Shell Scripting 1: Scripts & Variables

1. **Scripts and Variables**
   - **shells review**
   - **shell scripts**
   - **Bash syntax basics**
   - **variables**
   - **parameters**
   - **quoting**
2. Control Constructs
3. Shell Expansions
4. Functions and Arrays
5. Bash Development
6. Practical Advice
7. Bash Details and Advanced Features

Shells Review

Shells provide the **command line interface** (**CLI**) to an OS: users type commands to a *shell* program running in a *terminal window*, and the shell interacts with the OS **kernel** and other programs to carry out the commands.

Shells are **command interpreters**:
they parse what the user types, and may rewrite/expand various shell "shorthands," and finally carry out the prescribed actions.

Bash (**Bourne Again SHell**) is the default Linux shell.

Bash is a GNU extension of the original **Bourne shell** (**sh**).

Other shells are available: `csh`, `tcsh`, `tcsh`, `zsh`.

Shells Review (contd.)

Interacting with a shell via the CLI is termed **interactive use**.

Bash has a number of features designed specifically to enhance user productivity for *interactive use*:

- **command-line editing**
- **tab completion**
- **command history**
- **prompt** customization

In addition to supporting interactive use, most shells provide **control constructs** and other features that support writing simple **programs** using the shell language.

These features make Bash a type of **programming language**.

Shell Scripts

Programs written in shell languages are called **shell scripts**.

A **scripting language** is a type of programming language that allows for **rapid development** of simple programs, due to:

- being **interpreted** rather than **compiled**
- using simple **syntax** and **data typing** mechanisms
- having **automatic memory management**

Scripting languages have become popular, and several non-shell scripting languages are in wide use: PERL, Python, PHP, JavaScript.

Shell Scripts (contd.)

Shell scripts get their power because they are able to invoke all the same programs/commands/utilities that you can invoke interactively.

This allows one to implement complex tasks without having to write everything *"from scratch."*

Many Linux GUI programs invoke command-line programs, and these same programs can be called from shell scripts too.

Much of the startup of a Linux system used to involve running shell scripts (e.g., **service scripts** in `/etc/rc.d/init.d`).

Shell Scripts (contd.)

System administrators often write shell scripts to *automate tasks* that are done repeatedly.

They can be handy for normal users as well, when certain tasks need to be done repeatedly.

Shell scripts are also handy for those infrequent tasks that require difficult to remember commands.

You simply save the required commands in a script file, which you can easily invoke when needed.

Creating a Shell Script

A shell script is just a **text file**, which you can create with any **text editor** you want (though some editors are **syntax aware**).

Shell scripts can be executed just like a **binary** (**compiled**) program, as long as:

- the *first line* of the script file uses the following notation to tell the OS what interpreter to use:
  `#!/bin/bash`
  (the `#!` must be the first two characters in the file)
- the user has *execute permission* for the script file

Note that `#` is the **comment character**, and normally everything that follows on a line is ignored (`#!` is special).

Shell Script Example

Shell script to backup `~/Documents` to some directory:
(if script file is `backup-docs`, call like: `backup-docs ~/backups`)

```
#!/bin/bash

# Make certain that directory for backup exists:
if [[ ! -d "$1" ]]; then
    echo "Invalid directory for backup: $1"
    exit 1
fi

# Get date and  hostname:
date=$(date "+%y%m%d")
hostname="${HOSTNAME%%.*}"
backup_filename="Documents-${hostname}-${date}.tgz"

# Perform backup:
cd ~
tar -czf "$1/${backup_filename}" Documents

exit
```

Bash Syntax Basics

A programming language's **syntax** defines words and structure required for a program to be legal/valid in the language.

There are some critical differences between Bash's syntax and that of the C-family languages.

These differences often catch new Bash programmers!

Among the most important differences are those involving **line breaks** and **whitespace**.

(A **line break** is produced by a **newline**/**linefeed** character.)

(**Whitespace** refers to a sequence of space and tab characters.)

Bash Syntax Basics (contd.)

In C/C++/Java, *line break and whitespace differences generally are not meaningful*:

- programs could be written as a *single line* if desired (that's why you have to have `;`'s at the ends of most lines)
- whitespace is added only to improve readability:
  - `x=1;` is equivalent to `x = 1 ;`
  - `if(x==y)` is equivalent to `if ( x == y )`

This means that there are many different ways that the exact same code can appear in C/C++/Java.

Bash Syntax Basics (contd.)

In Bash, *line break and whitespace differences often are meaningful:*

- a *line break* has much the same effect as a "`;`" in C/C++/Java
- *whitespace* differences can change semantics:
  - `x=1` assigns 1 to variable x
  - `x = 1` invokes command x with arguments = and 1
  - `${x}` evaluates a variable to get its value
  - `${ x }` is an error
- *adjacent placement* produces **concatenation**:
  - `${x}${y}` produces a single (concatenated) result
  - `${x} ${y}` produces two separated results

Variables

In programming languages, **variables** are used to temporarily store a value during the run of the program.

Programming languages like C/C++/Java require that variables be created via **declarations** before they can be used:

```
int x;
```

Declarations also define the **type** of value a variable can store.

The value of a variable is changed with an **assignment statement**:

```
x = 10;
```

(In this context, "=" is the **assignment operator**, not the **equality** relationship.)

Variables (contd.)

In Bash, variables *do not have types* (mostly) and are *only rarely declared*.

Values are effectively kept as *strings*, which may be interpreted as integers depending on the context.

A **variable** is usually created by an **assignment**:
- `x=10` (lack of whitespace is important)
- `x=` or `x=""` assigns the default/empty value

(Again, note that "`x = 10`" is *not* a legal assignment, due to the whitespace around the =.)

The `unset` command will *delete* a variable:

```
unset x
```

Variables (contd.)

In languages like C/C++/Java, using a variable in an *expression* causes the variable to be **evaluated**.

Evaluating a variable means that its **value** is retrieved and then substituted for the variable's name:

```
int y = x + 100;
```

In Bash, variables are *not automatically evaluated*.

Instead, retrieving a variable's value requires explicitly **evaluating** the variable.

The synatx for evaluating a variable is:
- `${var}` (technically, this is called **parameter expansion**)
- `$var` (shorthand form, if end of variable name is clear)

Variables (contd.)

Example to demonstrate:

```
x=10
echo x     #prints out "x"
echo ${x}  #prints out "10"
```

Variable names can contain only alphanumeric characters plus underscores, and must begin with an alphabetic character or underscore

Variable names are *case sensitive*.

Standard style is to use *lowercase* variable names.

(Recall that **environment variables** are given *uppercase* names.)

Parameters

In Bash, **parameters** are *"entities that can store values."*

There are three types of parameters:
- **variables** – denoted by *names*
- **positional parameters** – denoted by *integers*
- **special parameters** – denoted by *special characters*

Positional parameters hold the *command-line arguments* passed to the shell script when it is run:
- e.g., if execute: `myscript norm 50`
- `$0` will give the script name: `myscript`
- `$1` will give the first argument: `norm`
- `$2` will give second argument: `50`

Parameters (contd.)

A positional parameter of more than a single digit must be enclosed in braces when evaluated (e.g., `${10}`).

Positional parameters' values *cannot be changed via assignment*.

The `set` builtin can be used to set them however.

E.g., the following sets the positional parameters 1, 2, 3 to have the values A, B, C:

```
set A B C
```

(This is primarily useful for interactively testing code.)

The `shift` builtin can be used to *remove* one or more of the positional parameters "from the left."

E.g., `shift` alone makes `$2` into `$1`, `$3` into `$2`, etc.

Parameters (contd.)

There are various **special parameters**:

- `$*` and `$@` both produce a **list** of the script arguments, but they differ if used inside of *double quotes*:
  - `"$*"` produces a *single word* from all the arguments
  - `"$@"` produces a *list* from the arguments
- `$#` gives the number of script arguments
- `$$` gives the process ID (PID) of the shell
- `$?` gives the last command's exit status

Quoting

Quoting disables the special meaning/treatment for certain words and characters.

For example, spaces usually are **word separators**, so filenames containing spaces must be quoted.

There are three types of quotes:

- **double quotes** (`""`): `"file with spaces"`
- **single quotes** (`''`): `'file with spaces'`
- **escapes** (`\`): `file\ with\ spaces`

There are three different mechanisms because they have slightly different behavior.

