can be referenced in scripts with positional parameters
  - **`$0`: the script/program itself**
  - **`$1`**: the first argument
  - **`$2`**: the second argument, … **`${10}`**
- **`Example:`**

```
sh simple.sh apple pear orange
```

  - `$0`➔`simple.sh`
  - `$1` ➔`apple`
  - `$2`➔ `pear`
  - `$3`➔ `orange`
  - `$4??? (empty)`

Positional Parameters

- **$0** references the name of the script
- **$1 … ${10}** references individual positional parameters
  - You need use {} for index of larger than 9, ex: ${10}
    - $10 ➔ $1 + 0 ???
- **$#**: the number of command line arguments (or number of position parameters excluding the program itself!!!)
- **$***: lists all the positional parameters, separated by a white space, $1 $2 $3
- **$@**: list of all the arguments, $1 $2 $3
- There is difference b/w **$@** & **$*** when double quotes are used.
  - **“$*”** → “$1 $2 $3” (a string)
  - **“$@”** → “$1” “$2” “$3” (a list of string => an array)

The `shift` command

- **`shift`** command-shell built-in command
  - `help shift`
  - Shifts the positional parameters to the left a specified number of times,
    - **`shift 5`**
      - shifts 5 times to the left
      - shifts left once if no number specified
- **`$0`** is not affected by "**`shift`**" command, it is still "storing" the program name

```
#!/bin/sh
# bash script fruit.sh

echo "\$0 is $0"
echo "\$1 is $1"
echo "\$2 is $2"
echo "\$3 is $3"
shift 2

echo "After shift 2 "
echo "\$0 is $0"
echo "\$1 is $1"
echo "\$2 is $2"
echo "\$3 is $3"
```

What if you run the script as:

```
sh fruit.sh apple pear peach
```

```
./fruit.sh apple pear peach
$0 is ./fruit.sh
$1 is apple
$2 is pear
$3 is peach
After shift 2
$0 is ./fruit.sh
$1 is peach
$2 is
$3 is
```

Arithmetic

- Bash can perform very simple integer operations
  - You can always use **awk** to process float numbers
- An integer variable can be declared with the shell built-in command `declare`, then followed with value assignment.
  - **`declare -i num #Create an integer variable`**
  - **`num=5+5; echo $num`** ➔ **`10`**
  - **`num=4*6; echo $num`** ➔ **`24`**
  - **`num=6.5`** #this will get error, NO FLOATING NUMBER OPERATION
  - If you attempt to assign a string to an integer variable, bash assigns 0 to the variable
    - **`num=TODAY; echo $num`** ➔ **`0`**
- NO SPACE AROUND "=" and "+", "*"

The 'let' Command

- The `let` command: A bash built-in command used to Evaluate arithmetic expressions
  - `let x=2+5`
  - `let y=" x + 5 "`
  - `let y=' 2 + 5 '`
  - `let y+=2`

  no space around "=", space is allowed if quotes are used.

  `$` sign is not used inside 'let'
- Arithmetic operators: `+ - * / %`
- More have been added now, check online with

  `help let`

Numeric Expression Expansion

- The square-brackets or double parentheses can be used to substitute the `let` command
  - `$[ expression ]`
    - `sum=$[ 5 + 4 - 2 ]`
    - This will be deprecated in future Bash
  - `$(( expression ))`
    - `echo $(( 5+4-2))`
    - `num=10;`
    - `num=$(( $num+10 ));`
    - `Num=$(($num+10))`

Note: Need to have a space for the `[` and `]`, `((`, and `))` in the old version shell, though it seems ok in the new version bash on CS Linux systems

The `expr` Command

Evaluate arithmetic expressions

- Usage: expr EXPRESSION
- Operations
  - `* / % + -`
  - Must have space around the operators
- Examples
  - `expr 1 + 4` ➔ `5`
  - `expr 1+4` ➔ `1+4 (no space!!)`
  - `expr 5 \* 4`
  - `expr 11 % 3`
  - `num=1; sum=`expr $num + 10`; echo $sum`

Built-in Test Operation

- Variable comparison: `[ arg1 opt arg2 ]`
  - MUST have a space after "[" and before "]" and around opt sign, strings/variables need to be quoted with double quotes
- opt for String Testing
  - [string1 = string2] <=> [string1 == string2]
  - [string1 != string2]
  - `-n str1` : str1 is not a null (defined)
  - `-z str1` : str1 is zero length (empty)
- opt for Numerical Comparison
  - Options: `-eq, -ne, -lt, -le, -gt, -ge`
  - Ex: `[ num1 -eq num2 ]`
- Logical comparison with `||`, `&&`, and `!` (not)

