Shell Scripting

UDIT-Research CyberInfrastructure (IT-RCI)

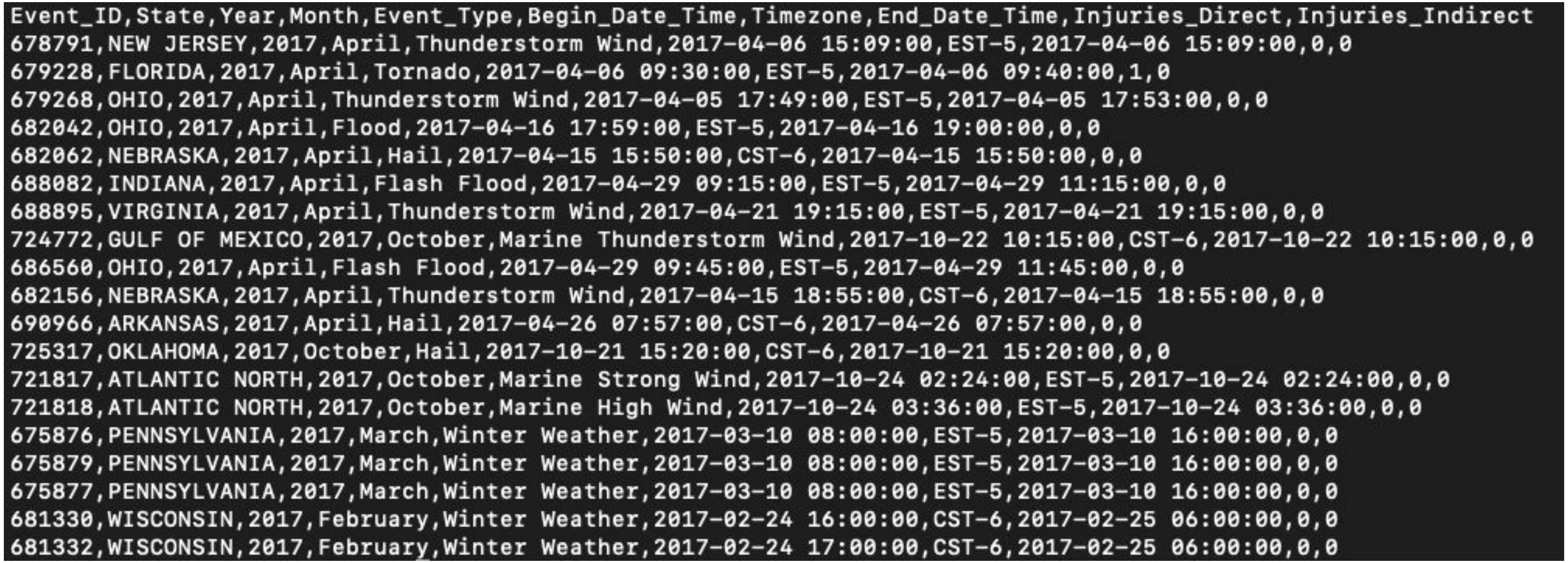
UNIVERSITY OF DELAWARE

Outline

- Shell
- Bash script layout
- Variables
- Bash constructs: conditions, loops
- Command substitution
- I/O redirection
- Subroutines
- Advanced substitution

Examples

- Data analysis directly from the shell



```
Event_ID,State,Year,Month,Event_Type,Begin_Date_Time,Timezone,End_Date_Time,Injuries_Direct,Injuries_Indirect
678791,NEW JERSEY,2017,April,Thunderstorm Wind,2017-04-06 15:09:00,EST-5,2017-04-06 15:09:00,0,0
679228,FLORIDA,2017,April,Tornado,2017-04-06 09:30:00,EST-5,2017-04-06 09:40:00,1,0
679268,OHIO,2017,April,Thunderstorm Wind,2017-04-05 17:49:00,EST-5,2017-04-05 17:53:00,0,0
682042,OHIO,2017,April,Flood,2017-04-16 17:59:00,EST-5,2017-04-16 19:00:00,0,0
682062,NEBRASKA,2017,April,Hail,2017-04-15 15:50:00,CST-6,2017-04-15 15:50:00,0,0
688082,INDIANA,2017,April,Flash Flood,2017-04-29 09:15:00,EST-5,2017-04-29 11:15:00,0,0
688895,VIRGINIA,2017,April,Thunderstorm Wind,2017-04-21 19:15:00,EST-5,2017-04-21 19:15:00,0,0
724772,GULF OF MEXICO,2017,October,Marine Thunderstorm Wind,2017-10-22 10:15:00,CST-6,2017-10-22 10:15:00,0,0
686560,OHIO,2017,April,Flash Flood,2017-04-29 09:45:00,EST-5,2017-04-29 11:45:00,0,0
682156,NEBRASKA,2017,April,Thunderstorm Wind,2017-04-15 18:55:00,CST-6,2017-04-15 18:55:00,0,0
690966,ARKANSAS,2017,April,Hail,2017-04-26 07:57:00,CST-6,2017-04-26 07:57:00,0,0
725317,OKLAHOMA,2017,October,Hail,2017-10-21 15:20:00,CST-6,2017-10-21 15:20:00,0,0
721817,ATLANTIC NORTH,2017,October,Marine Strong Wind,2017-10-24 02:24:00,EST-5,2017-10-24 02:24:00,0,0
721818,ATLANTIC NORTH,2017,October,Marine High Wind,2017-10-24 03:36:00,EST-5,2017-10-24 03:36:00,0,0
675876,PENNSYLVANIA,2017,March,Winter Weather,2017-03-10 08:00:00,EST-5,2017-03-10 16:00:00,0,0
675879,PENNSYLVANIA,2017,March,Winter Weather,2017-03-10 08:00:00,EST-5,2017-03-10 16:00:00,0,0
675877,PENNSYLVANIA,2017,March,Winter Weather,2017-03-10 08:00:00,EST-5,2017-03-10 16:00:00,0,0
681330,WISCONSIN,2017,February,Winter Weather,2017-02-24 16:00:00,CST-6,2017-02-25 06:00:00,0,0
681332,WISCONSIN,2017,February,Winter Weather,2017-02-24 17:00:00,CST-6,2017-02-25 06:00:00,0,0
```

