IT 244: Introduction to Linux/Unix
Class 27

Today's Topics

Review

Running a Command in the Background

- Normally, when you run a command ...
- you have to wait for it to finish
- Such commands are said to be running in the **foreground**
- Unix gives you a way to get the command prompt back ...
- after running a command that will take a long time to finish
- You can run the command in the **background**
- The background job loses it's connection to the keyboard ...
- and the shell will give you a prompt
- The shell will tell you when the background job has finished
- Every time a program runs, a **process** is created
- The process has access to system resources ...
- like memory (RAM) and a connection to the filesystem
- Unix, like most OSs, is a multitasking operating system
- This means you can have more than one process running at a time
- To run a command in the background ...
- enter an ampersand, **&** , at the end of the command line ...
- just before hitting Enter:

```
$ sleep 5 &
[1] 17895
$
```

Jobs

- Every time you type something at the command line ...
- and hit Enter ...
- you are creating a **job**
- Every time a program runs ...
- a process is created for that program
- A pipeline is a collection of commands joined by pipes
- Each command will generate its own process ...
- but the collection of all the separate processes ...
- is a single job
- Each process in a pipeline will have its own process ID
- So as the pipeline progresses, the currently running process will change ...
- but the job number does not change
- The job is the collection of all processes created at the command line
- You can have multiple jobs running at the same time ...
- but only one job can be in the foreground at any one time
- Every process has a process ID number ...
- and every job has a job number
- When you tell the shell to run a job in the background ...
- it returns two numbers:

```
$ sleep 5 &
[1] 7431
$
```

- The job number is enclosed in brackets and comes first
- The second, larger, number is the process identification number ...
- of the first process in the job
- The process identification number is also known as the PID
- When the job finishes, the shell prints out a message

```
[1]+  Done                    sleep 5
```

- The message does **not** appear the moment the job finishes
- The shell waits for the next time you hit Enter ...
- and it prints the message after the output from the command

Moving a Job from the Foreground into the Background

- There can only be one foreground job ...
- though you can have many background jobs
- Unix will let you move a job from the foreground ...
- to the background
- To do this, you must first suspend the foreground job
- A suspended job is not dead ...
- it is in a state of suspended animation
- You can reactivate it later
- To suspend a foreground job you must use the suspend key sequence
- On our systems you suspend a job by hitting Control Z
- After you do this, the shell stops the current process
- It also disconnects it from the keyboard
- Once the job is suspended ...
- you can place it in the background using the *bg* command
- *bg* stands for **b**ack**g**round
- Once placed in the background, the job resumes running
- If more than one job is running ...
- you must give *bg* the job number

Aborting a Background Job

- There are two ways to abort a background job
- You can bring a job from the background to the foreground ...
- using the *fg* (**f**ore**g**round) command
- Once you have the job in the foreground ...
- you can abort it using Control C
- When there is more than one job in the background ...
- you must specify the job number when using *fg*
- You can also terminate any job using the *kill* command

- But to use *kill* you must tell it what to kill
- The usual way to do this is to give *kill* a process ID
- If you don't remember the process ID ...
- run *ps* (**p**rocess **s**tatus) to get the process ID (PID)
- You can also use the job number with *kill* ...
- but you must precede a job number with a percent sign, **%**
- You can get the job number by using the *jobs* command

Pathname Expansion

- **Pathname expansion** allows you to specify a file or directory ...
- without typing the full name
- It also allows you to specify more than one file or directory ...
- with a single string of characters
- Pathname expansion uses characters with special meaning to the shell
- These special characters are called **meta-characters**
- Meta-characters are also sometimes called **wildcards**
- They allow you to specify a pattern
- When the shell sees one of these characters on the command line ...
- it replaces the pattern with a sorted list ...
- of all pathnames that match the pattern
- The shell then runs this altered command line
- The pattern is called an **ambiguous file reference**
- You can use as many meta-characters as you want to form a pattern
- Pathname expansion is different from **pathname completion** ...
- which you get by hitting Tab

The ? Meta-character

- The question mark, **?** , meta-character stands for a single instance of any character
- **?** can be used with any command ...
- even those that don't normally deal with files

```
$ echo dir?
dir1 dir2 dir3 dir4
```