Also, you can use single quotes inside of double quotes, but you cannot use single quotes inside of single quotes, etc.

Quoting (contd.)

Double quotes preserve literal characters, except:

- `$` and `` ` `` retain their special meaning
- `\` retains its special meaning when it precedes: `$`, `` ` ``, `"`, `\`, or *<newline>*
- special parameters `*` and `@` have special meaning when inside of double quotes

E.g., `"$variable"` evaluates `variable` and double quotes its value.

This is very handy if the value contains spaces, and you want it treated as a unit:

```
file="filename with spaces"
...
ls -l "$file"
```

Quoting (contd.)

Single quotes preserve literal characters
(a single quote cannot occur within single quotes).

The **backslash** is the **escape character**, preserving the literal value of the next character, except:

- `\`<*newline*> is treated as a **line continuation**
  (meaning it is effectively removed/ignored)

Please note that Bash does not follow the **C escape notation** for special characters, e.g., `\n` for a *newline* character in a `char`/string.

However, the form `$'`*string*`'` produces a string with *escaped characters* replaced as in C strings:

- `\n` by <*newline*>, `\r` by <*return*>, `\t` by <*tab*>, etc.
- e.g., `$'\n'` is a newline character (`"\n"` is *not*)

Shell Scripting 2: Control Constructs

1. Shells and Variables
2. **Control Constructs**
   - **input and output**
   - **exit status**
   - **command types**
   - **for and for-in loops**
   - **while loops**
   - **if-then-else**
   - **test-commands and expressions**
3. Shell Expansions
4. Functions and Arrays
5. Bash Development
6. Practical Advice
7. Bash Details and Advanced Features

Basic Input/Output

Bash has just one input command and two output commands:

`read [options] [var...]`
- reads *one line* from *standard input* (e.g., terminal)
- if var(s) given, assigns first word to first var and so forth, rest to last var
- if no vars, line assigned to variable `Reply`

`echo [-n] [arg...]`
- print arguments to *standard output* (e.g., terminal)
- `-n` option suppresses trailing <*newline*>

`printf format [arg...]`
- print arguments to *standard output* (terminal)
- `format` is a **format string** as in C's `printf()`

Exit Status

Recall that every Linux process (program) must return a *non-negative integer* that indicates its **exit status**:
- 0 (zero) means process completed *successfully*
- non-zero value means process *failed*
  (the particular value can denote the reason for failure)

Shell scripts return exit status with the `exit` builtin:
- `exit 0` (success)
- `exit 1` (failure, standard value)
- `exit` (returns status of last command run)

If a script does not end with `exit`, the script returns the exit status of its final command.

Command Types

There are several **command types** in Bash:
- **simple command** – command name along with its options and arguments:
  - `ls -lA Documents`
- **pipeline** – sequence of *simple commands* separated by |'s:
  - `grep printf lab.c | wc -l`
- **list** – sequence of *simple commands* and/or *pipelines* separated by ";" or "&&" or "||":
  - `gcc -o lab1 lab1.c && ./lab1`
- **compound command** – constructed using the various shell **control constructs** (for, if-else, etc.):
  - `for f in *; do ls -l ${f}; done`

Indenting Bash Code

Program code is **indented** according to certain patterns in order to enhance readability for humans.

While not necessary to be able to run the code, unindented code can be extremely difficult to understand and get correct.

As with most programming languages, you may see several alternative indentation patterns being used with Bash.

Syntax for the Bash control constructs is shown with the common indentation patterns that are preferred by the instructor.

If you wish to use alternative indentation patterns, make certain you understand when a *line break* or ";" is required by the syntax.

Looping: For

Looping constructs allow some code to be executed *multiple times*.

The code being executed each time through the loop is called the loop **body**.

A **for loop** is generally used to execute the body code a *definite number of times*.

For loops use an **index variable** that gets incremented/decremented until it reaches a certain value.

Often, the index variable is used in the body code

Looping: For (contd.)

For loop syntax:

```
for ((expr1 ; expr2 ; expr3)); do
  command1
  ...
done
```

Example: print sequence of numbers:

```
for ((x=1; x<=10; x++)); do
  echo $x
done
```

This would print out the numbers 1 through 10, each on a separate line.

Looping: For-In

A **for-in loop** is also referred to as a **mapping loop**.

It allows one to do something with each element of a **list**.

For-in loops are heavily used in shell scripting because **lists** are common from **filename expansion** and CLI arguments (`"$@"`).

For-in loop syntax:

```
for variable in list; do
  command1
  ...
done
```

Maps down **list**: successively binds **var** to each "word" in **list**, executing the body commands with each binding.

Looping: For-In (contd.)

Example: make backup copies of some files:

```
for file in *.text; do
  cp $file $file.save
done
```

Suppose the the CWD contained the files `file1.text`, `file2.text`, and `file3.text`.

`*.text` would expand to the *list:*

```
file1.text file2.text file3.text
```

The above for-in loop would set the variable `file` to `file1.text` and execute the `cp` command, then set `file` to `file2.text` and execute the `cp` command, and so forth.

Looping: For-In (contd.)

A `for-in` can be done on single line if working interactively:

```
for var in list; do command1; command2; done
```

Some examples for interactive use:

```
for n in 1 2 3; do touch test$n; done

for n in $(seq 1 10); do touch test$n; done

for f in *.text; do touch "$f"; done

for f in *.txt; do mv "$f" "${f%.txt}.text"; done
```

Looping: While

While loops are used when we want to keep executing body code while some **condition** is true.

While loop syntax:

```
while test-command; do
  command1
  ...
done
```

A **test-command** can be:

- a *command* – true if command has success **exit status**
- `[[ expression ]]` – true if **expression** is true

Test-Commands will be discussed further below.

Looping: While (contd.)

Example: reading lines from the terminal:

```
while read line; do
  echo "$line" > user-input.text
done
```

This while loop would read lines from the terminal, and write each out to the file `user-input.text`.

(Review **output redirection** (>) from the Bash lectures if necessary.)

Example: similar to for loop:

```
x=1
while [[ $x -le 10 ]]; do
  x=$((x + 1))
done
```

Looping: Until

Bash also provides **until loops** as a syntactic variant of *while loops*.

Until loop syntax:

```
until test-command; do
  command1
  ...
done
```

We say it is a syntactic variant, because the above could have been written as a *while* loop:

```
while ! test-command; do
  command1
  ...
done
```

Looping: break and continue

The `break` and `continue` builtins are available to use with the looping constructs:

- `break [N]` – break from Nth enclosing for/while/until
- `continue [N]` – resume next iteration of Nth enclosing for/while/until

Branching: If-Then-Else

Branching constructs allow you to execute code only if a certain condition is true or not true.

If-then syntax:

```
if test-command; then
  command1
  ...
fi
```

If-then-else syntax:

```
if test-command; then
  command1
  ...
else
  command1
  ...
fi
```

Branching: If-Then-Else (contd.)

Example: create a directory if it does not exist:

```
if [[ ! -d tempdir ]]; then
  mkdir tempdir
fi
```

Example: change `x`'s value depending on its current value:

```
if [[ $x -gt 10 ]]; then
  x=$((x - 10))
else
  x=$((x + 1))
fi
```

Branching: If-Elif

if-elif or **if-elif-else**, are like if-then-else but:

- with one or more instances of:

```
elif test-command; then
  command1
  ...
```

- optionally finishing with:

```
else
  command1
  ...
fi
```

Branching: If-Elif (contd.)

Examples:

```
if [[ $x -gt 10 ]]; then
  x=$((x*2))
elif [[ $x -gt 5 ]]; then
  x=$((x+5))
else
  x=$((++x))
fi
```

Alternatives: Case

A set of *alternatives* can be handled with **case**, which is similar (but more powerful) than C's **switch**.

case syntax:

```
case word in
  pattern [| pattern]...) commands-list ;;
  ...
  [*) commands-list ;;]
esac
```

Example:

```
case "$response" in
  y|Y|yes|YES) echo "OK" ;;
  n*|N*) exit 0 ;;
  *) echo "Invalid Response" ;;
esac
```

Test-Commands

Notice that in the above *control constructs*, we have `test-command` where you might expect to see `expression`.