Examples

- You performed 100 runs of a program
  - You varied two parameters a and b for each run
  - Each run is performed in a separate directory named `test_a_b`
  - For each run and in each directory, there are:
    - an input file named `input.txt`
    - an output file named output.txt
    - intermediate state files, named as `dump.step1`, `dump.step100`, etc.
- You need to:
  - save all the input and output files for record keeping purposes
    - name the files as `input_a_b.txt` and `output_a_b.txt`
  - extract certain data from the output file of each run for analysis, or benchmarking the run time

Examples

- Type similar commands manually in terminal for 100 times?

```
$ cp ./test22_6/input.txt ./data_bak/input_22_6.txt
$ cp ./test42_26/input.txt ./data_bak/input_42_26.txt
$ ...
```

- Write a script to perform the tasks

The Shell

What is a shell?

- a user program designed to read commands and execute programs.
- programming language interpreter
- features
  - defined variables
  - processes run in a shell
  - directory locations
  - shell levels
  - options

Bash Shell (bash)

- a widely used shell on Linux.
- different shells have different syntax and built-in functions.
- this workshop will be based on the bash shell.

bash: Bourne Again SHell

The Shell

find current shell type

```
$ echo $SHELL
/bin/bash

$ ps $$
   PID TTY      STAT   TIME COMMAND
 61651 pts/13   S      0:00 /bin/bash -il
```

find available shells on the system

```
$ cat /etc/shells
/bin/sh
/bin/bash
/usr/bin/sh
/usr/bin/bash
/bin/tcsh
/bin/csh
/usr/bin/tmux
```

What is shell scripting?

Shell Scripting - writing a series of commands in a text file

Benefits

automation	avoid repetitive typing
efficiency	combine multiple commands into a single workflow
reproducibility	easily re-run complex tasks with the same settings
customization	create your own tools tailored to your research needs

UNIVERSITY OF DELAWARE

An example

- The `stat` command display file status, write a bash script to display the file size and last access time

```
$ stat examplefile1
  File: ‘examplefile1’
  Size: 13              Blocks: 1            IO Block: 1048576 regular file
Device: 2dh/45d  Inode: 51877       Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 3241/   bdeng)   Gid: (  900/everyone)
Access: 2024-10-22 19:07:15.519228010 -0400
Modify: 2024-10-20 16:28:26.383164892 -0400
Change: 2024-10-20 16:28:26.383164892 -0400
 Birth: -
```

Bash script layout

```
# This script takes a file name as an argument
# and display the file's size and last access
# time
FILE="$1"

# Get the file's info using the stat command
FILE_SIZE=$(stat -c%s $FILE)
FILE_ACCESS_T=$(stat -c%x $FILE)

# Print the information
echo "File Name: $FILE"
echo "File Size: $FILE_SIZE bytes"
echo "Last accessed: $FILE_ACCESS_T"
```

display_file_info0.sh

- A plain text file with a collection of commands
- Each line of text in the script:
  - can have leading and trailing whitespace
  - is a comment if starts with #
  - contains all of or a portion of a shell command
    - can be continued on the next line if it ends with \

Bash script layout

- To execute the script as a bash script:

```
$ bash display_file_info0.sh examplefile1
File Name: examplefile1
File Size: 13 bytes
Last accessed: 2024-10-20 16:28:26.382164865 -0400
```

UNIVERSITY OF DELAWARE

Bash script layout

```
#!/bin/bash

# This script takes a file name as an argument
# and display the file's size and last access
# time
FILE="$1"

# Get the file's info using the stat command
FILE_SIZE=$(stat -c%s $FILE)
FILE_ACCESS_T=$(stat -c%x $FILE)

# Print the information
echo "File Name: $FILE"
echo "File Size: $FILE_SIZE bytes"
echo "Last accessed: $FILE_ACCESS_T"
```

display_file_info1.sh

- Adding shebang, “#!” (aka. sha-bang, sh-bang, etc.) at the head of a script to specify the interpreter
- To make the bash script executable:

```
$ ls -l display_file_info1.sh
-rw-r--r-- 1 bdeng everyone …

$ chmod +x display_file_info1.sh

$ ls -l display_file_info1.sh
-rwxr-xr-x 1 bdeng everyone …

$ ./display_file_info1.sh examplefile1
File Name: examplefile1
File Size: 13 bytes
Last accessed: 2024-10-20 16:28:26.382164865
-0400
```

Additional Shebang examples

```
#!/bin/sh
#!/usr/bin/python
#!/usr/bin/awk -f



#!/usr/bin/env bash
#!/usr/bin/env python
```

- Use other interpreters

- Another variant
  - /usr/bin/env search for bash or python in the user's environment, making it more portable

Variables

- A variables is a placeholder for its value
- Variables help generalize a program

```
#!/bin/bash

# This script takes a file name ...
FILE="$1"

# Get the file's info ...
FILE_SIZE=$(stat -c%s $FILE)
FILE_ACCESS_T=$(stat -c%x $FILE)

# Print the information
echo "File Name: $FILE"
echo "File Size: $FILE_SIZE bytes"
echo "Last accessed: $FILE_ACCESS_T"
```

Variables

- Use the assignment operator (=) to create a variable
    - **NO** space before or after
- Use ($) to dereference a variable
- Variable names are case sensitive

```
$ var1=apple
$ Var1=pineapple
$ echo $var1
apple
$ echo $Var1
pineapple
```

