Intro
What is a Script?

- A sequence of Bash commands
- A Bash program
- Stored as a text file
  - *Interpreted* by the Bash shell
Why Write Scripts

- Convenience
- Sequences of oft-performed operations can be placed in a script, executed as a single command
- Shell provides access to many useful utilities
- I have scripts for simple tasks, and scripts for some fairly complex tasks:
  - Rename all files of the form DSCnnnn.JPG to nnnn.jpeg
  - Organise the flat directory of heavily decorated filenames downloaded from Blackboard into subdirectories by student, restoring the original filenames
  - A testing framework, for grading student programs. Calls individual test cases (themselves scripts) for each program to grade
Shells are built for comfort. This comes at a cost
Shell scripts are generally not as well-suited to large tasks
  • They run more slowly, are more resource-intensive
  • Scripts do not give the programmer nearly the same control over resources
Languages such as C allow for much more structured programs
Shell scripts are more difficult to harden against security attacks
Hello, Script!
Consider the following file, hello

```
#!/bin/bash
echo "Hello, $USER"
exit 0 # success
```

- `#!/bin/bash` – sha-bang
  - First line
  - Identifies the interpreter who is to execute this script
- *Always* quote variables in scripts
  - If you don’t think you need to, *quote it anyway!*
- `exit 0` – The return value (status) of the script
- `# success` – Line comment
Running a Script

- Can be given explicitly to Bash as a simple input text file:

  ```bash
  $ bash hello
  Hello, kschmidt
  ```

- Or, we can give it execute permissions, run it as any other utility:

  ```bash
  $ chmod +x hello
  $ hello
  hello: command not found
  ```

- Whoops! The current directory isn’t in my PATH (nor should it be). Tell the shell where it is:

  ```bash
  $ ./hello
  Hello, kschmidt
  ```
Tests for Branches and Loops

We need tests for branches and loops

- We’ve already seen the return value of commands
  - Zero (0) is `true` (success)
  - All else is false (signals some failure)
  - Can be negated using `!` before

- There are special utilities and Bash built-ins to provide various tests
  - `test`, `[ ]` Provides string, numeric, and file tests
  - `[[ ]]` Similar to `[ ]`, but gentler syntax
  - `let`, `(( ))` Provides numeric tests and arithmetic
[ expr ]

- Built into Bash
  - But behaves like the disk utility (less than pretty)
  - Only such test available in Bourne shell

- Note, the spaces around the [ ] are necessary

- Provides:
  - String tests
  - File tests
  - Numeric tests
  - Logical operators
We have the normal binary, relational operators

\(< \ = \ != \ >\)

```bash
$ [ a = b ] || echo false
false
$ [ jaga < kurt ] && echo true
bash: kurt: No such file or directory
```

Whoops! `<` is a shell metacharacter. Needs to be escaped

```bash
$ [ jaga \< kurt ] && echo true
ture
```
We have unary tests for strings

- `z` True if string is empty
- `n` True if string is not empty

```
[ -z "$1" ] && echo "Script requires a filename as an argument"
[ -n "$1" ] || echo "Script requires a filename as an argument"
```
Many unary tests for files. Here are a few:

- `e file` True if file exists
- `d file` True if file is a directory
- `f file` True if file is a regular file
- `L file` True if file is a symbolic link
- `r file` True if file is readable by you
- `w file` True if file is writable by you
- `x file` True if file is executable by you
- `O file` True if file is effectively owned by you

\[1\] Do a help test, or man test for more
There are some binary operators for files:

\[ f1 \ -nt \ f2 \quad \text{True if } f1 \text{ is newer than } f2 \]
\[ f1 \ -ot \ f2 \quad \text{True if } f1 \text{ is older than } f2 \]
\[ f1 \ -ef \ f2 \quad \text{True if } f1 \text{ is a hard link to } f2 \text{ (they are the same file)} \]

```bash
[ -f "$log" ] && echo "Next status line" >> "$log"
[ -r "$input" ] || echo "I can't read $input"
[ "$f1" -ef "$f2" ] &&
  echo "I can remove $f1 and $f2 will still be there"
```
We have different relational operators for arithmetic\(^1\):

