

CS265
Advanced Programming
Techniques

Bash Shell
Bash Scripting - Part 1

pushd and popd commands

Can be used to remember directories that we visit

- The `pushd` command saves the current working directory current in memory so it can be returned to at any time, optionally changing to a new directory.
- The `popd` command returns to the path at the top of the directory stack.
- The command `dirs` and `dirs -v` show the state of the stack

Bourne Again Shell (bash)

We'll teach `bash` in this course

- Extension of the `Bourne Shell`
- Contains many of the `Korn Shell (ksh)` extensions
- There are other shells: `tcsh` (Tenex C Shell), `ksh` (Korn Shell), `zsh`, `dash`

`bash` can be used as

- as a shell (a command line interface)
- as a scripting language

Login and Interactive Shells

- Each Unix shell can be interactive or non-interactive
- A shell can also act as a login vs a non-login shell
- In all cases, they all are the same program (`/bin/bash`)
- Login shells **.bash_profile**
 - The shell that starts when you first login to Unix
 - Where general setup happens (set environment variables, the search path, prompts, ...)
- Non-login shells **.bashrc**
 - Shells that are started by the user but after login
- Interactive shells
 - A shell that handles commands from a user at the keyboard
- Non-Interactive shells
 - A shell that handles commands from a file (scripting) or from programs like `ftp`, `scp`, etc.
 - Usually don't read the initial files

Bash Startup Files

- Place customizations in startup files
 - `/etc/profile` – system-wide
 - `/etc/bash.bashrc` – system-wide
 - `~/.bash_profile` – user
 - `~/.bashrc` – user
- Read the bash man pages to see when each is invoked

.bashrc



interactive non-login shells

.bash_profile



login shells

Shell as a Scripting Language

Has features commonly found in languages for structured programs

- Allows shell scripts to be used as filters
- Controls flow, variables
- Controls I/O file descriptors
- The environment allows context to be established at startup
 - Provides a way for scripts to pass information to processes without using positional parameters

Bash Customization

- The shell supports various customizations
- Set through shell options or environment variables
 - User prompt
 - Bindings for command-line editing
 - Aliases (shortcuts)
 - Functions – like little scripts (who run in the current shell)
 - Other behaviors

Interpreting Commands

- Shell prints a prompt, awaits a command
- When the shell gets a line of input
 - 1 It expands aliases (recursively)
 - 2 Checks to see if command is a shell builtin (or a function)
 - 3 If not, assumes it is a disk utility (e.g., `ls`)

Shell builtins

- A shell builtin is a command the shell will do for you
 - `cd` , `type` , `pushd` , `set` , `pwd` , ...
- They are faster
- The shell provides builtins for some common disk utilities
 - `alias` , `echo` , `printf` , ...
- Use a path to invoke the disk utility (`/bin/echo`)
- The builtin type will determine if a command is a builtin, or tell you where the utility is on disk
- The `help` builtin will give you help on any builtin, or show you all of the the shell builtins

Running Programs from Disk

- Disk programs are run in a subshell
- The subshell `execs` the program
 - Replaces itself with the program
- If the command isn't a shell builtin, the shell will search for a disk utility (using your `$PATH`)
- If the command token contains a path, then that utility will simply be run

```
$ /usr/bin/firefox & # kick firefox off in the background
```

```
$ /usr/bin/python myScript.py # invoke the python interpreter
```

Logging off

Use the `exit` builtin

- Exits the shell
- If it is the login (top-level) shell, then it disconnects you
- A shell is just another program
- Can recursively invoke shells
- Don't just disconnect without exiting
- ctrl-D (end-of-file) will also log you out
 - Unless you have the `ignoreeof` shell option set

Shell Metacharacters

Shell Metacharacters

- A **metacharacter** is a character that has special meaning to the shell
- Wildcards
* ? []
- I/O redirection
< > |
- Others
& ; \$ # ! n ()
- These characters must be escaped or quoted to inhibit their special behavior

```
ls "some file" another\&file `and;yet;a;third`
```

Wildcards

- Also known as **name globbing** and **filename expansion**
 - `*` . . . matches 0 or more characters
 - `?` . . . matches exactly 1 character
 - `[list]` . . . matches any single character from list
 - Wildcards are **not** regular expressions
- E.g.:
 - `ls *.cc` list all C++ source files in directory
 - `ls ?a*` list all files whose second letter is 'a'
 - `ls [a-cf]*.jpeg` list JPEGs that start with a, b, c, or f
 - `ls [!ac-e]*.jpeg` list all JPEGs that do not start with a, c, d, e
 - `ls *` try it with non-empty subdirectories present

Shell Variables

Called parameters

- bash uses shell variables to store information
- Used to affect the behavior of the shell, and other programs
- Simple mechanism, just stores text
- bash does have arrays and associative arrays (see `declare` builtin)