Scope of variables

- Ordinary variables
  - defined within a single shell and are not propagated to child shells, subshells or parent shells
- Exported variables
  - becomes part of the environment of current shell
  - extended to any child processes, including child shells and subshells
- When running a bash script, it’s running in its own environment in a subshell

Scope of variables

```
#!/bin/bash
echo
echo $VAR_P1 $VAR_P2
VAR_C1=23
export VAR_C2=24
sleep 60
```

program1.sh

```
$ VAR_P1=notExported
$ export VAR_P2=exported
$ ./program1.sh &
[1] 16868
$
exported
$ echo $VAR_C1 $VAR_C2

$ ps --forest
   PID TTY          TIME CMD
122438 pts/16   00:00:01 bash
 16868 pts/16   00:00:00  \_ program1.sh
 16869 pts/16   00:00:00  |   \_ sleep
 16992 pts/16   00:00:00  \_ ps
```

Executing script in current shell

- Script can also be executed in the current shell without starting a new process with its own environment using the `source` command or the dot (`.`) operator

```
$ ./display_file_info1.sh examplefile1
File Name: examplefile1
File Size: 13 bytes
Last accessed: 2024-10-20 16:28:26.382164865 -0400
$ echo $FILE

$ source display_file_info1.sh examplefile1
File Name: examplefile1
File Size: 13 bytes
Last accessed: 2024-10-20 16:28:26.382164865 -0400
$ echo $FILE
examplefile1

$ . display_file_info1.sh examplefile1
```

Subshells

- Surround one or more commands with parenthese
  - the commands will be executed in a subshell

```
#!/bin/bash
VAR_C1=23
(export VAR_C2=24; VAR_C3=25)
echo $VAR_C1 $VAR_C2 $VAR_C3
```

program2.sh

```
$ ./program2.sh
23
```

Variables - Numeric

- By default, bash variables are strings

```
$ a=3
$ b=a+1
$ echo $b
a+1
$ c=$a+1
$ echo $c
3+1
```

- The `declare` command can be used to modify a variable property as integer (`declare -i`) or other types
  - arithmetic operations can be performed on integer variables

```
$ declare -i x
$ declare -i y
$ x=3
$ y=x+1
$ echo $y
5
```

- Variables NOT declared as integer type are converted during arithmetic evaluation

```
$ ax1=a+x
$ echo $ax1
a+x
$ ax2=$((a+x))
$ echo $ax2
7
```

arithmetic operators	
+	addition
–	subtraction
*	multiplication
/	division
%	modulo
**	exponentiation

Special variables

- Positional parameters
  - arguments passed to the script from the command line: `$1`, `$2`, `$3`… `$9`
  - after `$9` use brackets: `${10}`, `${11}`
  - `$0` is the script itself
  - the `shift` command reassigns the positional parameters
    - `$1` ← `$2`, `$2` ← `$3`, `$3` ← `$4`, etc.
    - `$0` does not change

Special variables

- Additional special variables

`$*`	all command line arguments	`$?`	exit status of last program
`$@`	all command line arguments	`$$`	pid of this program
`$#`	number of command line arguments	`$!`	pid of last-started background job
`$_`	starts as full path to script, changes to last command argument list		

`$*` and `$@` behave differently when inside double quotes

```
#!/bin/bash

echo "The full path to this script is $_"
echo "This script is named $0 and is running with pid $$"
echo "You provided $# arguments"

echo "What does \$ equal now: $_"

echo "The first two arguments are $1 and $2"

echo "The full argument list:  $@"

echo "A printf with \$*:"
printf "  %s\n" "$*"

echo "A printf with \$@:"
printf "  %s\n" "$@"
```

program3.sh

```
./program3.sh "a b c" "d e f" g h i
The full path to this script is
./program3.sh
This script is named ./program3.sh and is
running with pid 25156
You provided 5 arguments
What does $ equal now: You provided 5
arguments
The first two arguments are a b c and d e f
The full argument list:  a b c d e f g h i
A printf with $*:
  a b c d e f g h i
A printf with $@:
  a b c
  d e f
  g
  h
  i
```

Quoting

- In Bash, Quotes are used to define string literals
- Single Quotes (' ')
  - everything is considered as string literal
- Double Quotes ("   ")
  - everything except special characters (e.g. $, \)
- Escape(\)
  - quoting single characters, the character after \ is treated literally

```
#!/bin/bash
name=Alice

echo 'Hello, $name'

echo "Hello, $name"

echo "She said, \"Hello!\""
```

What's the output?

```
Hello, $name
Hello, Alice
She said, "Hello!"
```

Quoting

Feature	Single Quotes (')	Double Quotes (")
Interpretation	Literal (no expansion)	Variable expansion & command substitution
Special Characters	Treated literally	Interpreted (e.g., $, \, `)
Escape Characters	Literal	Can escape certain characters

Quoting - additional notes

- Type the last echo command in the previous example directly in command line will result in error
  - " ! " is interpreted as a history command in command line, but disabled in bash

```
$ echo "She said, \"Hello!\""
-bash: !\"": event not found
```

- How to create the string:
  - Alice’s dog’s name is Hachi

```
$ echo 'Alice's dog's name is Hachi'
Alices dogs name is Hachi

$ echo 'Alice'\''s dog'\''s name is Hachi'
Alice's dog's name is Hachi
```

if constructs

- General formats

```
if expr
then
    actions
fi
```

```
if expr; then
    actions
fi
```

```
if expr; then
    actions
else
    actions
fi
```

```
if expr; then
    actions
elif expr2; then
    actions
elif expr3; then
    actions
else
    actions
fi
```

```
if expr; then
    actions
else
    if expr2; then
        actions
    elif expr3; then
        actions
    fi
fi
```

```
if expr; then actions; fi
```