- All parameter values are just strings
- Shell can’t tell from context which comparison is meant

\[-lt \, -le \, -eq \, -ne \, -ge \, -gt\]

\[
\begin{align*}
\$ & \ [ \ 13 \ \text{\textless} \ 2 \ ] \ \&\& \ echo \ true \\
& true \\
\$ & \ [ \ 13 \ \text{-lt} \ 2 \ ] \ |\| \ echo \ false \\
& false
\end{align*}
\]

\(^1\) The `let` utility makes this prettier
- Logical Operators

\[ \neg \ expr \quad \text{NOT - True when } expr \text{ is false, false otherwise} \]

\[ \text{exp1} -a \text{ exp2} \quad \text{AND - True when both } exp1 \text{ and } exp2 \text{ are true, false otherwise} \]

\[ \text{exp1} -o \text{ exp2} \quad \text{OR - False when both } exp1 \text{ and } exp2 \text{ are false, true otherwise} \]
- Supports all the same tests as `[ ]`
- Is a built-in, so, syntax is gentler
  - Shell metacharacters `<`, `>`, etc., don’t need to be escaped
  - Shell knows it’s in a test
- Mind, install scripts (and makefiles) tend to use Bourne syntax; it’s the default for all Unix systems
Familiar logical operators

- !
- &&
- ||
- == = both test for equality
- == != treat the right operand as a pattern (glob)

```bash
$ [[ abcde.f == a*e.? ]] && echo true
true
```

New operator, =~, treats the right operand as an extended regular expression

```bash
$ [[ abcde.f =~ a.*e\..? ]] && echo true
true
```
let is a Bash built-in

Bash provides syntactic sugar: `(( ))`\(^1\)

Treats values stored in parameters as integers\(^2\)

- *N.B.* – Only does integer arithmetic (division)

Allows you to evaluate relational expressions

- Same logical operators

\[
< \quad <= \quad == \quad != \quad >= \quad >
\]

```
$ x=13 ; y=87
$ (( x > 7 )) && echo true
true
$ (( x!=0 && y/x >= 6 )) && echo true
true
```

\(^1\)Note, the `$[ ]$` form is deprecated

\(^2\)For float arithmetic, see the `bc` utility
let can be used to evaluate arithmetic exceptions

- Arithmetic: `**` `*` `/` `%` `+` `-`
- Bit-wise: `~` `<<` `>>` `^` `&` `|`
- Pre- and post-fix increment/decrement: `++` `--`
- A C-like ternary operator: `?:`
- Assignment `=`, and the usual operator/assignment operators: `+=` `-=` `&=`, etc.
$ x=13
$ echo $(( x+15 ))
28
$ echo $x
13
$ (( y = x*4 ))
$ echo $y
52
$ (( y-=1 ))
$ echo $y
51
$ echo $((x>>2))
3
$ (( 5 && 2 )) && echo true
true
$ (( 5 & 2 )) || echo false
false
$ # Same as in C, Python, etc. Why?
Integer Literals

Be careful. Try the following out.

```bash
$ echo $(( 012 ))
10
$ echo $(( 0x23 ))
35
```

This is not an odd artifact of Bash, nor of `let`. You will find similar behavior in C, AWK, Python2, Java, etc.
Control Flow
Bash Control Structures

We have branching:

- if
- if-else
- if-elif-else
- case

And loops:

- while
- until
- for
- select

**Note:** Bash may not care about proper indenting, but your grade may well care
if

if tests; then cmds; fi

- *tests* is executed
- If the exit status is 0 (success), *cmds* is executed

```bash
if grep Waldo * &> /dev/null ; then
echo "Found Waldo!"
fi
```

```bash
if [[ -d "$paris" && -r "$paris" ]]; then
 echo "I see "$paris"!
fi
```

```bash
if (( cats > 3 )); then
 echo "Too many cats"
 echo "People will talk"
fi
```
if tests; then cmds; else cmds; fi

if grep Waldo * &> /dev/null ; then
    echo "Found Waldo!"
else
    echo "Dude’s a slippery one"
fi

if [[ -d "$paris" && -r "$paris" ]]; then
    echo "I see $paris’!
else
    echo "Might be on the wrong continent"
fi

if (( cats > 3 )); then
    echo "Too many cats"
    echo "People will talk"
else
    echo "You might yet be sane"
fi
if tests; then cmds; {elif cmds;} else cmds; fi

read grade
if (( grade >= 90 )); then
echo "A"
elif (( grade >= 80 )); then
echo "B"
elif (( grade >= 70 )); then
echo "C"
elif (( grade >= 60 )); then
echo "D"
else
 echo "F"
fi
while Loop

while tests; do cmds; done

- tests is executed
- If the exit status is 0 (success), cmds is executed
- Execution returns back to tests, start again

```bash
i=0
while (( i<=12 )); do
    echo $i
    (( i+=1 ))
done
```

```bash
cat list | while read f ; do
    # Assume list contains one filename per line
    stat "$f"
done
```
for Loop

for name [in list]; do cmds ; done

- Executes cmds for each member in list
- "$@" used if list isn’t there

```bash
$ for i in a b c ; do
  echo $i
done
a
b
c
```

```bash
for id in $(cat userlist) ; do
  echo "Mailing $id..."
  mail -s "Good subject" "$id"@somemail.edu < msg
done
```
Use `read` to Preserve Spaces

- You must be aware of filenames with spaces
- Consider the different approaches:

```
$ ls -1
'a space'
nospace

$ for f in $(ls -1) ; do
  echo $f
  done
a
space
nospace

$ ls -1 | while read f ; do
  echo $f
  done
a space
nospace
```
Counting for Loop

Bash has a C-like for loop:

```bash
$ for (( i=0; i<3; ++i )) ; do
  echo $i
done
0
1
2

$ for (( i=12; i>0; i-=4 )) ; do
  echo $i
done
12
8
4
```
{x..y} – Brace Expansion

{x..y[..inc]}  
- Generates sequences in a natural way

```bash
$ echo {5..13}
5 6 7 8 9 10 11 12 13
```

```bash
$ echo {a..g}
a b c d e f g
```

- Brace expansion will pad numbers on the left

```bash
$ for i in {000..010..2} ; do echo -n "\$i " ; done
000 002 004 006 008 010
```

- This is quite handy in loops:

```bash
for i in {00..05} ; do \
\rm proc\${i}.log
done
```

YMMV, depending on your platform
Loops – continue, break

- **break** exits a loop
- **continue** shortcircuits the loop, resumes at the next iteration of the loop

```bash
$ for i in {1..42} ; do
  > (( i%2 == 0 )) && continue
  > (( i%9 == 0 )) && break
  > echo $i
  > done
1
3
5
7
$`
case $opt in
  n ) DRY_RUN=1 ;;
  x ) ECHO=1 ;;
  ? | h | H ) usage() ; exit 1 ;;
  ?) echo "Unknown character ;;
esac

Selectively execute `cmds` if `word` matches the corresponding `pattern` (glob)

Commands are separated by `;`

Cases are separated by `;;`
select name [in list]; do cmds ; done

- Much like the for loop
- Displays enumerated menu of list
- Puts user’s choice in name