- The **?** meta-character **does not** match a leading period in a filename
- You must explicitly enter a leading period, **.** ...
- when specifying an "invisible" file

The * Meta-character

- An asterisk, ***** , will match any number of characters in a pathname
- It will even match no characters
- ***** can be used with any command ...
- even those that don't normally deal with files

```
$ echo dir*
dir dir1 dir10 dir2 dir3 dir4
```

- * cannot be used to match the initial period, . , in a hidden filename
- But you can list all the hidden file in a directory using * ...
- if you put it after a period

```
$ ls .*
.addressbook   .bashrc  .forward  .pinerc
.bash_history  .cshrc   .login    .plan
.bash_profile  .emacs   .msgsrc
```

The [and] Meta-characters

- The square brackets, [and] , are also meta-characters
- They work somewhat like ?
- They only match a single character in a pathname ...
- but the pathname character must match **one** of the characters ...
- within the brackets
- No matter how many characters are within the bracket ...
- the pattern can match only a **single character**
- You can use the bracket meta-characters with any program
- You can use a range to avoid listing all characters
- A range is specified by listing the first and last characters of a sequence ...
- separated by a dash, -
- The sequence is specified in alphabetical order
- The square brackets provide another shortcut
- If you insert an exclamation mark, ! , or a caret, ^ ...
- immediately after the opening bracket ...
- the shell will match any single character ...
- that is NOT included within the brackets

Built-ins

- Not all commands can be found on disk as executable files
- Some are actually written as part of the shell
- Such commands are called **built-ins**
- When you run a built-in ...
- the shell does not have to create a new **process** ...
- when you run these programs
- Instead the command runs in the same process as the shell
- This makes execution faster

Ways a Shell Can Be Created

- There are three ways a user can create a shell

  - Login shell
  - Interactive non-login shell
  - Non-interactive shell
- There are subtle differences between these three types of shells

Your Login Shell

- The login shell is the shell you get ...
- after your password has been accepted
- Each login session has one, and only one, login shell
- Your default shell version you run is set ...
- when your account is created
- The **absolute pathname** of this shell is contained in the variabale SHELL
- When your login shell starts up ...
- it runs the commands found in /etc/profile
- This is a file customized by the system administrator ...
- for all users
- You can create your own customizations in a **startup file** ...
- in you home directory
- The file **must** have one of these names
  - .bash_profile
  - .bash_login
  - .profile
- We will use .bash_profile

Interactive Non-login Shells

- You can run another shell from your login shell
- This sub-shell is called an interactive non-login shell

```
$ ps
  PID TTY          TIME CMD
12778 pts/1    00:00:00 bash
12969 pts/1    00:00:00 ps

$ bash

$ ps
  PID TTY          TIME CMD
12778 pts/1    00:00:00 bash
12970 pts/1    00:00:00 bash
12973 pts/1    00:00:00 ps

$
```

- Notice that the first entry for *bash* ...
- has the same process ID each time you run *ps*
- This is your login shell

- The second *bash* process is the sub-shell
- So your login shell is still running ...
- but you are talking to a sub-shell ...
- and the login shell is sleeping ...
- waiting for the sub-shell to finish
- the sub-shell is **not** a login shell
- It is an interactive non-login shell
- An interactive non-login shell is a shell that you create ...
- without having to enter a password
- Interactive non-login shells have their own startup file ...
- called .bashrc ...
- and it must be in your home directory

Non-interactive Shells

- A shell scripts is a file containing Unix commands
- When you run this file, all the commands in the file are executed
- The program that understands these commands and runs them ...
- is a shell
- So your current shell has to create a sub-shell ...
- to run the commands in the shell script
- This sub-shell does not give you a prompt ...
- so it is not an interactive shell
- It is a non-interactive shell
- There is no standard startup file for such a shell

Creating Startup Files

- A startup file contains Unix commands ...
- that are run just before you get a prompt
- The startup file normally used by Bash is .bash_profile
- This file must be placed in your home directory

Running a Startup File after a Change has been Made

- Usually, when you change a startup file ...
- you want the changes to take place immediately
- But if you made a change to .bash_profile ...
- the changes won't take effect until the next time you login
- Unix gives you a way to make the changes take effect immediately
- You do this by running the *source* command

```
source .bash_profile
```