Evaluating the conditional expression

- The if constructs tests the **exit status** of an expression that evaluate to 0 = true (non-0 = false)
  - In Linux, every command returns an exit status
    - a successful command returns a 0
    - an unsuccessful one returns a non-0 value, an error code
  - Most programming languages equate 0 with false, and non-0 with true

```
#!/bin/bash

x=0

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

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

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

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

Conditionals - tests

- `[ ]`
  - alias for the `test` command
  - spaces are required around the operators

```
if [ "$x" -gt 5 -a "$y" -lt 10 ]; then …
```

- `[[ ]]`
  - extended test. Introduced in newer versions of Bash
  - spaces are required around the operators
  - supports additional string and arithmetic operations, logical operators (`&&, ||`)

```
if [[ $x > 5 && $y < 10 ]]; then …
```

Arithmetic tests using (( ))

- The `((    ))` construct expands and evaluates an arithmetic expression.
  - If the expression **evaluates** as zero, it returns an **exit status** of 1, or "false".
  - A non-zero expression returns an exit status of 0, or "true".

```
$ (( 0 ))
$ echo $?
1

$ (( 1 ))
$ echo $?
0
```

```
$ if (( 0 )); then echo "true"; else echo "false"; fi
false
$ if (( 1 )); then echo "true"; else echo "false"; fi
true
```

```
$ if (( 5 > 2 )); then echo "true"; else echo "false"; fi
true
$ if (( 5 - 3 > 2 )); then echo "true"; else echo "false"; fi
false
```

UNIVERSITY OF DELAWARE

Exercise

if `aabbcc` is not an environmental variable or command?

What happens when executing the following scripts?

```
#!/bin/bash

if aabbcc; then
    echo "condition is True"
else
    echo "condition is False"
fi
```

```
#!/bin/bash

if [ aabbcc ]; then
    echo "condition is True"
else
    echo "condition is False"
fi
```

```
$ ./prog4_1.sh
./prog4_1.sh: line 3: aabbcc: command not
found
condition is False
```

```
$ ./prog4_2.sh
condition is True
```

What to compare

string comparison

`a = b`	equal
`a != b`	not equal
`a > b`	greater than (lexicographically)
`a < b`	less than (lexicographically)

integer comparison

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

What to compare

file test operators

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

logical operators

expr1 `-a` *expr2*	AND
expr1 `-o` *expr2*	OR
`!`*expr1*	logical negation
`(`*expr1*`)`	compound grouping

Conditionals

- Add a condition to check if the file exist
  - if file exist, display its information
  - otherwise, print error message and exit

display_file_info2.sh

```
#!/bin/bash

# This script takes a file ...
FILE="$1"

if [ -f "$FILE" ]; then
    # Get the file's info ...
    FILE_SIZE=$(stat -c%s $FILE)
    FILE_ACCESS_T=$(stat -c%x $FILE)

    # Print the information
    echo "File Name: $FILE"
    echo "File Size: $FILE_SIZE bytes"
    echo "Last accessed: $FILE_ACCESS_T"
else
    echo "Error: File $FILE does not exist."
    exit 1
fi
```

```
$ ./display_file_info2.sh examplefile1
File Name: examplefile1
File Size: 13 bytes
Last accessed: 2024-10-20
16:28:26.382164865 -0400

$ ./display_file_info2.sh examplefile2
Error: File examplefile2 does not exist.
```

case statements

- Many-valued branch table
  - more concise, clear than a lengthy if..elif..else..fi

```
case "$variable" in
    pattern1)
        # Code block for pattern1
        ;;
    pattern2)
        # Code block for pattern2
        ;;
    patternN)
        # Code block for patternN
        ;;
    *)
        # Default case (optional)
        ;;
esac
```

```
#!/bin/bash
echo "Enter a character:"
read char
case "$char" in
    [a-z])
        echo "You entered a lowercase letter."
        ;;
    [A-Z])
        echo "You entered an uppercase letter."
        ;;
    [0-9])
        echo "You entered a digit."
        ;;
    *)
        echo "You entered a special character."
        ;;
esac
```

Loops

- Loops allow a sequence of statements to be executed zero or more times (as long as the loop control condition is true)
    - iterate over a set of items
    - iterate a fixed number of times
    - iterate until a condition is satisfied

FOR loops

- General format

```
for arg in [list]
do
  actions
done
```

```
for arg in [list]; do
  actions
done
```

```
for arg in [list]; do actions; done
```

- C-style constructs

```
for (( initialization; condition; increment/decrement ))
do
    # commands to execute
done
```

Loops - Set of items

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

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

Loops - Set of items

- display the file information for more than one input files

```
#!/bin/bash
# display file info of input files

for FILE in "$@"; do
  if [ -f "$FILE" ]; then
    # Get the file's info ...
    FILE_SIZE=$(stat -c%s $FILE)
    FILE_ACCESS_T=$(stat -c%x $FILE)

    # Print the information
    echo "File Name: $FILE"
    echo "File Size: $FILE_SIZE bytes"
    echo "Last accessed: $FILE_ACCESS_T"
  else
    echo "Error: File $FILE does not exist."
  fi
  echo
done
```

display_file_info3.sh

```
$ touch examplefile3
$ ./display_file_info3.sh examplefile1 examplefile2
examplefile3
File Name: examplefile1
File Size: 13 bytes
Last accessed: 2024-10-21 09:28:40.977299083 -0400

Error: File examplefile2 does not exist.