Bash differs from C-family languages in that the conditional control constructs (if, while, etc.) *do not evaluate an expression*.

Rather, they *execute a command*, and use its *exit status* to determine true/false:

- 0 (success) means true
- $>$ 0 (failure) means false

These commands are what we have shown as `test-command`: a command is executed, its exit status determines which branch to take or whether to continue looping.

Test-Commands (contd.)

One can use *any command* as a test-command:

- the command's exit status will determine branching/looping
- the command's actions get performed as a **side effect**

Example: see if a file contains some string:

```
if grep --quiet "$string" "$file"; then
  #file contains string:
  ...
fi
```

Example: read from file until hit file-end:

```
while read input; do
  #next line from file is in variable input:
  ...
done
```

Test-Commands (contd.)

Note that any *output* from a command being used as a test-command will be printed out, including *error messages*.

It is often necessary/desirable to suppress any error output—or even all output—when using a command to test a condition.

Can suppress error messages (stderr) with "`2>/dev/null`":

```
if metaflac "$file" 2>/dev/null; then
  ...(process flac file tags)...
else
  echo "Failed to retrieve FLAC file tags!"
  exit 1
fi
```

Test-Commands (contd.)

Can suppress all output (stdout + stderr) with "`&>/dev/null`":

```
if ! which metaflac &>/dev/null; then
  echo "metaflac program must be installed!"
  exit 1
fi
```

Test-Command Expressions

Test-commands are often written using one of *three special notations*, which effectively *turn expressions into commands*:

- `[ conditional_expression ]`
- `[[ conditional_expression ]]`
- `(( arithmetic_expression ))`

`[ expression ]` is the older style of test, whose options are more limited, with somewhat stranger syntax.

`[[ expression ]]` is the newer style of test, which uses a more standard syntax and has more functionality, so preferred.

`(( expression ))` evaluates an arithmetic expression:

- 0/true status, if the expression evaluates to non-zero
- 1/false status, if the expression evaluates to zero

Test-Command Expressions (contd.)

Notes:

- *whitespace* is generally required between `expression` and the brackets (e.g., `[[` and `]]`)
- any value besides the empty string (`""`) is considered true inside of `[ ]` or `[[ ]]` (e.g., `[[ false ]]` is true)

There are *three classes of expressions* that can occur inside of `[ ]` or `[[ ]]`:

- **string comparisons**
- **arithmetic comparisons**
- **file conditions**

String Comparison Expressions

String comparison expressions compare or test strings:

- `==` – binary equality operator (can just use `=`)
  (RHS argument can be a *globbing pattern*)
- `!=` – binary inequality operator
- `>` – **lexicographic ordering** (“sorting”) comparison
- `<` – same
- `-n` – unary operator tests if non-null/non-empty string
- `-z` – unary operator tests if null/empty string

(Remember that variable/parameter values are effectively stored as strings.)

Arithmetic Comparison Expressions

Arithmetic comparison expressions compare values as *integers*:

- `-eq` – $=$
- `-ne` – $\neq$
- `-lt` – $<$
- `-le` – $\leq$
- `-gt` – $>$
- `-ge` – $\geq$

E.g., `[[ $x -gt 5 ]]`

File Condition Expressions

File condition expressions test/compare file characteristics.

Unary file conditions: (partial list)

- `-e` file – file exists
- `-f` file – file exists and is regular
- `-d` file – file exists and is a directory
- `-s` file – file exists and has size greater than zero

Binary file conditions:

- file1 `-nt` file2 – file1 newer than file2 (mtime)
- file1 `-ot` file2 – file1 older than file2
- file1 `-ef` file2 – file1 and file2 are same (i.e., inode)

e.g., `[[ -f "$file" ]]`

Expression Logical Operators

Expressions inside of `[[ ... ]]` can be combined using several **logical operators**:

- `! expression` – true if expression is false
- `expr1 && expr2` – true if both expressions true
- `expr1 || expr2` – true if at least one expression true
- `( expression )` – used to affect operator order

Pattern Matching Expressions

One thing that is sometimes required of expressions is to be able to check if a filename or variable value *matches some pattern*.

The `==` and `!=` test conditions both allow the RHS argument to be *filename expansion pattern*.

Examples:

```
if [[ $file == *.bak ]]; ....

if [[ $x == 1*?1 ]]; ....
```

Pattern Matching Expressions (contd.)

A **regular expression** matching operator is also available for use in test-command expressions: `=~`

This is a binary operator, where the RHS is to be an *extended regular expression*.

Examples:

```
if [[ $x =~ 10*1 ]]; ....

if [[ $y =~ test.+ ]]; ....
```

true & false

`true` and `false` are commands:
- `true` simply returns 0/success
- `false` simply returns 1/failure

They are useful as **Boolean/flag** variable values, because they function as desired when used as test-commands:

```
#Exit if no regular files in CWD:
regfile=false
for file in *; do
  if [[ -f $file ]]; then
    regfile=true
  fi
done

if ! $regfile; then
  exit 1
fi
```

Shell Scripting 3: Shell Expansions

1. Shells and Variables
2. Control Constructs
3. **Shell Expansions**
   - **parameter expansion**
   - **command substitution**
   - **arithmetic expansion**
   - **filename expansion**
   - **filename expansion quirks**
4. Functions and Arrays
5. Bash Development
6. Practical Advice
7. Bash Details and Advanced Features

Expansion Phases

Recall that a key aspect of Bash "interpreting" each command line prior to executing it is the various **shell expansions**:

1. **brace expansion**
2. **tilde expansion**
3. **parameter expansion**
4. **command substitution**
5. **arithmetic expansion**
6. **filename expansion** (also known as **globbing**)

These expansion are applied to each line in a shell script just as they are to each command-line you type interactively.

Some expansions are more commonly used in shell scripts than in interactive use.

Parameter Expansion

At its most basic, **parameter expansion** is *variable evaluation*: e.g., `$x` or `${x}`.

However, there are a number of **operators** that can be used within the `${}` form to modify the value returned from the parameter evaluation.

These operators are among the few functions that Bash has available for "**string processing**" (manipulating strings).

Complex string processing is generally done by invoking other programs, such as **AWK** and **SED**.

Parameter Expansion (contd.)

Key parameter expansion operators:

- ${param%pattern} – remove shortest match against `pattern` from the RHS of `param`'s value
- ${param%%pattern} – remove longest match against `pattern` from the RHS of `param`'s value
- ${param#pattern} – remove shortest match against `pattern` from the LHS of `param`'s value
- ${param##pattern} – remove longest match against `pattern` from the LHS of `param`'s value
- ${#param} – length of `param`'s value (in chars)
- ${param/pattern/string} – replace (longest) match against `pattern` in `param` with `string` (which may be empty)

Parameter Expansion (contd.)

Key parameter expansion operators: (contd.)

- `${param:offset:length}` – substring of `param`'s value, starting at 0-based `offset`, `length` optional
- `${@:offset:length}` – sublist of positional parameter, starting at 1-based `offset`, `length` optional
- `${!param}` – *indirect expansion*: use `param`'s *value* as the name of the variable to evaluate

The *patterns* used with the parameter expansion operators are constructed with the *filename expansion metacharacters*: `*`, `?`, and `[ ]`.

These are *not regular expression* patterns!

Parameter Expansion (contd.)

Examples:

```
file=test.text.save

echo ${file%.*} ==> test.text

echo ${file%%.*} ==> test

echo ${file#*.} ==> text.save

echo ${file##*.} ==> save

echo ${file/text/txt} ==> test.txt.save
```

Command Substitution

Command substitution allows the output of a command to be substituted into another command or stored in a variable.

There are two syntax alternatives:
- `$(command)` (newer)
- `` `command` `` (older)

E.g., get (absolute) path of program `ls`:
- `dayofweek=$(date +%A)`
- `` dayofweek=`date +%A` ``

Command substitution is widely used in shell scripting.

It is frequently used in an *assignment*, to capture the output of a command into a variable, so that it can be processed/used.

Command Substitution (contd.)

E.g., given one of the above assignments, we could do:
```
echo "Today is $dayofweek"
```

Of course we could also just do:
```
echo "Today is $(date +%A)"
```

However, if we need to have the day multiple times, repeatedly invoking command substitution will be much slower (each command will require creating a **subprocess** in which to run `date`, with its output captured by the shell, etc.).