Setting & Viewing Parameters

- To assign a variable (in sh, ksh, bash)
- Note, no whitespace around the =

```
VAR=something  
OTHER_VAR="I have whitespace"
```

- Precede with \$ to view (dereference) a parameters:

```
$ echo $OTHER_VAR  
I have whitespace  
$ echo "My name is $USER"  
My name is dv35
```


Common Parameters

- **PATH** list of directories searched by shell for disk utilities
- **PS1** primary prompt
- **USER** user's login name
- **HOME** user's home directory
- **PWD** current working directory

Other Useful Shell Variables

- `$SHELL` The login shell
- `$$` The PID of the current shell
- `$?` The return value of the last command
- `TERM` Terminal type (what the shell thinks the terminal interface is)
- `HOSTNAME` Machine's hostname (see `uname`)
- `EDITOR` Some programs (mutt, sudoedit, git, etc.) might look here, when opening a text file
- `SHELLOPTS` Status of various bash options (see the `set` builtin)

\ – the escape character

- Use the backslash \ to inhibit the special meaning (behavior) of the metacharacter that follows.

```
$ echo $USER
```

```
dv35
```

```
$ echo \ $USER
```

```
$USER
```

- So, now \ is a metacharacter. Escape it to get just the character:

```
$ echo a\\b
```

```
a\b
```

\ followed by newline

- The backslash, when followed immediately by a newline, effectively removes the newline from the stream

```
$ echo On the bloody morning after\  
> One tin soldier rides away  
On the bloody morning after One tin soldier rides away
```

- Use quotes, if you want the newline in the output:

```
$ echo "On the bloody morning after  
> One tin soldier rides away"  
On the bloody morning after  
One tin soldier rides away
```

Weak Quoting

(The shell will interpret things before passing them to the command)

- Double quotes inhibit all but `\ ` $!`

```
$ echo "$USER is $USER"
```

```
dv35 is dv35
```

```
$ echo "\$USER is $USER"
```

```
$USER is dv35
```

```
$ echo "I said, \"Well, we shan't\""
```

```
I said, "Well, we shan't"
```

```
$ echo "It is now $(date +%H:%M) "
```

```
It is now 19:27
```

Strong quoting

(The shell will keep
its hands out)

- Single quotes preserve the literal value of all enclosed characters
- May not contain a single quote (can't be escaped)

```
$ echo 'I said, "Wait!"'
I said, "Wait!"
$ echo 'My name is $USER'
My name is $USER
```

Be careful when cutting & pasting single quotes

Bash Command Execution

Bash commands can be executed one at a time or they can be joined together in several ways

1. Sequenced
2. Grouped
3. Subshell Group
4. Conditional

1. Sequenced Commands

- Commands to be executed serially
- No direct relationship between them
- Commands can be separated by a newline or ;

```
cmd1 ; cmd2 ; cmd 3
```

```
$ echo a ; javac test.java
```


2. Grouped commands

```
{ cmd1 ; cmd2 ; cmd2 ; }
```

- Sequences can be grouped using {}
- List must be terminated by a ;
- Runs in the context of the current shell
- Useful for redirecting I/O
- Return value is the status of the last command executed
- Spaces are important to the right of { and to the left of }

```
$ echo a ; echo b ; echo c > out
a
b
$ cat out
c
```

```
$ { echo a ; echo b ; echo c; } > out
$ cat out
a
b
c
```

3. Grouped commands for subshell

```
( cmd1 ; cmd2 ; cmd2 )
```

- Sequences grouped with ()
- Also handy for redirecting I/O
- Runs in a subshell
- Can be run in the background
- No changes persist
- Return value is the status of the last command executed

```
$ y=30 ; ( y=20 ; echo $y ) ; echo $y
20
30
```

No spaces on either side of the = symbol

- `y = 30` won't work but `y=30` will

Assignment: No spaces before or after =

There should be no spaces before or after = when assigning a value

So this is correct

`a=b`

while this is not correct (note the spaces)

`a = b`

4. Conditional Commands

- Operators `&&` and `||`
- Conditional execution depends on the return value of the command on the left
- This value available to the caller (parent shell)
 - Look at special variable `$?` for return status of last command
- Value on `[0, 255]`
 - Zero (0) signals success; is true
 - We enumerate errors (failure), starting at 1
- `&&` and `||` have the same precedence, associate left-to-right

Conditional Execution Operators

`cmd1 && cmd2`

- `cmd1` is executed first
- If `cmd1` succeeds, then `cmd2` is executed

`cmd1 || cmd2`

- `cmd1` is executed first
- If `cmd1` fails, then `cmd2` executed

```
$ cp file1 file2 && echo "Copy succeeded"
Copy succeeded
$ cp no_such_file file2 2> /dev/null || echo "Copy failed"
Copy failed
```

Bash Scripting

What is a shell?