File Attribute Checking

Operator	True if
`[ -d file ]`	*file* exists and is a directory
`[ -e file ]`	*file* exists (any type)
`[ -f file ]`	*file* exists and is a regular file
`[ -r file ]`	You have read permission
`[ -s file ]`	*file* exits and is not empty
`[ -w file ]`	You have write permission
`[ -x file ]`	You have execute permission on *file* For directory, it's the search permission
`[ file1 -nt file2 ]`	*file*1 is newer than *file*2
`[ file1 -ot file2 ]`	*file*1 is older than *file*2
`[ -x f1 -a ! -d f1 ]`	Logical AND
`[ -x file -o -d file ]`	Logical OR

Note: No space between "-" and the option

The `if/then/fi` Command

```
if command1
then
   command2
   command3
fi
```

The exit status of command1 will be examined. Command2 and command3 will be executed only if the exit status of command1 is zero (successful)

```
if grep "$name" /etc/passwd > /dev/null 2>&1
then
    echo Found $name
fi
```

The if/else Command

```
if cmd1
then
   cmd2
   cmd3
else
   cmd4
fi
```

```
if cmd1; then
   cmd2;cmd3
else
   cmd4
fi
```

```
#!/bin/sh -x
export name=cs390
echo name=$name
echo number of argument: $#
if [ $# -lt 2 ]; then
   echo "need two arguments!"
   exit 1
fi
file=$1
str=$2
if  grep "$str" $file ; then
   echo Found $str in $file
else
   echo  "$str is not in $file"
   exit 1
fi
```

Flow Control with `if/then/elif/else/fi`

```
#!/bin/bash
if [ ! -e $1 ]; then
    echo file $1 does not exist.
    exit 1
fi
if [ -d $1 ]; then
    echo – n  "$1 is a directory that you may "
    if [ ! -x $1 ]; then
          echo -n "not "
    fi
elif [ -f $1 ]; then
   echo "$1 is a regular file."
else
    echo "$1 is a special type of file."
fi
if [ -r $1 -a  -w $1 ]; then
    echo "You have read and write permission on file $1"
fi
if [ -x $1 -a ! -d $1 ]; then
    echo "you have execute permission on file $1"
fi
```

The “case”

```
case $type in
     [Qq]) echo –n “Please enter quiz grade: ”
          read gradeQ
          ;;
     H|h) echo –n “Please enter hw grade:”
          read gradeH
          ;;
     M|m) echo –n “Please enter test grade: ”
          read gradeM ;;
     F|f) echo –n “Please enter final grade:”
          read gradeF
          ;;
     *) echo “Wrong grade type!”
          ;;
esac
```

The `select` Command

- **`select`** is not available in other conventional programming languages
- Generates a menu of each item in the list and indexes the item
- Repeats the process forever
- Normally used together with “case”

```
filelist="ab abc ad QUIT"
select name in $filelist ; do
  case $name in
  "QUIT")    echo "Exiting."
             break
            ;;
          *) echo "You picked $name "
             chmod go-rwx "$name"
             ;;
   esac
done
```

```
1) ab
2) abc
3) ad
4) QUIT
#?
```

The Looping Commands

```
#!/bin/bash
answer="yes"
while [ "$answer" == "yes" ]
do
  echo -n “Build grade record for student (yes or no): "
  read answer
  if [ “$answer” == "yes" ] ; then
     echo You have selected to enter student record!
  else
     echo You have done with entering student score!
  fi
done
```

```
answer="yes"
until [ "$answer" == "no" ]
do
  echo -n "Do you want to build up score record for students (yes or no): "
  read answer
  ……….
done
```

The “for” Loop

```
filelist=“apple peach”
for file in $filelist; do
  echo file is $file
done
```

```
for file in `ls -1`
do
    echo file is $file
done
```

```
filelist=`ls -1`
for file in $filelist; do
  echo file is $file
done
```

```
for file in `ls -1`; do
  echo file is $file
done
```

loop over range of integers

- If step is ‘1’

```
for i in {1..10}; do  echo $i; done
```

- Otherwise use ‘seq’: print a sequence of numbers

```
#Usage: seq first increment last
for i in `seq 1 2 10`; do echo $i; done
for i in $(seq 1 2 10); do echo $i; done
```