File Name: examplefile3
File Size: 0 bytes
Last accessed: 2024-10-21 11:54:11.607417810 -0400
```

Brace expansion

- Generate a set of strings
  - simple list: `{item1,item2,item3}`
  - a sequence: `prefix{x..y..z}suffix`
    - `x`,`y` are integers or single characters
    - `z` is an optional increment

```
$ echo {apple,orange}
apple orange
$ mkdir project/{src,bin,docs}
$ echo my{in,out}put.txt
myinput.txt  myoutput.txt
$ echo {1..5}
1 2 3 4 5
$ echo {1..5..2}
1 3 5
$ echo {5..1}
5 4 3 2 1
$ echo {a..d}
a b c d
$ echo {a..d..2}
```

```
$ for i in {1..3};do filename=run$i.input; echo $filename; done
run1.input
run2.input
run3.input
```

WHILE loop

- General format

```
while [ condition ]
do
  actions
done
```

```
while [ condition ]; do
  actions
done
```

```
while [ condition ]; do actions; done
```

```
while ((condition))
do
  actions
done
```

WHILE loop

```
#!/bin/bash

counter=1
while [ $counter -le 5 ]
do
    echo "Counter: $counter"
    (( counter++ ))  # Increment the counter
done
```

- loop run a set number of times

```
#!/bin/bash

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

- loop can also run indefinitely or until a condition is met

WHILE loop

```
#!/bin/bash

file="input.txt"

# Check if the file exists
if [[ ! -f $file ]]; then
    echo "File $file not found!"
    exit 1
fi

# Read the file line by line using a while loop
while IFS= read -r line; do
    # Process each line
    echo "Read line: $line"
done < "$file"
```

- use while loop to read a text file line by line

UNTIL loop

- Exit loop when expression is true

```
#!/bin/bash

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

Command Substitution

- Reassigns the output of command(s)
- Two types of syntax:
  - `$( command )`
    - recommended
    - easy to nest substitution
  - `` `command` ``
    - must escape certain characters
    - hard to nest substitution

```
$ echo ls
ls

$ $(echo ls)
display_file_info0.sh  display_file_info2.sh  examplefile1
display_file_info1.sh  display_file_info3.sh  examplefile3
```

UNIVERSITY OF DELAWARE

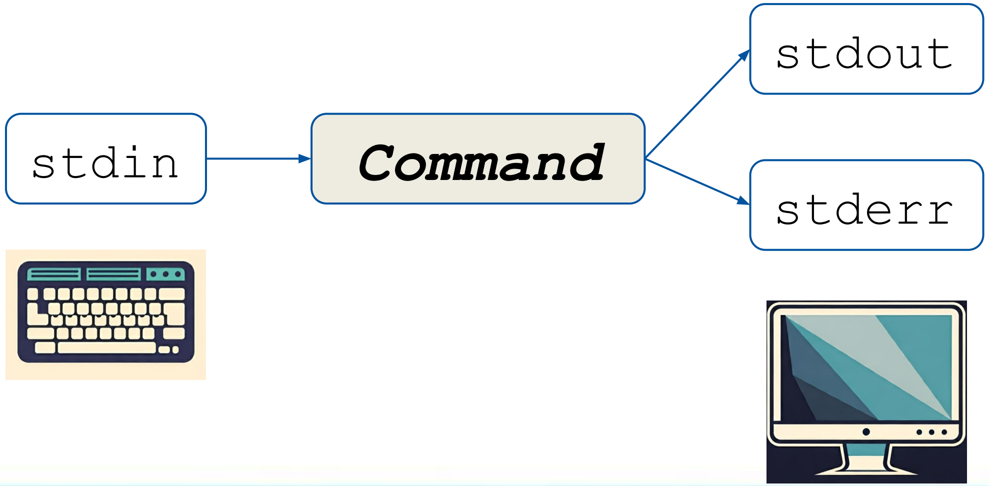
```
$ cat tests_list.txt
Test18
Test88
Test96
Test05
$ mkdir $(cat tests_list.txt)
$ ls -ld *
drwxr-xr-x 2 bdeng everyone  2 Oct 22 19:35 Test05
drwxr-xr-x 2 bdeng everyone  2 Oct 22 19:35 Test18
drwxr-xr-x 2 bdeng everyone  2 Oct 22 19:35 Test88
drwxr-xr-x 2 bdeng everyone  2 Oct 22 19:35 Test96
-rw-r--r-- 1 bdeng everyone 49 Oct 22 19:28 tests_list.txt

$ echo $(ls -s $(cat tests_list.txt))
Test05: total 0 Test18: total 0 Test88: total 0 Test96: total 0
```