- A program that interprets your requests to run other programs
- A shell is a high-level programming language
- Most common Unix shells:
 - Bourne shell (sh)
 - C shell (csh - tcsh)
 - Korn shell (ksh)
 - Bourne-again shell (bash)
- In this course we focus on bash shell (`bash`)

What is a bash script?

- A sequence of bash commands
- A bash script is a program
 - Stored as a text file
 - Interpreted by the bash shell
 - Made up of
 - Variables
 - Control structures

Why write scripts?

- Convenience
- Sequences of performed operations that are performed often can be placed in a script, executed as a single command
- Shell provides access to many useful utilities

Shell scripts vs Java-like languages

- Shell scripts are generally not as well-suited to large tasks
 - they run more slowly
 - are more resource-intensive
 - do not give the programmer the same degree of control over resources
- Languages such as Java and C/C++ allow for much more structured programs

Shell Scripts

- Text files that contain one or more shell commands
- The first line is always identifying the interpreter (the shell) that will execute the script

`#!/bin/bash (sha-bang)`

or

`#!/bin/sh`

- Except on line 1, # indicates a comment

```
% cat welcome
#!/bin/bash
echo 'Hello World!'
```

Shell scripts must be executable

```
% welcome
```

```
welcome: Permission denied.
```

```
% chmod 744 welcome
```

```
% ls -l welcome
```

```
-rwxr--r-- 1 dv35 ...
```

```
% welcome
```

```
welcome: command not found
```

```
% ./welcome
```

```
Hello World!
```

command will run if the current directory (the . directory) is in the \$PATH

tells the shell to run the command from the . directory

Just like a command

```
% welcome > greet_them
```

```
% cat greet_them
```

```
Hello World!
```

What's in a shell script?

Bash Scripts are text files that include one or more shell commands. In addition, they can have

1. Shell Script Variables
2. Control structures

Schell Script Variables

Shell Script Variables

There are five possible types of shell script variables

1. Command-line arguments
2. Process-related variables
3. Environment variables
4. Shell variables
5. User-defined variables

1. Command Line Arguments

\$0 is the name of the script

\$1 is the first command-line argument

\$2 is the second command-line argument

...

\$# is the number of arguments

2. Process-related variables

\$?	The exit status of the last command 0 – successful execution of last command Non-zero – something went wrong
\$#	The number of arguments
\$*	All arguments
\$@	All arguments (individually quoted)
\${n}	The n-th positional argument
\$\$	The process ID (pid) of the shell

Redirection Tricks

- Want to run a command to check its exit status and ignore the output?

```
diff f1 f2 > /dev/null
```

- Want to ignore both standard error and standard output?

```
diff f1 f2 >& /dev/null
```

In Unix, `/dev/null` is a device that discards all data written to it.

3. Environmental Variables

- Contain information about the system
- Available in all shells
- Examples: \$USER, \$HOME, \$PATH
- To display your environment variables, type

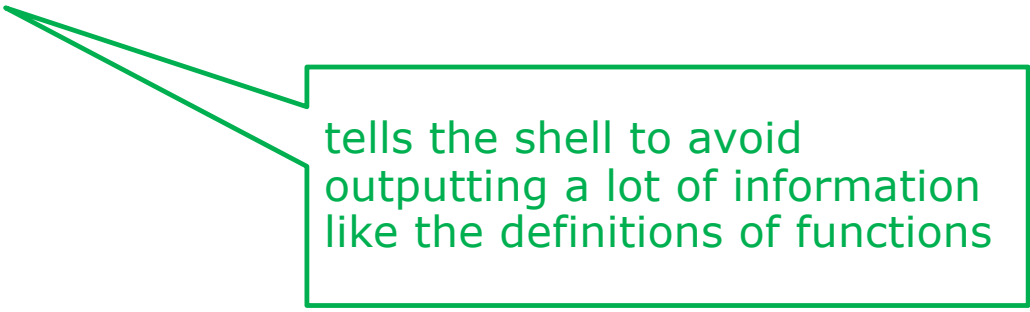
```
printenv
```

4. Shell Variables

- Used to tailor the current shell
- A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.
- Examples: `cwd`, `prompt`
- To display your shell variables, type

```
set -o posix
```

```
set
```



tells the shell to avoid outputting a lot of information like the definitions of functions

5. User-Defined Variables

- Variable name: combination of letters, numbers, and underscore character (_) that do not start with a number
- Avoid existing commands and shell/environment variables
- Assignment:

`name=value`

- No space around the equal sign!

Variables

- To use a variable: `$varname`
- Operator `$` tells the shell to substitute the value of the variable name

This topic continues to next week....