Looping Control Commands

- **`break`**
  - Shell built-in command
  - Used to force immediate exit from a loop, NOT from a program
- **`continue`**

  Returns control to the top of the loop (skip the rest inside the loop)
- **`exit`**
  - Exit the program regardless where it is
    - Can be inside a loop, or function
  - Be careful when using “exit”
  - Normally provide an integer for the exit status: `exit n`
    - 0 for success
    - 1 or other integer for failure or different type of errors

Function

- Function is a script-within-a-script
- Defining a function
  - **`function`** `functname { shell commands }`
  - **`functname ()`** `{ shell commands }`
- Function arguments
  - Just like running a script and has its own position parameters
  - Can have its own local variable defined using `local var` inside the function
  - Conventionally, functions are defined (put) before the main part inside the script
- Function's return Value
  - Using return cmd: **`return num`** **# num from 0-255,**
  - The value is stored in the special variable ?
  - Assign the STDOUT of the function to a variable using `var=$(funcname) (similar to command substitution)`

```
#!/bin/bash
finfo()
{
      echo Got $1
}
for filename in $@ ; do
    finfo $filename
    echo
done
```

To execute the ` script

```
./fileinfo2.sh `ls`
```

Or

```
sh fileinfo2.sh `ls`
```

The `trap` Command

- Shell scripts terminate when an interruption signal is received (such as through key press)
- Put “trap” statement before other shell commands inside the shell script
- The trap command allows you to control the way a program behaves instead of termination when it receives certain signals, such as the following:
  - Behave normally (the default action)
  - Ignore the signal
  - Do some cleanup (signal handling function) before exiting

```
trap 'command_list' signal_list
```

- Examples
  - Ignore: `trap "" 1 2` ←→ `trap "" HUP INT`
  - Do something: `trap ' rm tmp*; exit 1' 1 2 15`

When 1-SIGHUP or 2-SIGINT or 15- SIGTERM signal is received, the script will cleanup tmp files and exit

```
#!/bin/sh
# a bash script for testing “trap” command
trap "echo ignore interrupt " 2 3 15
num=$1
i=0
while [ "$i" -lt "$1" ]
do
   let i++
   echo I will take one minute nap...
   sleep 60
done
```

- To run the script:

  sh trap.sh NUM

- To terminate such process

  kill -9 PID

Shell Variable: Array

- **Index starts at zero**
- Created on the fly
  - **`names[0]=Jone; names[1]=Amy; names[2]=Alex`**
  - **`names=(Jone Amy Alex)`** # separated by space, NOT comma
- **`${names[1]}:`** Reference an element:
- **`${names[@]}:`** List all the elements of an array:
- **`${#names[@]}:`** The number of elements of an array
  - Here @ can be replaced with * → **`${#names[*]}:`**
- Length of a string: **`${#str}`**
  - **`echo ${#names[1]}`** ➔ **`3`**
  - **`echo $#names`** ➔ **`4 (default to the first element)`**
  - **`student="John"; echo ${#student}`** ➔ **`4`**
- Curly brackets are needed

Positional Parameters And Array

- **bash fruit.sh apple pear peach**

```
argvs=$@; echo opt1=${argvs[0]}, opt2=${argvs[1]}
→   opt1=apple pear peach, opt2=
argvs=$*; echo opt1=${argvs[0]}, opt2=${argvs[1]}
→   opt1=apple pear peach, opt2=
argvs=$#; echo opt1=${argvs[0]}, opt2=${argvs[1]}
→   opt1=3, opt2=
argvs="$@"; echo opt1=${argvs[0]}, opt2=${argvs[1]}
→   opt1=apple pear peach, opt2=
argvs="$*"; echo opt1=${argvs[0]}, opt2=${argvs[1]}
→   opt1=apple, opt2=pear
argvs=( $* ); echo opt1=${argvs[0]}, opt2=${argvs[1]}
→   opt1=apple, opt2=pear
argvs=( "$@" ); echo opt1=${argvs[0]}, opt2=${argvs[1]}
→   opt1=apple, opt2=pear
```

Read from File

- Read the whole file into array with “readarray”

```
readarray files < ls.txt
# read each line into array
read -r -a property <<< ${files[3]}
echo "length " ${#property[@]}
echo ${property[8]}
```

- Read line by line from file “ls.txt”

```
while read -r line
do
    echo $line
done < ls.txt
```