Command substitution is not limited to *simple commands*, we frequently use *pipelines* as well:
```
os=$(grep DESCRIPTION /etc/lsb-release | cut -d= -f2 | tr -d "\"" | tr " " "_")
```

(Gets name of current Linux distro as a single word, e.g., `Mageia_5`.)

Command Substitution (contd.)

Another thing to be clear about is that what gets substituted in is what `command` writes to **standard output**.

Since only "standard output" is captured, it is often necessary to suppress *error messages* from being printed out to the terminal:

- e.g., `cmdpath=$(which $cmd 2>/dev/null)`
- get path of a command in variable `cmd` by using `which`
- if the command is not found, though, `which` will print an error message to **standard error**
- `2>/dev/null` causes **standard error** (**file descriptor** 2) to be *redirected* to a fake device that just throws data away
- note that `cmdpath` ends with *empty string* value if `which` fails

(Recall that we saw something similar with `command-test` commands!)

Command Substitution (contd.)

Note that being inside of `$( )` does not protect *parameter expansion* results from **word splitting**.

Thus, variables evaluated inside of command substitution forms must often be protected by quoting:

```
echo $(ls -l "$file")
```

Word splitting may also applied to command substitution's results, so command subsitution forms must themselves often be quoted:

```
ls -l "$(realpath "$file")"
```

The Bash reference manual says: "If the substitution appears within double quotes, word splitting and filename expansion are not performed on the results."

Command substitutions may even be **nested** (`$()` form is easier):

```
listing=$(ls -l "$(realpath "$file")")
```

Arithmetic Expansion

While variable values are effectively stored as strings, it is possible to get them *interpreted as numbers* (integers).

`$(( math_expression ))` causes `math_expression` to be evaluated as an arithmetic (integer) expression, producing a number result.

Both numeric constants and variables can appear in expressions (variables do *not* require `$` for evaluation).

Can use most standard arithmetic operators:

- unary: `++` and `--`
- binary: `+`, `-`, `*`, `/`, `%`, `**`, `<<`, `>>`, `&`, `|`, etc.

E.g., `x=$((x + 25))`

Filename Expansion

Filename expansion was covered in the lectures on Bash.

Filename expansion uses several **metacharacters** to do *pattern matching on filenames*, allowing *multiple files* to be specified compactly.

The filename expansion metacharacters:

- `*` – matches any string, including the null/empty string
- `?` – matches any single character (but not none)
- `[...]` – matches any single enclosed character:
  `[yY]` `[abcd]` `[0123456789]`

Filename Expansion (contd.)

Filename pattern example:

- `i*-[1-5]??.{txt,text}`
- will match these files:
  - `i-123.text` (`*` matches empty string, `?`'s match 23)
  - `iabc-5ab.txt` (`*` matches abc, `?`'s match ab)
- will not match these files:
  - `i123.txt` (no dash)
  - `abc-123.txt` (doesn't start with i)
  - `ia-923.txt` (9 is not in range 1-5)
  - `ia-12.txt` (second `?` has no match)
  - `i-123.text.save` (does not end in "`txt`" or "`text`")

Filename Expansion Quirks

Filename expansions are used frequently in shell scripts, but there are a few quirks one needs to be aware of.

While the `*` meta-character "matches any string, including the null string," filename patterns that start with `*` do *not* match **hidden files** (set the `dotglob` option to change this behavior).

Thus, the following will not list hidden files (in the CWD):

```
for file in *; do
  echo "$file"
done
```

Begin filename patterns explicitly with `.` to match hidden files.

You can match *all* files with a *list* of both `.*` and `*`:

```
for file in .* *; do
  echo "$file"
done
```

Filename Expansion Quirks (contd.)

Using the pattern `.*` often leads to unexpected results, however, becase *every directory* contains `.` and `..` as the first two entries.

These will be matched by `.*`, but are often not really desired:

```
for file in .* *; do
  ls "$file"
done
```

⇒ causes the CWD to be `ls`'d twice effectively, plus parent.

Instead do:

```
for file in .* *; do
  if [[ "$file" != . && "$file" != .. ]]; then
    ls "$file"
  fi
done
```

Filename Expansion Quirks (contd.)

If a filename expansion pattern matches no files, what results is *the pattern itself* (unless `nullglob` or `failglob` options are set).

This can cause errors in scripts:

```
for file in *.text; do
  ls "$file"
done
```

⇒ `ls: cannot access *.text: No such file or directory`

It is generally a good idea to check that the match is a file:

```
for file in *.text; do
  if [[ -f "$file" ]]; then
    ls "$file"
  fi
done
```

Filename Expansion Quirks (contd.)

/'s in patterns represent *directories,*, so:

- `*/` – all the directories in CWD
- `*/*` – all the files in all the subdirectories of CWD
- `*/*/` – all the directories in all the subdirectories of CWD

Filename Expansion Shell Options

The following **shell options** affect *filename expansion* when set:

- `dotglob` – match filenames beginning with "." even if dot not explicitly in pattern (i.e., match *hidden filenames* by default)
- `extglob` – extended pattern matching features enabled
- `failglob` – patterns that fail to match filenames produce error
- `nocaseglob` – match filenames in a *case-insensitive* manner
- `nullglob` – patterns that fail to match filenames result in *null string* (rather than pattern itself)

Set/enable option with: `shopt -s OPTNAME`

Unset/disable option with: `shopt -u OPTNAME`

Shell Scripting 4: Functions & Arrays

1. Shells and Variables
2. Control Constructs
3. Shell Expansions
4. **Functions and Arrays**
   - **functions**
   - **arrays**
   - **calling AWK and SED**
   - **advanced I/O**
5. Bash Development
6. Practical Advice
7. Bash Details and Advanced Features

Functions

Bash supports the use of **functions** (**subroutines**).

A function can be defined using any of the following syntax variations:

- `function` *name* `() {` *...body...*`; }`
- `function` *name* `{` *...body...*`; }`
- *name* `() {` *...body...*`; }`

Functions are called and executed just like simple commands (but are run in the *current shell process*—not a **subprocess**).

Inside the **function body**, *arguments* to the function call can be accessed using the **positional parameters** `$1`, `$2`, etc.

(The original shell positional parameters are *restored* upon return from the function.)

Functions (contd.)

Function to create a required directory if it does not exist:

```
# Usage: ensure_dir DIRECTORY
function ensure_dir ()
{
  if [[ ! -e "$1" ]]; then
    #DIRECTORY does not exist so make it:
    if ! mkdir "$1"; then
      echo "Failed to mkdir $1!"
      return 1  #failure return from function
    fi
  elif [[ ! -d "$1" ]]; then
    #DIRECTORY exists, but as non-directory file:
    echo "Error: $1 exists but is not a directory!"
    exit 1  #failure exit from entire script
  fi
  return 0  #success return from function
}
```

Functions (contd.)

Bash functions differ from C-family language functions/methods:

- their *"parameter list"* is not explicit in the definition
- they run in the *same shell context* as the caller
- their return value is *exit status*

While `ensure_dir` must be called with a single argument, `DIRECTORY`, this is not made explicit in the syntax of the function's definition.

Running in the "same shell context" means that variables defined *outside* the function can be accessed *inside* the function, and any changes to their values will be reflected upon return.

(The only exception to "same shell context" is that *positional parameters* are bound to the function arguments on entry, and restored upon return.)

Functions (contd.)

Since you cannot return *values* from Bash functions, changing the values of external variables is the only method available to return info.

Functions can be **recursive**, but may require care due to running in the same shell context (which is different from most languages).

Variables *local to the function* in context may be created with a `local` **declaration**:

```
local cwd=$PWD
```

Arrays

Bash supports **one dimensional arrays**:

- zero-based indexing
- no size limits, dynamically expandable
- elements need not be indexed/assigned contiguously
- can use `declare` to make array variable:
  ```
  declare -a arr
  ```
- can also just assign elements:
  ```
  arr[0]=ford; arr[1]=gm; arr[2]=honda...
  ```
- can initialize entire array with paren notation:
  ```
  arr=(ford gm honda...)
  ```
- can initialize entire array with paren notation and list:
  ```
  elements="ford gm honda..."
  arr=(${elements})
  ```

Arrays (contd.)