```
$ mkdir `cat tests_list.txt`
$ echo `ls -s \`cat tests_list.txt\``
```

I/O redirection review

- Three communication channels for a process
  - stdin - standard input - file descriptor: 0
  - stdout - standard output - file descriptor: 1
  - stderr - standard error - file descriptor: 2

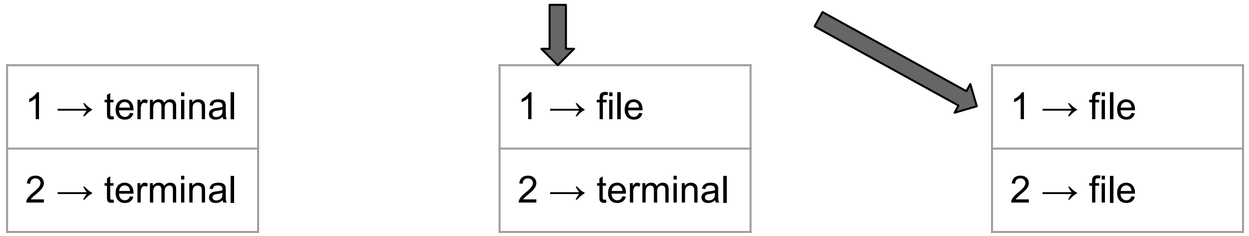


I/O redirection review

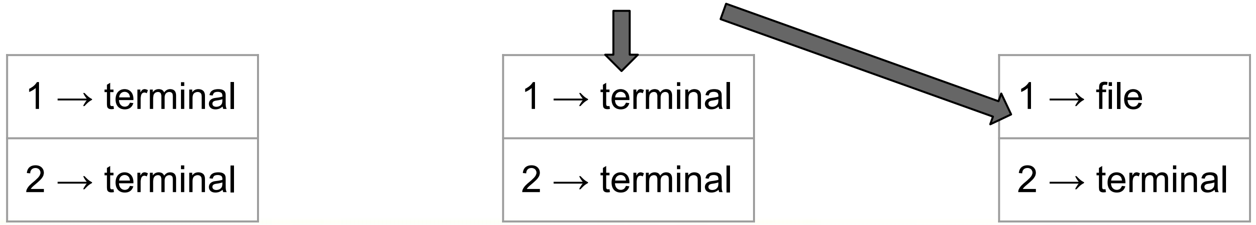
- redirecting stdin
  - `command < file`
- redirecting stdout to a file
  - `command > file`
- redirecting stderr to a file
  - `command 2> file`
- redirecting stdout and stderr
  - `command &> file`
  - `command > file 2>&1`
  - `command > file 2> errfile`
- redirecting all
  - `command < infile > file 2> errfile`

I/O redirection review

- Use >> for appending operation
- Order is important
  - `command > file 2>&1`

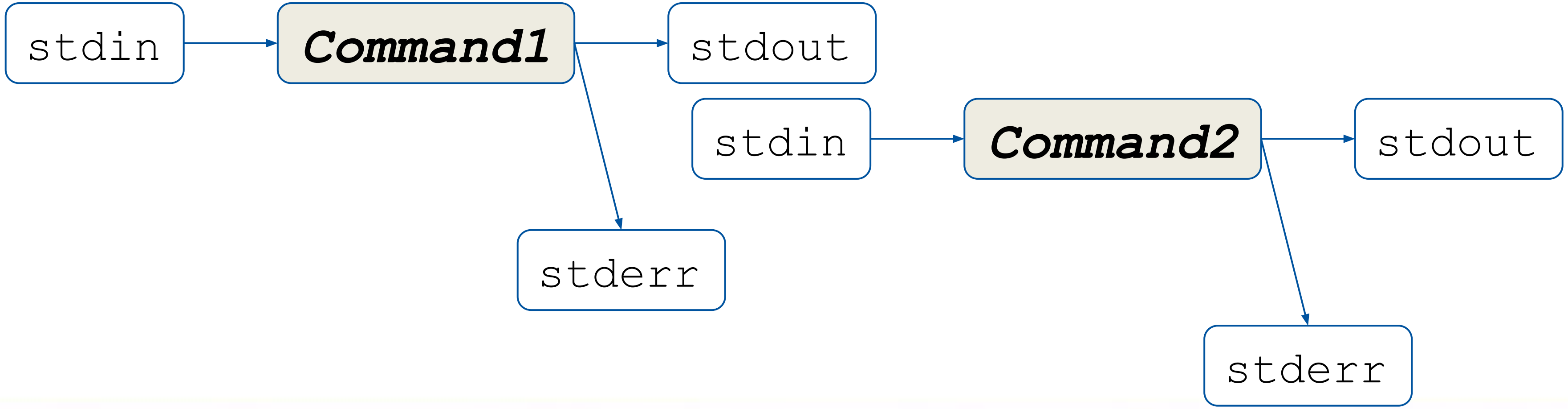


  - `command 2>&1 > file`



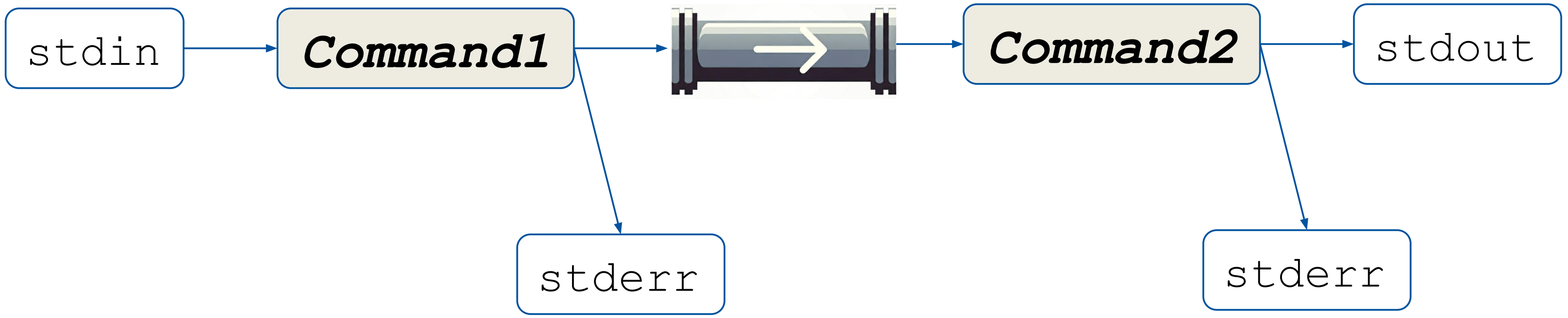
Pipes

- Use output of one program as input for another
  - write to a temporary file: `command > tmpfile`
  - read from the temporary file: `command2 < tmpfile`
  - remove the temporary file: `rm tmpfile`



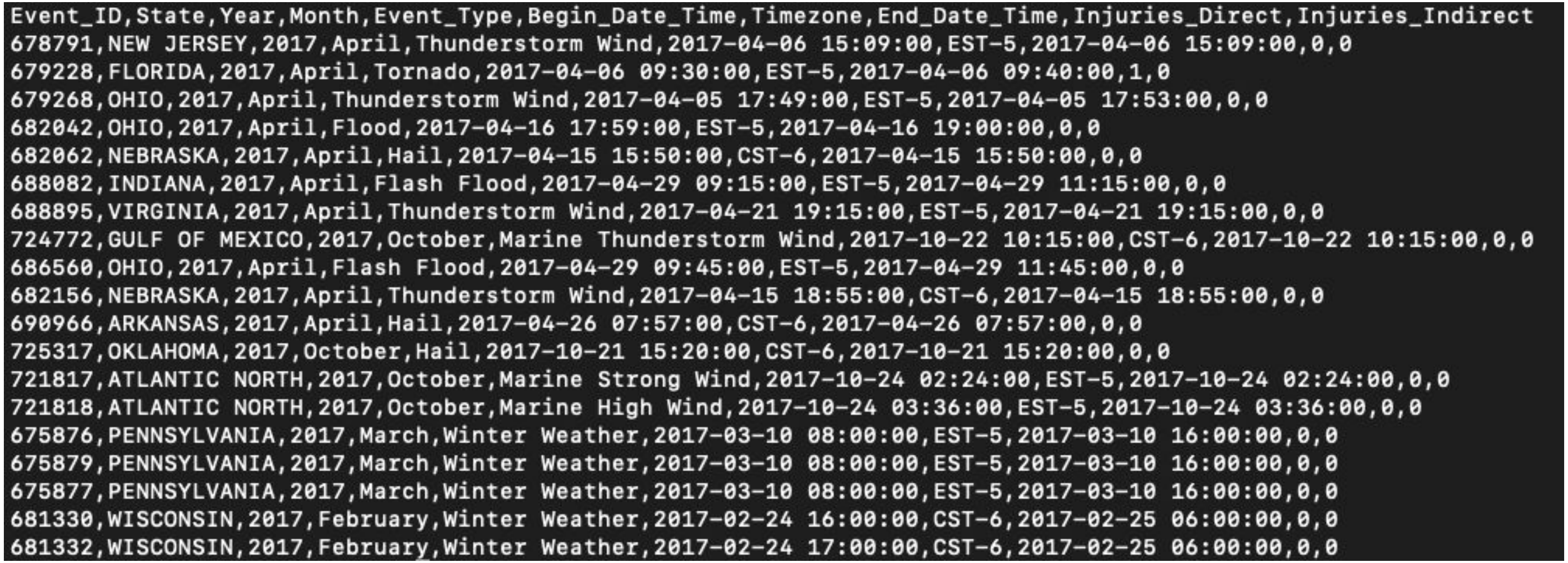
Pipes

- Use |
  - *command1 | command2*
  - No temporary files written to disk



