Command Line Time-Savers

- Bash Features and Options
- Processing the Command Line
- History Expansion
- Alias Substitution
- Brace Expansion
- Tilde ~ Expansion
- Parameter and Variable Expansion
- Arithmetic Expansion
- Command Substitution
- Word Splitting
- Pathname Expansion
- Process Substitution

Bash Features and Options

- There are a number of shell features that you can turn **on** and **off**
- One example is the ***noclobber*** option
- When this option is set, you *cannot overwrite* the contents of a file with redirected output
- To set a feature, use ***set -o*** followed by a *space* and the *feature name*

```
$ set -o noclobber
```

Bash Features and Options

- If I now try to redirect output to a file, the shell will *prevent* this

```
$ echo "Go Red Sox" > output.txt
bash: output.txt: cannot overwrite existing
file
```

- To *unset* a feature use ***set +o*** followed by a *space* and the *feature name*

```
$ set +o noclobber
```

Bash Features and Options

- I can *now* overwrite a file with redirection

```
$ cat output.txt
foo

$ echo "Go Red Sox" > output.txt

$ cat output.txt
Go Red Sox
```

- You can find a list of shell features and options in Sobell
- Shell features and options *will not* be on the final

Processing the Command Line

- The shell can *modify* what you enter at the command line
- It does this to provide features like **aliases**
- To do this properly, the shell must modify the command line *in a specific order*
- Otherwise, things could become terribly confused
- There are **10** different ways in which the shell can modify the command line

Processing the Command Line

- The order in which the shell performs them is as follows:

1. History Expansion
2. Alias Substitution
3. Brace Expansion
4. Tilde ~ Expansion
5. Parameter and Variable Expansion
6. Arithmetic expansion
7. Command substitution
8. Word splitting
9. Pathname expansion
10. Process substitution

History Expansion

- The first substitution Bash performs is history expansion
- History expansion occurs when you use the exclamation mark **!** in front of an **event** ID to recall a previous command from the history list

```
$ history 5
  540  cat output.txt
  541  echo "Go Red Sox" > output.txt
  542  cat output.txt
  543  echo foo
  544  history 5

$ !543
echo foo
foo
```

Alias Substitution

- After history expansion, Bash performs **alias** substitution

```
$ alias ll='ls -l'

$ ll
total 2
lrwxrwxrwx 1 it244gh man    34 Sep  6 21:09 it244 ->
   /courses/it244/f12/ghoffmn/it244gh
drwxr-xr-x 2 it244gh ugrad 512 Oct 27 09:16 work
```

- The shell evaluates an alias by substituting the **value** of an alias for the **name** of the alias

Brace Expansion

- After alias substitution, Bash performs *brace expansion*
- Braces **{ }** allow you to write *several strings* in *one operation*
- The braces contain strings of characters, separated by **commas**
- The shell expands a brace by *creating multiple strings* – one for each string contained in the braces

Brace Expansion

- If I wanted to create "foo" files numbered **1** to **5**, I could use braces expansion as follows

```
$ touch foo{1,2,3,4,5}.txt

$ ls
foo1.txt  foo2.txt  foo3.txt  foo4.txt  foo5.txt
```

- The shell expanded the braces to create as many files as there were strings inside the braces
  - The shell takes the string that appears before the braces
  - Sticks it in front of every string inside the braces
  - Followed by the text the follows the braces

Brace Expansion

- This creates many new strings on the command line
- The strings *inside* the braces can contain one or more characters, but each string must be *separated* from the others by a *comma*

```
$ touch {a,ab,abc}.txt

$ ls
abc.txt  ab.txt  a.txt
```

Brace Expansion

- There should not be *any* unquoted spaces or tabs within the braces!
- If there is, the expansion will not work properly

```
$ touch {b , bc, b c d}.txt

$ ls -l
total 0
-rw-r--r-- 1 it244gh ugrad 0 Nov 14 10:37 ,
-rw-r--r-- 1 it244gh ugrad 0 Nov 14 10:37 b
-rw-r--r-- 1 it244gh ugrad 0 Nov 14 10:37 {b
-rw-r--r-- 1 it244gh ugrad 0 Nov 14 10:37 bc,
-rw-r--r-- 1 it244gh ugrad 0 Nov 14 10:37 c
-rw-r--r-- 1 it244gh ugrad 0 Nov 14 10:37 d}.txt
```

Tilde Expansion

- After brace expansion, Bash performs *tilde expansion*
- Whenever Bash sees a tilde **~** by itself, it substitutes *the absolute address of your home directory*

```
$ echo ~
/home/it244gh
```

- Whenever Bash sees a **~** *followed by a Unix username*, it substitutes the absolute address of the home directory *of **that** account*

```
$ echo ~ghoffmn
/home/ghoffmn
```

Tilde Expansion

- If there is no username matching the string following the `~`, then no expansion is performed

```
$ echo ~xxx
~xxx
```

- There are two other tilde expansions

`~+` `~-`

Tilde Expansion

- When Bash sees **~+** , it substitutes the value of the *current* directory

```
$ pwd
/home/it244gh/work

$ echo ~+
/home/it244gh/work
```

- When Bash sees **~-** , it substitutes the value of the *previous* directory

```
$ pwd
/home/it244gh/work

$ cd

$ pwd
/home/it244gh

$ echo ~-
/home/it244gh/work
```