Array access:

- retrieving array element value:
  ```
  ${arrvar[1]}
  ```
- get entire array of values as single word:
  ```
  "${arrvar[*]}"
  ```
- get entire array of values as separate words:
  ```
  "${arrvar[@]}"
  ```
- get sub-array of values as separate words:
  ```
  "${arrvar[@]:offset:length}"
  ```
- get length of array (number of elements):
  ```
  ${#arrvar[*]}
  ```
- get length of an element:
  ```
  ${#arrvar[1]}
  ```

Arrays (contd.)

Array examples:

Initializing 10 array elements to be index + 5:

```
for ((i=0; i<9; i++)); do
  arr[$i]=$((i+5))
done
```

Non-contiguous elements:

```
arr[0]=a
arr[10]=b
echo ${#arr[*]}
echo ${arr[*]}
```

```
⇒  2
⇒  a b
```

Calling AWK and SED

Bash has limited *string processing* capabilities.

Because of this, the programs **AWK** and **SED** are often used in shell scripts to do complex string processing.

Basic usage is as follows:

- `awk -e AWK_CODE FILE`
- `... | awk -e AWK_CODE`
- `sed -e SED_CODE FILE`
- `... | sed -e SED_CODE`

Calling AWK and SED (contd.)

Examples:

- Get device for mountpoint `/backup`:
  `dev=$(awk -e '/\/backup/{print $1}' /etc/fstab)`
- Get file size:
  `size=$(ls -l "${file}" | awk -e '{print $5}')`
- Compress whitespace in a file:
  `sed -e  's/[[:blank:]]\+/ /g' "${file}"`
- Replace spaces with underscores in filename:
  `newname=$(echo "$file" | sed -e 's/ /_/g')`

See the awk and sed man pages plus slides for further info.

Advanced I/O

Redirections are normally *temporary*, in that they apply only to one command (that might be run in a new *subprocess(es)*).

E.g., using `nl` to number lines from a file:

```
nl -ba < "test.text"
```

E.g., same thing using `read` and a loop:

```
n=1
while read line; do
  printf "%3d: %s\n" $((n++)) "$line"
done < "test.text"
```

Advanced I/O (contd.)

The `exec` builtin can be used to (permanently) apply redirections to the current shell process.

This capability can be used to *open files for reading/writing*:

- open file for reading:
  `exec FD< FILENAME`
- open file for writing:
  `exec FD> FILENAME`
- where:
  - `FD` is a **file descriptor** (non-negative integer, *file handle*)
  - `FILENAME` is the file name/path

Advanced I/O (contd.)

Now, reading and printing numbered lines from a file can be done:

```
exec 3< "test.text"
n=1
while read -u3 line; do
  printf "%3d: %s\n" $((n++)) "$line"
done
```

Note that this does not change *standard input*, we simply use `exec` to open the file on FD 3 and read from that FD.

Opening a file for output and periodically writing to it:

```
exec 4> "test.text"
...
echo ... >&4
...
echo ... >&4
```

Advanced I/O (contd.)

`exec` can also be used to (temporarily) change standard input/output:

```
exec 3<&0  #save stdin in FD 3
exec 4>&1  #save stdout in FD 4

exec 0<"input.text"   #redirect stdin to input.text
exec 1>"output.text"  #redirect stdout to output.text

...code using stdin and stdout, now redirected to files...

exec 0<&3 3<&-  #restore stdin and remove FD 3
exec 1>&4 4>&-  #restore stdout and remove FD 4
```

Advanced I/O (contd.)

Other advanced I/O features are **here documents** and **here strings**, which allow *standard input* to be supplied from within a script file itself.

Here document:

```
cat <<-end
        line one
        line two
        end
```

(`<<-` vs. just `<<` causes leading `<tab>` characters in lines to be ignored, so can indent for readability—but must use tabs.)

Advanced I/O (contd.)

Here string:

```
cat <<<$'line one\nline two'
```

So, instead of using `echo` to start a pipeline:

```
echo "$var" | awk '{print $2}'
```

we could use a *here string*:

```
awk '{print $2}' <<<"$var"
```

Shell Scripting 5: Bash Development

1. Shells and Variables
2. Control Constructs
3. Shell Expansions
4. Functions and Arrays
5. **Bash Development**
   - **Bash programming style**
   - **debugging shell scripts**
6. Practical Advice
7. Bash Details and Advanced Features

Becoming a Bash Programmer

One of the hardest aspects of learning a new programming language is learning the best **idioms** for that language.

Often, those new to a language instead try to force code into models they are used to from other languages.

Bash is quite different from C-family languages, so if you try to emulate Java/C approaches, you will often end up being more verbose or less efficient than necessary.

For example, *arrays* and *numeric for loops* are much used in C-family languages, but are generally *not* likely to be the most direct way to do things in Bash.

A key reason is that Bash supports *lists* and the `for-in` list mapping construct, and both command-line arguments and file expansions produce lists.

Becoming a Bash Programmer (contd.)

Suppose invocations of your script have the following syntax:

```
myprog  FILE ARG...
```

If you need to loop through all the args and process each of them, the simplest way to do this in Bash is:

```
file=$1  #save FILE
shift  #remove FILE from the arguments list
#Loop through each supplied ARG:
for arg in "$@"; do
  ...process $arg...
done
```

This is the Bash way to process command-line arguments!

Becoming a Bash Programmer (contd.)

Since Java/C programmers don't have the experience (or option) of using lists, they will probably be tempted to make use of arrays and a numeric for loop:

```
args_arry=("$@")  #turn arguments list into an array
#Loop through each supplied ARG, now in args_arr:
for (( i=1; i<=${#args_arr[@]}; i++ )); do
  ...process $args_array[i]...
done
```

This is clearly neither the easiest nor most efficient way to do things in Bash!

It also clearly marks you as an inexperienced (or incompetent) Bash programmer.

Becoming a Bash Programmer (contd.)

On the other hand, sometimes people new to Bash end up making frequent (and unnecessary) use of fairly esoteric Bash elements.

A good example is the `eval` builtin:

- `eval [ARG...]`
- Combine `ARG`s into a single string, execute as a shell command.

While `eval` enables one to do some pretty slick things, it will generally be required only *extremely rarely* (consider that very few programming languages have a comparable capability).

In particular, one does not want to do:

```
  eval ls -l
```

when the following is completely identical in functionality:

```
  ls -l
```

Becoming a Bash Programmer (contd.)

Here is an example when `eval` is required:

```
# Print the capital letters from A to Z:
char_decimal=65  #ASCII code for A
while [[ char_decimal -le 90 ]]; do
  char_octal=$(echo "ibase=10; obase=8; $char_decimal" | bc)
  eval char="$'\\${char_octal}'"
  echo $char
  char_decimal=$((char_decimal + 1))
done
```

Key is to construct form `$'\nnn'` which represents the character with octal value `nnn`, and then `eval` it to get the actual character.

Developing Shell Scripts

Programs in C/C++/Java and other languages must be **compiled** before they can be run.

Bash programs are **interpreted**, so do not require compilation.

However, they can still contain both **syntax errors** and **logic errors**.

Bash provides an **interactive environment** in which you can *evaluate individual lines of code*.

You should test each line of code by running it in a shell before you commit it to a script file.

If you follow this approach, Bash scripts should contain fewer initial errors than with compiled languages

Debugging Shell Scripts

Unlike a compiled language, script **syntax errors** will manifest themselves *when the script is run*.

A syntax error will produce results like:

```
  myscript: line 22: syntax error near unexpected token ‘else’
```

For such a message, start at line 22 of the script file and *look backwards through the script* to see what is wrong to make Bash think an `else` is not valid where it is on line 22.

The above message resulted from forgetting the `;` before `then`:

```
  if [[...]] then
    ...
  else
    ...
  fi
```

Debugging Shell Scripts (contd.)

Bash can be run in **debugging mode** to assist in debugging scripts.

If Bash is started with the `-x` option, a *trace* of each command plus its arguments gets printed to standard output after the command has been expanded but before it is executed.