```
$ ls | grep "display_file"
```

- e.g. Storm related events in 2017 saved in a csv file StormEvents_2017.csv
    - perform some quick data analysis



```
Event_ID,State,Year,Month,Event_Type,Begin_Date_Time,Timezone,End_Date_Time,Injuries_Direct,Injuries_Indirect
678791,NEW JERSEY,2017,April,Thunderstorm Wind,2017-04-06 15:09:00,EST-5,2017-04-06 15:09:00,0,0
679228,FLORIDA,2017,April,Tornado,2017-04-06 09:30:00,EST-5,2017-04-06 09:40:00,1,0
679268,OHIO,2017,April,Thunderstorm Wind,2017-04-05 17:49:00,EST-5,2017-04-05 17:53:00,0,0
682042,OHIO,2017,April,Flood,2017-04-16 17:59:00,EST-5,2017-04-16 19:00:00,0,0
682062,NEBRASKA,2017,April,Hail,2017-04-15 15:50:00,CST-6,2017-04-15 15:50:00,0,0
688082,INDIANA,2017,April,Flash Flood,2017-04-29 09:15:00,EST-5,2017-04-29 11:15:00,0,0
688895,VIRGINIA,2017,April,Thunderstorm Wind,2017-04-21 19:15:00,EST-5,2017-04-21 19:15:00,0,0
724772,GULF OF MEXICO,2017,October,Marine Thunderstorm Wind,2017-10-22 10:15:00,CST-6,2017-10-22 10:15:00,0,0
686560,OHIO,2017,April,Flash Flood,2017-04-29 09:45:00,EST-5,2017-04-29 11:45:00,0,0
682156,NEBRASKA,2017,April,Thunderstorm Wind,2017-04-15 18:55:00,CST-6,2017-04-15 18:55:00,0,0
690966,ARKANSAS,2017,April,Hail,2017-04-26 07:57:00,CST-6,2017-04-26 07:57:00,0,0
725317,OKLAHOMA,2017,October,Hail,2017-10-21 15:20:00,CST-6,2017-10-21 15:20:00,0,0
721817,ATLANTIC NORTH,2017,October,Marine Strong Wind,2017-10-24 02:24:00,EST-5,2017-10-24 02:24:00,0,0
721818,ATLANTIC NORTH,2017,October,Marine High Wind,2017-10-24 03:36:00,EST-5,2017-10-24 03:36:00,0,0
675876,PENNSYLVANIA,2017,March,Winter Weather,2017-03-10 08:00:00,EST-5,2017-03-10 16:00:00,0,0
675879,PENNSYLVANIA,2017,March,Winter Weather,2017-03-10 08:00:00,EST-5,2017-03-10 16:00:00,0,0
675877,PENNSYLVANIA,2017,March,Winter Weather,2017-03-10 08:00:00,EST-5,2017-03-10 16:00:00,0,0
681330,WISCONSIN,2017,February,Winter Weather,2017-02-24 16:00:00,CST-6,2017-02-25 06:00:00,0,0
681332,WISCONSIN,2017,February,Winter Weather,2017-02-24 17:00:00,CST-6,2017-02-25 06:00:00,0,0
```