```
$ select resp in "This" "That" "Quit" ; do
  echo "You chose $resp"
  [ "$resp" == Quit ] && {echo 'bye!'; break; }
done
1) This
2) That
3) Quit
#? 2
You chose That
#? 3
You chose Quit
bye!
```

Scripts, Arguments
Do *not* write the whole thing at once!

Guys, don’t write the entire script, any program, then try to compile or run. Break it into smaller, logical pieces, and build upwards:
- Get the loop/input correct, e.g., selecting only files of interest
- Separately, get the processing for a single file correct
- Now, put those pieces together

Interactive interpreters help you, here, because you can test a line or a loop interactively as you’re writing your script.
More Bash Parameters

These should be familiar:

\[
\begin{align*}
\$\$ & \text{ The process ID (pid) of the shell} \\
\$? & \text{ The exit status of the last command}
\end{align*}
\]

These are handy inside scripts and functions

\[
\begin{align*}
\$\# & \text{ The number of arguments} \\
\$* & \text{ All arguments} \\
\$@ & \text{ All arguments (individually quoted)} \\
${n} & \text{ The } n^{th} \text{ positional argument}
\end{align*}
\]
Arguments to Scripts

- If the script has a proper sha-bang, and the execution bit is set, the script may be invoked directly.
- Arguments may be supplied, as with any other command:

  myScript arg1 arg2 ...

- Or, `bash` may be invoked explicitly, and given a script as input:

  bash myScript arg1 arg2 ...

Bash, in turn, can take options

- **Dry run. File is parsed, but commands aren’t executed. Check syntax**
  
  -n  

- **Echo on. Commands are echoed to stderr as they’re executed (after parameter, file, etc. expansion has happened)**
  
  -x  

```bash
$ bash -x ./hello
+ echo 'Hello World'
Hello World
+ echo 'My name is kschmidt'
My name is kschmidt
```
Arguments in Scripts

- Arguments are positional
- $0$ is the name of the command (how it was invoked)
- In Bourne, $1$ – $9$
  - Can’t access $10$, it’d be $1$ followed by $0$
- In Bash, we can wrap parameters in curly braces, so, ${1}$, ${12}$, etc.

(See Labs/Bash/args.bash)

```bash
echo "Here is how the script was invoked: $0"
echo "Here are the arguments: $*"
echo "This is the number of arguments: $#"
echo "We’ll show each arg:"
for a in "$@" ; do echo "$a" ; done
```
**Processing Args – ** `shift`

`shift [cnt]`

- Shifts args to the left \(cnt\) (default 1) positions
- \$1\ is gone, \$2 \(\rightarrow\) \$1, etc.

```bash
i=0
while [ ! -z "$1" ] ; do
    (( i+=1 )) # just here to enumerate output
    echo -e "$i\t\$1"
    shift     # old \$1\ is gone
done
```
getopts *optstring* *name*

- Bash built-in
- Just handles short (single character) options\(^1\)
- Returns SUCCESS when it finds an option
- *optstring* is the list of options
  - Options that take an arg are followed by a colon (:\)
- *name* holds the current option
- Index of current arg stored in `$OPTIND`
- If option takes an arg, it is stored in `$OPTARG`

\(^1\)See `getopt` utility
while getopt "ab:cC" opt; do
  case $opt in
    a ) echo "option a, at index $OPTIND";;
    b ) echo "option b with arg = $OPTARG, at index $OPTIND";;
    c | C ) echo "option $opt, at index $OPTIND";;
    ? ) echo "usage: $0 [-a] [b arg] [-c] args..." ; exit 1;;
  esac
done

echo "\$OPTIND = $OPTIND"
shift $((OPTIND -1))  # shift off the options

echo -e "\nHere are the remaining arguments:"
for i in "$@" ; do
  echo -e "\t$i"
done
Functions
function name {body}  
name() {body}

- Executed in the same environment
- Arguments to function are handled the same as arguments to a script
- Can be called recursively (see $FUNCNEST)
- Built-in `return rv` can be used in a function, to return execution (and optional status `rv`) to caller
  - Otherwise, status of last command is used
- Export to make available to subshells:

```
export -f funcname
```

---

1 In fact, the curly braces and body can be any compound command
2 Except that $0 is unchanged. See $FUNCNAME
Functions – Parameters

• Accessed just as in a script
• $1, $2, …
  • $0 is the caller’s $0
  • See $FUNCNAME
• $#, $*, $@ behave the same
• shift works the same
function hello
{
    echo "hello $1"
    if [[ -n "$2" && "$2" -gt 1 ]]; then
        hello $1 $((2-1))
    fi
}

Called as a script would be:

$ hello Vera 3
hello Vera
hello Vera
hello Vera
hello Vera
Local Variables in Functions

`local {var}`
- Defines variable(s) local to function
- Won’t step on caller’s environment

```bash
function hello {
    local USER='Elmer Fudd'
    FOO='Hunting Wabbit'
    echo "Hello, $USER, you are $FOO"
}
```

```bash
$ FOO='Baking Cookies'
$ echo $USER
kschmidt
$ hello
Hello, Elmer Fudd, you are Hunting Wabbit
$ echo $FOO
Hunting Wabbit
$ echo $USER
kschmidt
```
Parameter Expansion
Unset or Null Parameters

Expansions for unset (or null) parameters:

- `${param:-word}`: Use `word` if `param` is not set or is null
- `${param:=word}`: Use `word` if `param` is not set or is null, set it to `word`
- `${param:?word}`: If `param` is not set or is null, print `word` to stderr, exit shell (if not interactive)
- `${param:+word}`: If `param` is set, use `word`, otherwise use null
E.g. – Null Parameters

```bash
$ unset foo
$ echo ${foo:-"Hello!"}
Hello!
$ echo $foo

$ echo ${foo:?"Houston, we have a problem"}
bash: foo: Houston, we have a problem
$ echo ${foo:+"We here?"}

$ echo ${foo:='I am de Fault'}
I am de Fault
$ echo $foo 
I am de Fault
$ echo ${foo:?"Houston, we have a problem"}
I am de Fault
$ echo ${foo:+"We here?"}
We here?
```
Removing Patterns

Remove leading or trailing Patterns ( globs )

\[
\begin{align*}
\text{Remove shortest leading pattern} & \quad \text{${{param}\text{#pattern}}$} \\
\text{Remove longest leading pattern} & \quad \text{${{param}\text{##pattern}}$} \\
\text{Remove shortest trailing pattern} & \quad \text{${{param}\text{%pattern}}$} \\
\text{Remove longest trailing pattern} & \quad \text{${{param}\text{%%pattern}}$}
\end{align*}
\]

$ f=a^b^c$

$\text{echo} \ \text{${f\text{#*^}}$}$

$ b^c$

$\text{echo} \ \text{${f\text{##*^}}$}$

$ c$

$\text{echo} \ \text{${f\text{%%^*}}$}$

$ a$

$\text{echo} \ \text{${f\text{%^*}}$}$

$ a^b$
Parameters – Pattern Substitution

`${param/pattern/string}`
- Substitutes `string` for longest matching `pattern`
  - Again, `pattern` is a glob, not a regular expression
  - If `pattern` starts with `/`, it substitutes all occurrences

```bash
$ f="The cat sat on the hat"
$ echo ${f/[hcs]at/XXX}
The XXX sat on the hat
$ echo ${f//[hcs]at/XXX}
The XXX XXX on the XXX
```
Parameters – Substrings

\$\{param:offset\}\$
\$\{param:offset:length\}\$
  - Extract substring starting at index offset
  - Extract length (all remaining) characters

$ n='Kurt Schmidt'$
$ echo \$\{n:3\}\$
  t Schmidt
$ echo \$\{n:5:3\}\$
  Sch
Parameters – Change Case

`${param<OP>pattern}`

- Change case of characters matched by `pattern`
- `pattern` should not attempt to match a larger string
- If `pattern` is missing, treated as a `?`
- `^` – Convert first (each) matched character to upper
- `,` – Convert first (each) matched character to lower

```bash
$f=SHOUT
$ echo ${f,}
sHOUT
$ echo ${f,,}
shout
$ echo ${f,[SOT]}
sHOUT
$ echo ${f,,[SOT]}
sHoUt
$f=quiet
$ echo ${f^^}
QUIET
```