You can run a script with this option like:

```
bash -x myscript
```

The `set` builtin can also be used to turn debugging on and off for different portions of the script:

```
set -x  #debugging on
...code to debug...
set +x  #debugging off
```

Debugging Shell Scripts (contd.)

Remember that the `set` builtin can be used to set the *positional parameters* for testing code lines in the interactive environment:

```
set A B C
for param in "$@"; do echo $param; done
```

$\Rightarrow$

```
A
B
C
```

Shell Scripting 6: Practical Advice

1. Shells and Variables
2. Control Constructs
3. Shell Expansions
4. Functions and Arrays
5. Bash Development
6. **Practical Advice**
   - **verifying script invocation and command installation**
   - **verifying commands succeeded**
   - **lists and whitespace and word splitting and IFS**
   - **processing text files**
   - **reading lines from files**
   - **file paths**
   - **decoding command line arguments**
7. Bash Details and Advanced Features

Verifying Proper Script Invocation

It is nearly always a good idea to have some initial code in a script to test if the script was called appropriately.

This is standard practice for Linux commands, and incorrect calls typically result in a brief **usage message** that gives the command's correct call syntax.

For example, suppose we have a script named `myprog` that is to take a *single argument*, which is to be a *directory* path.

In this case, we would at least want to verify that the script was called with *exactly one command line argument*, and we might also verify that the argument names a valid directory.

Verifying Proper Script Invocation (contd.)

We can do both these checks with the following code:

```
# Make certain script was called with a single argument:
if [[ $# != 1 ]]; then
    echo "Usage: myprog DIRECTORY" >&2
    exit 1
fi

# Make certain that DIRECTORY exists and is a directory:
if [[ ! -d "$1" ]]; then
    echo "Invalid DIRECTORY argument: $1" >&2
    exit 1
fi
```

Note that usage messages and other error messages are typically printed to **standard error** (`stderr`), which is done above with the **redirection**: `>&2`

Verifying Proper Script Invocation (contd.)

Many scripts takes some fixed number of arguments plus an *unlimited* number of arguments, such as files:
e.g., `myprog HEADERSTR FOOTERSTR FILE...`

In this case our usage test will have to be like:

```
if [[ $# -lt 2 ]]; then
```

Sometimes scripts can take *any number* of arguments, including zero: e.g., `myprog [FILE...]`
(if no `FILE` is supplied, read from stdin)

In such cases, the script will either not have a usage message test, or one can print the message if the user calls the script like:

```
myprog --help
```

Verifying Command/Executable Installed

Another aspect of writing *robust* scripts is making certain that programs/commands required by the script have been installed on the system.

The best way to do this is usually using the `which` command: "For each of its arguments it prints to stdout the full path of the executables that would have been executed when this argument had been entered at the shell prompt. It does this by searching for an executable or script in the directories listed in the environment variable PATH using the same algorithm as bash(1)."

Example checking for `enscript` (text to Postscript convertor):

```
if ! which enscript &>/dev/null; then
  echo "Program enscript required, please install first!"
  exit 1
fi
```

Verifying Commands Succeeded

Many commands run by shell scripts can possibly *fail*.

Writing *robust* shell scripts means including code that ensures commands have succeeded before proceeding in the script.

Otherwise, invalid data may be passed on, leading to incorrect results or unexpected and difficult to interpret error messages.

(Remember also that we may want to suppress error messages from commands by *redirecting standard error:* "`2>/dev/null`".)

Success/failure of commands is most commonly determined by their **exit status**.

Another approach is to look at the result/output of the command.

Verifying Commands Succeeded (contd.)

There are three approaches for running a command and checking its exit status:

- run the command as a `test-command` inside an `if`:

```
if ! mkdir "$newdir" 2>/dev/null; then
  ...handle mkdir failure...
fi
```

- test the status of a command with `$?`:

```
mkdir "$newdir" 2>/dev/null
if [[ $? != 0 ]]; then
  ...handle mkdir failure...
fi
```

- run a sequence of commands using `&&` or `||`:

```
mkdir "$newdir" 2>/dev/null && cp "$file" "$newdir" || ...handle failure...
```

Verifying Commands Succeeded (contd.)

Sometimes the only way to determine if a command "fails" is by testing its output.

Typically this will mean testing whether results are empty or not.

E.g., we might expect/need a set of files in some directory, but they may fail to be found:

```
files=$(ls "$dir")
if [[ -z "$files" ]]; then
  ...handle failure to find any files in $dir...
fi
```

We will generally use the following *string operators*:

- `-n` – unary operator tests if non-null/non-empty string
- `-z` – unary operator tests if null/empty string

Using a Variable in Subsequent Commands

Variables are often used in shell scripts to capture the output from a command, to allow for further processing.

However, it may not be clear how text that is captured in a variable, especially in later *pipelines*.

The key is know that `echo` writes to *standard output*, which is what gets passed along a pipeline, so it can start a pipeline:

```
  line=$(grep -m1 $pattern "$csv_file")
  field1=$(echo "$line" | cut -d, -f1)
  ...
```

Of course if we only need `field1` from `line`, we can simply do:

```
field1=$(grep -m1 $pattern "$csv_file" | cut -d, -f1)
```

Lists and Whitespace

In Bash, a **list** is a string of words separated by separators. (Note: a data list is not the same as a **list command**.)

E.g., `A B C` (this is a list of three letters, `A`, `B`, `C`)

Lists are heavily used with **for-in** loops:

```
  for file in *; do ...; done
```

(filename expansion of `*` produces a list of filenames)

A fairly frequently encountered issue is that the elements of a list may contain whitespace, so might get split by **word splitting**.

If this happens, an incorrect, longer list results, with the pieces of the intended elements now treated as individual list elements.

Lists and Whitespace (contd.)

For example, suppose you had this 3-element list of filenames in the CWD:
`"file 1 .txt" "file 2 .txt" "file 3 .txt"`

Without the double quotes around each filename, word splitting would produce a 9-element list:
`file 1 .txt file 2 .txt file 3 .txt`

The good news is that lists can work properly with such files:
(`ls -i` lists the filesystem *inode number* for the file)

```
  $ for file in *; do ls -i "$file"; done
  2494675 file 1 .txt
  2494676 file 2 .txt
  2494677 file 3 .txt
```

This code works because *word splitting* is *not* applied to the results of *filename expansion*.

Lists and Whitespace (contd.)

Other approaches can lead to problems however:

```
  $ for file in $(ls); do ls -i "$file"; done
  ls: cannot access file: No such file or directory
  ls: cannot access 1: No such file or directory
  ls: cannot access .txt: No such file or directory
  ls: cannot access file: No such file or directory
  ...
```

The problem here is that word splitting *is* applied to the result of *command substitution*, breaking the filenames produced by the `ls` call apart.

Lists and Whitespace (contd.)

You might surmise that the fix for that is easy, we simply need to double-quote command substitution to supress word splitting:

```
$ for file in "$(ls)"; do echo ls -i "$file"; done
ls: cannot access file 1 .txt
file 2 .txt
file 3 .txt: No such file or directory
```

Yes, we surpressed word splitting, but now we no longer have a *list* of filenames, rather, we have `ls`' output as a *single word:* the filenames *separated by newlines* (i.e., on separate lines).

Bottom line: list elements that contain whitespace can be problems, best to stick to filename expansion for producing filename lists.

The intricacies of word splitting are covered further in the **Bash Details and Advanced Features** lecture.

Word Splitting and Resetting IFS

The Bash variable `IFS` ("inter-field separator") is a string that controls which characters result in **word splitting**.

`IFS` and thus word splitting behavior can be easily changed.

The most common thing one might want to do is change `IFS` so that word splitting occurs *only* on *newlines*.

This is done with: `IFS=$'\n'`

Doing so can make it possible to work with lists produced by commands like `find`, `grep`, `awk`, `sed`, etc. that return results as *lines* (separated by newlines).

Word Splitting and Resetting IFS

By changing `IFS`, we can use `ls` to get the files:

```
$ IFS=$'\n'
$ for file in $(ls); do ls -i "$file"; done
2494675 file 1 .txt
2494676 file 2 .txt
2494677 file 3 .tx
```

Note that `IFS` has now been changed for our shell session!

It be set back to the default with: `IFS=$' \t\n'`

(Since a *shell script* runs in a *new subprocess*, changing `IFS` in scripts won't affect its value in the calling shell.)