Parameter and Variable Expansion

- After tilde expansion, Bash performs *parameter and variable expansion*

```
$ echo $SHELL
/bin/bash

$ echo $?
0
```

- Notice that this expansion comes *after* alias expansion, so you can use variables and parameters when defining aliases

Arithmetic Expansion

- After parameter and variable expansion, Bash performs **arithmetic expansion**
- Unix treats everything on the command line as text *unless told otherwise*

```
$ echo 5 + 4
5 + 4
```

- Arithmetic expansion allows Bash to
  - Interpret characters as numbers and to
  - Perform ordinary *arithmetic* upon them

Arithmetic Expansion

- But, it does more than that
- Whenever bash sees the **$((** , it treats everything that follows as a number or an arithmetic operator, until it sees **))**

```
$ echo $(( 5 + 4 ))
9
```

- It then *evaluates* the arithmetic expression inside the double parentheses and *substitutes* the resulting numeric value for the entire **$((** … **))** expression

Arithmetic Expansion

- The rules for evaluating arithmetic expressions are the same as for the C programming language
- They are mostly what you would expect
- You can use variables within an arithmetic expression

```
$ a=5

$ b=3

$ echo $a $b
5 3

$ echo $(( $a - $b ))
2
```

Arithmetic Expansion

- Inside the arithmetic expression itself, you do not have to use a **$** to get the value of a variable

```
$ echo $a $b
5 3

$ echo $(( a * b ))
15
```

Command Substitution

- After arithmetic expansion, Bash performs **command substitution**
- Command substitution uses the following format

  `$(COMMANDS)`
- Where COMMANDS are any valid Unix commands
- The commands inside the **( )** are run in a subshell and the entire command substitution expression **$( )** — along with whatever is inside it— is replaced by the *output* of the commands

Command Substitution

- For example, if I wanted to set a variable to the current time and date, I could use

```
$ today=$(date)

$ echo $today
Tue Oct 25 17:00:07 EDT 2011
```

- There is an alternate format for command substitution
- You can place the command within back tics `` `...` ``

```
$ ls -l `which bash`
-rwxr-xr-x 1 root root 954896 2011-03-31 17:20 /bin/bash
```

Command Substitution

- Before running ***ls***, Bash first runs the command

```
which bash
```

- And *replaces* the command with the value returned by ***which***
- ***ls*** can now take **/bin/bash** as its argument
- Command substitution can be used inside double quotes

```
$ echo "Today is $(date +'%A, %B %d, %Y')"
Today is Wednesday, November 13, 2013
```

Command Substitution

- The *back tic* is the character you get by holding down the **Shift** key and pressing the same key you use for ~
- The back tics do not work in the TC shell and is easily mistaken for the single quote ' , so I will not use it in this course

Word Splitting

- After command substitution, Bash performs *word splitting*
- When Bash gets a command line, it splits the text into tokens
- Tokens are strings of characters, usually separated by whitespace
  - Spaces
  - Tabs
  - Newlines (carriage returns)
- But, there are some situations where you want *another* character to separate tokens

Word Splitting

- You can do this using the Unix keyword variable **IFS**
- **IFS** stands for **I**nternal **F**ield **S**eparator
- If you give **IFS** a value – such as the colon **:** — then it will be use to separate tokens
  - With the *read* command
  - With the *set* command
  - In command substitution
  - In variable substitution
- Word splitting *will not* be on the final

Pathname Expansion

- After word splitting, Bash performs **pathname expansion** , also known as **globbing**
- Pathname expansion is where you use **meta-characters** to specify one or more **pathnames**
- The metacharacters are used to create patterns that are called **ambiguous file references**
- The metacharacters are

? ***** **[ ]**

Pathname Expansion

- Here are some examples

```
$ ls t*
test1.txt  test2.txt  test3.txt

$ echo t*
test1.txt test2.txt test3.txt
```

Process Substitution

- After pathname expansion, Bash performs **process substitution**
  - Process substitution allows you to create a file *on the fly*
  - A command is run in a sub-shell, and the lines generated by that command are *treated as a "file"* which can be used by other Unix/Linux commands
- Process substitution uses the following format

  <(COMMAND)

Process Substitution

- The output of the command that appears between the parentheses is placed in a Unix structure called a *named pipe*
- Normal Unix pipes connect the *output* of one command to the *input* of another command
  - Each command runs *inside its own process*, so a pipe allows one process to talk to another
  - This is called *interprocess communications*
  - Normal Unix pipes are also known as *anonymous* pipes because they have no name
  - Anonymous pipes only last as long as it takes for the *first* command to talk to the *second* command

Process Substitution

- *Named* pipes can last longer than anonymous pipes
- They can actually be *created* and *removed* at the command line
- When Unix performs process substitution…
  - It *creates a process* to run a command
  - And *sends the output* of that command to a named pipe
- Then, Unix redirects input to come from the named pipe

Process Substitution

- We can use named pipes to compare two directories:

```
$ diff -y <(ls -1 tia777/ce) <(ls -1 jgreen/ce)
ce1                             ce1
ce10                            ce10
ce11                          <
ce2                             ce2
ce3                             ce3
ce4                             ce4
ce5                             ce5
ce6                             ce6
ce7                             ce7
ce8                             ce8
ce9                             ce9
```

  - Here we have two ***ls*** commands, each *running* in their own subshell and each one *sending* output to its own named pipe
  - We can then run ***diff*** to look for differences in these two "files"

- Process substitution *will not* be on the final