```
$ # find events in Delaware
$ $ grep DELAWARE StormEvents_2017.csv | head -n 3
120480,722023,DELAWARE,2017,October,Strong Wind,2017-10-30 ...
111521,665521,DELAWARE,2017,January,Winter Storm,2017-01-07 ...
111521,665522,DELAWARE,2017,January,Winter Storm,2017-01-07 …

$ # find events in Delaware in July
$ grep DELAWARE StormEvents_2017.csv | grep July | head -n 3
117045,704171,DELAWARE,2017,July,Heavy Rain,2017-07-23 ...
117049,704202,DELAWARE,2017,July,Heavy Rain,2017-07-29 ...
116251,701615,DELAWARE,2017,July,Heavy Rain,2017-07-07 …

$ # count unique events
$ grep -v "EpisodeID" StormEvents_2017.csv | cut -d',' -f6 StormEvents_2017.csv | sort | uniq -c |
head
     13 Astronomical Low Tide
     27 Avalanche
    356 Blizzard
...
```

Pipes

- Duplicate output across two files
  - `command1 | tee file`
  - log output, debugging intermediate results

```
$ ls | tee list.txt | grep "display_file"


$ make 2>&1 | tee build.log
```

Pipes

- Special files to use in redirection

`/dev/null`	Discards all data written to it. Reading from it produces nothing (EOF).
`/dev/zero`	Produces an endless stream of null (zero) bytes. Used to create zero-filled files or initialize memory.
`/dev/random` `/dev/urandom`	provides random data
`/dev/tty`	Refers to the current terminal.

- Named pipes

Arrays

- Newer versions of bash support 1d arrays
- `declare -a arrayname`
- Can store both strings and numbers
- Each element can be accessed using an index, starting from 0
  - `${arrayname[index]}`
- delete array
  - delete an element: `unset arrayname[index]`
  - delete the entire array: `unset arrayname`
- getting the number of elements:

```
$ fruits=("apple" "banana")
$ echo ${fruits[0]}
apple
$ echo ${fruits[2]}

$ fruits[2]="cherry"
$ echo ${fruits[2]}
cherry
$ echo ${fruits[*]}
apple banana cherry
```

Arrays

```
$ fruits=("apple" "banana")
$ echo ${fruits[0]}
apple
$ echo ${fruits[2]}

$ fruits[2]="cherry"
$ echo ${fruits[2]}
cherry
$ echo ${fruits[*]}
apple banana cherry
```

```
$ for w in "${fruits[@]}";do echo $w;done
apple
banana
cherry
```

- loop over array values
  - can also loop over array indices

Arrays

```
fruits[3]="cherry pie"
$ for w in ${fruits[@]};do echo $w;done
apple
banana
cherry
cherry
pie

for w in "${fruits[@]}";do echo $w;done
apple
banana
cherry
cherry pie
```

Subroutines

```
#!/bin/bash
# Function to calculate the square of a number
square() {
    local num=$1
    if (( num < 10 )); then
      echo $(( num * num ))
      return 0
    else
      return 1
    fi
}

# Main part of the script
# Call the square function and store the result
num=$1
result=$(square $num)

if [ $? -eq 0 ]; then

    echo "The square of $num is: $result"
else
    echo "unable to compute"
fi
```

- encapsulate often-used command sequences in a sub-program

- Used `echo` to return actual values
- Used `return` to return exit status

```
$ ./func2.sh 4
The square of 4 is: 16
$ ./func2.sh 44
unable to compute
```

Advanced expansion

- Variable names can also be enclosed within curly braces, e.g. `${VARNAME}`
- Curly braces allow for additional logic and transformation w.r.t. the variable’s value

Advanced expansion

```
$ echo $PATH
/home/3241/.local/bin:/home/3241/bin

$ VAR=PATH

$ echo ${!VAR}
/home/3241/.local/bin:/home/3241/bin
```

- Indirect expansion
- the value of $VAR is itself the name of a variable
- substitute the value of that variable

```
$ VERBOSE=1
$ VARIABLE=1
$ echo ${!V*}
VAR VARIABLE VERBOSE
$ echo ${!VAR*}
VAR VARIABLE
```

- Names of variables whose name start with “VAR”

Advanced expansion examples

- `${parameter:-default}`
  - Expands to default if parameter is unset or null.
- `${parameter:?message}`
  - Assigns default to parameter if it’s unset or null.
- `${parameter:offset:length}`
  - Extracts a substring starting at offset and up to length characters.

```
$ name=""
$ echo ${name:-"Guest"}
Guest

$ unset name
$ echo ${name:?"Variable name is not set"}
-bash: name: Variable name is not set

$ str="Hello World"
$ echo ${str:6:5}
World
```

When not to use shell scripts

- Resource-intensive tasks
- Heavy-duty math operations
- native support for multi-dimensional arrays
- need data structures
- need graphics
- Cross-platform portability required
- Extensive file operations required
- proprietary, closed-source applications

Upcoming Workshops

- Upcoming workshops
  - HPC Software Development and Installation: 10/31 @ 10 am-noon
- Past workshops
  - Introduction to Linux
  - Introduction to HPC
  - Getting Started with DARWIN
  - Introduction to Slurm
  - Shell Scripting

Office Hours

- Research Computing Office Hours
  - Time: Tuesdays and Wednesdays @ 11am – noon
  - Location: 002C Smith Hall, Conference Room or via Zoom.
- Research Software Engineering (RSE) Support For The College Of Arts And Science (CAS)
  - Time: Tuesdays and Wednesdays @ 1pm – 2pm
  - Location: Room 110 Sharp Lab or Zoom

UNIVERSITY OF DELAWARE

UNIVERSITY OF DELAWARE

Thank you!