Word Splitting and Resetting IFS

The freqently used command `find` has similar issues and solutions (since it returns its filename results separated by newlines):

```
$ for file in "$(find . -name "*.txt")"; do ls -i "$file"; done
ls: cannot access ./file 1 .txt
./file 3 .txt
./file 2 .txt: No such file or directory

$ IFS=$'\n'
$ for file in $(find . -name "*.txt"); do ls -i "$file"; done
2494675 ./file 1 .txt
2494677 ./file 3 .txt
2494676 ./file 2 .txt
```

Processing Text Files

Because shell scripts can use any of the large number of Linux/UNIX filters/utilities combined in pipelines, they are particularly appropriate for working with *text files*.

`grep` is one of the more heavily used filter utilities.

E.g., we can test if a file contains any lines matching a **regular expression** (**regex**):

```
if grep --quiet "$regex" "$file"; then
  ...work on file containing regex lines...
fi
```

We can also retrieve those lines for further processing:

```
lines=$(grep "$regex" "$file")
```

Processing Text Files (contd.)

The following `grep` *options* are particularly useful in shell scripts:

- `-q/--quiet` – Quiet; do not write anything to standard output. Exit immediately with zero status if any match is found.
- `-v/--invert-match` – Invert the sense of matching, to select *non-matching* lines.
- `-c/--count` – Print only a *count* of matching lines (or non-matching lines With the `-v` option).
- `-o/--only-matching` – Print only the matched (non-empty) parts of a matching line.
- `-m NUM/--max-count=NUM` – Stop reading file after `NUM` matching lines.
- `-n/--line-number` – Prefix each line of output with the 1-based line number from file.

Processing Text Files (contd.)

Some other commands/filters that are particularly useful in shell scripts dealing with text files (e.g., as part of pipelines) are:

- `cut OPTION... [FILE]...` – Print selected parts of lines from each `FILE` to standard output.
- `tr [OPTION]... SET1 [SET2]` – Translate, squeeze, and/or delete characters from standard input, writing to standard output.
- `xargs [OPTION]... COMMAND [INITIAL-ARGUMENTS]` – Reads *items* from standard input and executes COMMAND for each, using INITIAL-ARGUMENTS followed a read item.

AWK and SED are also heavily used for manipulating text files.

Reading Lines from Files

There are a number of approaches that can work for reading through the lines in a file, but several have issues with *leading whitespace* in lines and/or with *blank lines*.

The following approach will loop through all the lines in a file (whose path is in the variable `filepath`), placing each entire line into the variable `line`, and terminating at the file-end.

```
IFS=$'\n'
while read line; do
  echo "$line"
done < "$filepath"
```

If *leading whitespace* in lines is not an issue, resetting of `IFS` can be *eliminated*.

Reading Lines from Files (contd.)

Instead of using *input redirection*, `exec` can be used to open the file on a new **file descriptor**:

```
exec 3<"$filepath"
IFS=$'\n'
while read -u3 line; do
  echo "$line"
done
```

Assumes this is only file opened, so ends up on *file descriptor* 3.

Instead of "`read -u3 line`" one can do "`read line <&3`".

Dealing with File Paths

Shell scripts frequently have to deal with **file paths** (pathnames).

E.g., the script is in one directory, source files in another, and target is yet another directory.

Code can often be simplified by `cd`'ing to an appropriate directory.

This is because **filename expansion** patterns that include directory info, will produce results containing that directory info:

```
$ for file in dir/*; do echo "$file"; done
dir/file1.txt
dir/file2.txt
...
```

Dealing with File Paths (contd.)

However, users may provide file/directory path arguments to scripts using either **absolute** or **relative** paths.

Once a script `cd`'s, arguments using *relative paths* will be *invalid!*

This often requires a script to determine the *absolute paths* for its arguments and/or the original CWD.

Key commands for dealing with paths are:

- `basename` – strip directory (and maybe suffix) from filename
- `dirname` – strip last component from filename (leaving directory)
- `realpath` – print resolved absolute file path (follow symlinks)
- `which` – print absolute path of command/executable (as would be found per `PATH`)
- `pwd` – print CWD

Dealing with File Paths (contd.)

Get/save CWD absolute path, e.g., so can return after `cd`'ing:

```
cwdsave=$(pwd)
```

Get absolute path of current script (required to call *recursively*):

```
cmd=$(which $0)
```

(**Note:** *Debian distros* use non-GNU version of `which` that returns `./command` when given as argument, instead of returning absolute path; suggest you replace it with *much better* GNU `which`!)

Getting absolute path of a file/directory argument (`$1`):

```
abs=$(realpath "$1")
```

(`realpath` does not gurantee returned path exists unless use *option* `-e`/`--canonicalize-existing`.)

Dealing with File Paths (contd.)

Example of path issues: copy all files from directory `$1` to `$2`.

Approach without `cd`'ing:

```
for file in "$1"/*; do
  cp "$file" "$2"/"$(basename "$file")"
done
```

Approach `cd`'ing to source directory:

```
targetdir=$(realpath "$2")
cd "$1"
for file in *; do
  cp "$file" "$targetdir"
done
```

(Note many *double quotes* to handle names containing *whitespace*.)

Decoding Command Line Arguments

Most shell scripts will require arguments be passed to them when invoked on the command line.

We have seen that command line arguments can be accessed via the **positional parameters** (e.g., `$1`) and certain **special parameters** (particularly `$@`).

However, many scripts must deal with *variable numbers of arguments*, for example when they can handle any number of file arguments (including sometimes, no file arguments).

In addition, it makes sense for scripts to use the same kind of **options** notation that is used for standard Linux/UNIX commands.

Dealling with options and variable numbers of arguments can complicate the code required to decode command line arguments (retrieve and use their values).

Decoding Command Line Arguments (contd.)

One typical pattern in script syntax is some number of required arguments followed by an arbitrary number of arguments of a certain type: e.g., `myscript ARG1 ARG2 FILE...`

This pattern is easily handled by retrieving the fixed args, calling `shift` to remove them from the arguments lists, and then using `"$@"` in a `for-in` loop:

```
arg1=$1
arg2=$2
shift2
for file in "$@"; do
  ...code to process file (a FILE argument)...
done
```

Decoding Command Line Arguments (contd.)

Frequently, one wants a script to make all `FILE` arguments optional, and read from **standard input** (`STDIN`) if no `FILE` arguments are given: e.g., `myscript ARG1 ARG2 [FILE...]`

The straightforward way to allow this leads to code duplication:

```
arg1=$1
arg2=$2
shift2
if [[ $# -gt 0 ]]; then
  for file in "$@"; do
    ...code to process file (a FILE argument)...
  done
else
  ...code to process STDIN...
fi
```

Decoding Command Line Arguments (contd.)

Various people have come up with somewhat tricky approaches that can avoid code duplication, though many are quite hard to understand.

In truth, it is very simple to handle this situation:

```
arg1=$1
arg2=$2
shift2
for file in "${@:-/dev/stdin}"; do
  ...code to process file (a FILE argument or STDIN)...
done
```

Trick: `${param:-word}` form gives value of `param`, unless it is unset/null, then the expansion of `word` is substituted.

Decoding Command Line Arguments (contd.)

We can do something similar if we might have only a single file argument or read from `STDIN`: e.g., `myscript ARG1 ARG2 [FILE]`:

```
arg1=$1
arg2=$2
shift2
file="${1:-/dev/stdin}"
...code to process file (FILE or STDIN)...
done
```

(Thanks is owed to the various posters on *stackoverflow* for their suggestions about handling files vs. stdin, nonetheless, the solution on the previous slide, synthesized from those postings, is either simpler or more general than every single posted solution!)

Decoding Command Line Arguments (contd.)

As already noted, when scripts need to allow options to be specified, following the standard Linux/UNIX syntax makes the most sense since users should understand it..

To recap, there are two Linux/UNIX options styles:

- short/old-style options:
  - start with a single dash/hyphen (“-”)
  - are single letters (e.g., `-a`)
  - can take arguments (e.g., `-n10`)
- long/new-style options:
  - start with two dashes/hyphens (“--”)
  - are words (e.g., `--all` or `--almost-all`)
  - can take arguments (e.g., `--number=10`)

Decoding Command Line Arguments (contd.)

If a script must handle only a small number of options, they can probably be decoded with relatively simple code—particularly if they are required to come first, args cannot be separated by whitespace, etc.

Many standard Linux/UNIX commands support a large number of options and allow significant flexibility with options (short options can be combined (e.g., `-acl`), arguments can be adjacent or whitespace-separated (e.g., `-n10` or `-n 10`), and options may even be allowed to be interspersed with command parameters).

Writing code to handle many options and/or allow such flexbility, would be a significant undertaking!

Instead, most programs decode options by using the POSIX/GNU C functions `getopt()` and `getopt_long()`.

Decoding Command Line Arguments (contd.)

For Bash shell scripts, there are two options that provide similar capabilities to the `getopt()` and `getopt_long()` C functions.

- `getopts` – a Bash builtin command
- `getopt` – a standalone utility provide by kernel.org's util-linux package

`getopts` is builtin to Bash so will be available with every Bash, is not as complicated to use for short options, but was not designed for long options so requires a bit of fussing to handle them.

`getopt` was designed to be more portable (across different shells), and can handle long options, but may not be installed by default and is more complicated to use.

Decoding Command Line Arguments (contd.)

Consider a script with a small number of options that come first:
E.g., `myscript [-a|--all] [-nN|--number=N] ARG1 ARG2`:

Decoding this script's options could be done like:

```
all=false; num=10  #default settings
while [[ "$1" == -* ]]; do
  case "$1" in
    -a|--all)   all=true ;;
    -n*)        num=${1#-n} ;;
    --number=*) num=${1#--number=} ;;
    *)          echo "Invalid option: $1" ;;
  esac
  shift  #remove processed option
done
arg1=$1
arg2=$2
```

Decoding Command Line Arguments (contd.)

Noq consider that script with only short options:
E.g., `myscript [-a] [-nN] ARG1 ARG2`:

`getopts` could be used to decode options like:

```
all=false; num=10  #default settings
while getopts ":an:" opt; do
  case $opt in
    a)  all=true ;;
    n)  num=$OPTARG ;;
    \?) echo "Invalid option: $1" ;;
  esac
done
shift $((OPTIND-1))
arg1=$1
arg2=$2
```

Decoding Command Line Arguments (contd.)

Returning to the script with both short and long options we can modify our use of `getopts` to handle this too:

```
all=false; num=10  #default settings
while getopts ":an:-:" opt; do
  case $opt in
    a)  all=true ;;
    n)  num=$OPTARG ;;
    -)  #Handle possible long option:
       LONG_OPT=${OPTARG%=*}
       if [[ "$LONG_OPT" == "$OPTARG" ]]; then LONG_OPTARG=""
       else LONG_OPTARG=${OPTARG#*=}; fi
       case $LONG_OPT in
         all) if [[ -n "$LONG_OPTARG" ]]; then echo "opt error: extra_arg"; fi
              all=true ;;
         number) if [[ -z "$LONG_OPTARG" ]]; then echo "opt error: missing_arg"; fi
                 num=$LONG_OPTARG ;;
         *) opt_error "invalid_opt" ;;
       esac ;;
   \?) echo "opt error: invalid_opt" ;;
    :) echo "opt error: missing_arg" ;;
  esac
done
shift $((OPTIND-1))
arg1=$1
arg2=$2
```

Shell Scripting 7: Details & Advanced Features

1. Shells and Variables
2. Control Constructs
3. Shell Expansions
4. Functions and Arrays
5. Bash Development
6. Practical Advice
7. **Bash Details and Advanced Features**
   - **command interpretation details**
   - **exit status details**

Command Interpretation Details

When Bash reads and executes a command, it does the following:

1. Reads its input from a file or from the user's terminal.
2. Breaks the input into **words** and **operators**, obeying the **quoting rules**.
3. **Alias** expansion is performed (but not in scripts by default).
4. Parses the tokens into **simple commands** and **compound commands**.
5. Performs the various shell **expansions**.
6. Performs any necessary **redirections** and removes the redirection operators and their operands from the argument list.
7. Executes the commands.
8. Optionally waits for the command to complete and collects its exit status

Command Interpretation Details (contd.)

There are seven kinds of **expansion**:

1. brace expansion
2. tilde expansion
3. parameter and variable expansion
4. command substitution
5. arithmetic expansion
6. word splitting
7. filename expansion

The order of expansions is:
brace expansion; tilde expansion; parameter, variable, and arithmetic expansion and command substitution (done in a left-to-right fashion); word splitting; filename expansion.

Command Interpretation Details (contd.)

Only brace expansion, word splitting, and filename expansion can change the number of words of the expansion; other expansions expand a single word to a single word. The only exceptions to this are the expansions of `"$@"` and `"${name[@]}"` .

Variable assignment `var=[val]`:
Word splitting is not performed, with the exception of `"$@"`.

Test-command `[[ ... ]]`:
Word splitting and filename expansion are not performed on the words between the `[[` and `]]`; tilde expansion, parameter and variable expansion, arithmetic expansion, command substitution, process substitution, and quote removal are performed.

Command substitution:
If the substitution appears within double quotes, word splitting and filename expansion are not performed on the results.

Command Interpretation Details (contd.)

Word Splitting:
The shell scans the results of parameter expansion, command substitution, and arithmetic expansion that did not occur within double quotes for word splitting.

The shell treats each character of `$IFS` as a delimiter, and splits the results of the other expansions into "**words**."

The *default value* of `IFS` is: `<space><tab><newline>`

Word splitting behavior can be changed by changing the value of `IFS`.

E.g., doing `IFS=$'\n'` will cause splitting only on newlines.

This can be useful when capturing multiple lines which contain filenames that may include embedded whitespace.

Command Interpretation Details (contd.)

Word splitting details:

- If `IFS` is unset or the default, then any sequence of `IFS` characters serves to delimit words.
- If `IFS` has a value other than the default, then sequences of the whitespace characters space and tab are ignored at the beginning and end of the word, as long as the whitespace character is in the value of `IFS` (an `IFS` whitespace character).
- Any character in `IFS` that is not `IFS` whitespace, along with any adjacent `IFS` whitespace characters, delimits a field.
- A sequence of `IFS` whitespace characters is also treated as a delimiter.
- If the value of `IFS` is null, no word splitting occurs.
- Explicit null arguments (`""` or `''`) are retained.
- Unquoted implicit null arguments, resulting from the expansion of parameters that have no values, are removed.
- If a parameter with no value is expanded within double quotes, a null argument results and is retained.
- If no expansion occurs, no splitting is performed.

Exit Status Details

Many commands that are run in shell scripts can *fail*.

Writing *robust* shell scripts means including code that ensures commands have succeeded before proceeding in the script.

Otherwise, invalid data may be passed on, leading to incorrect results or unexpected and difficult to interpret error messages.

(Remember also that we may need to suppress error messages from commands by *redirecting standard error:* "`2>/dev/null`".)

Success/failure of commands is most commonly determined by their **exit status**.

While the exit status of a **simple command** is clear, this is not necessarily the case for other types of Bash commands.

Exit Status Details (contd.)

Pipelines are the most complex:

- pipeline exit status is the exit status of the *last command* in the pipeline
- if Bash `pipefail` *option* enabled, then is exit status of last (rightmost) command *to exit with a non-zero status*, or zero if all commands exit successfully
- if "`!`" symbol precedes the pipeline, exit status is the negation of the exit status
- shell *waits* for all commands in the pipeline to terminate before returning status

Enable or disable `pipefail` option with:
"`set -o pipefail`" or "`set +o pipefail`".

Exit Status Details (contd.)

Other types of commands have exit status as follows:

- **lists**:
  exit status is the exit status of the last command executed in the `;`/`&&`/`||` list
- **while/until**:
  exit status is the exit status of the last command executed in body, or zero/success if none was executed
- **for-in**: exit status of the last command that executes, but if empty list (no commands executed) then exit status is zero
- **for**: exit status of the last command executed in body; failure if any of the three `for` form expressions is invalid
- **if-elif-else**: exit status of the last command executed, or zero/success if no condition tested true
- **case**: zero/success if no pattern is matched, or exit status of the (one) `commands-list` that was executed