- *source* runs a Unix script in the current shell ...
- **not** a subshell

Commands that are Symbols

- Unix has some commands that are symbols rather than words
- I'll just mention them now and go into greater detail in future classes

( )	Runs whatever commands are enclosed in the parentheses in a sub-shell
$( )	Command substitution: rruns the commands enclosed in the paretheses in a subshell and returns their value to the command line, replacing the dollar sign, the parentheses and everything in them with this value
$(( ))	Arithmetic expansion: evaluates an arithmetic expression and returns its value at that place on the command line
[ ]	The test command: used to evaluate a boolean expression in constructs like if clauses

File Descriptors

- Every time the shell creates a **process** ...
- it gives that process a connection to three "files"
  - Standard input
  - Standard output
  - Standard error
- A program can open other files besides these
- **File descriptors** are data structures that Unix creates ...
- to handle access to files
- File descriptors are the abstract representation ...
- of the files that are connected to a process
- Each file descriptor is assigned a positive number ...
- starting with 0
- Think of a file descriptor as an integer that refers to a file
- **Standard input**, **standard output** and **standard error**
- each have their own file descriptors

Name	File Descriptor
Standard input	0
Standard output	1
Standard error	2

- So while we think of standard input, standard output and standard error ...
- Unix thinks of the file descriptors 0, 1 and 2

Redirecting Standard Error

- Standard error is the "file" into which error messages are sent
- Redirecting standard error allows a program to separate its output stream ..
- from its error messages
- To redirect standard input we use the less than symbol, <
- followed by a pathname
- This construction is really a shorthand ...
- for a notation using file descriptors
- When you type

  ```
  ./repeat.sh < test.txt
  ```

  Unix thinks of this as

  ```
  ./repeat.sh 0< test.txt
  ```

  where 0 is the file descriptor for standard input
- Similarly, when we use output redirection

  ```
  $ echo "Hello there"  >  hello.txt
  ```

  Unix thinks of this as meaning

  ```
  $ echo "Hello there"  1>  hello.txt
  ```

- Again the file descriptor precedes the redirection symbol, >
- So how do we redirect standard error?
- We place a 2 in front of the greater than symbol

  ```
  $ ls xxxx
  ls: cannot access xxxx: No such file or directory

  $ ls xxxx 2> error.txt

  $ cat error.txt
  ls: cannot access xxxx: No such file or directory
  ```

- Remember, 2 is the file descriptor for standard error
- Unix also gives you a way to redirect **both** standard output and standard input ...
- to the same file
- You can do this using the ampersand and greater than symbols together, **&>**

Shell Scripts

- A shell script can use any shell feature
- that is available at the command line
  - Ambiguous file references using the metacharacters **?** , ***** and **[ ]**
  - Redirection

- Pipes
- But not those features which are provided by *tty*
  - **Command line editing** (arrow keys, control key combinations)
  - **Pathname completion** (hit Tab to get more of a filename)
  - The history mechanism (up arrow to recall previous command line)
- Unix also provides control structures
  - If statements
  - Loops

Making a Shell Script Executable

- You must have both **read** and **execute** permission ...
- to run a shell script
- Because the shell has to read the contents of the script ...
- you need read permission
- You need execute permission so the script can actually be run ...
- without calling *bash*
- Normally you would give a shell script file 755 permissions
- The owner can read, write and execute
- The group and everyone else can read and execute

Specifying Which Version of the Shell Will Run a Script

- When the shell runs a shell script ...
- it creates a new shell ...
- inside the process that will run the script
- Normally this **sub-shell** will be the same version of the shell ...
- as your login shell
- A script can use the **hashbang** line ...
- to specify which shell version to use ...
- when running a script
- The hashbang line **must** be the first line of the script
- The first two characters on the line
- **must** be a hash symbol, # ...
- followed by an exclamation mark, **!**
- After these two characters, you need to have the **absolute pathname** ...
- of the version of the shell which will run the script

Comments in Shell Scripts

- Scripts have to be read by the people
  - Who write the program
  - Who maintain the program
  - Who use the program
- To make clear what is happening inside a program ...

- use comments
- Anything following a hash mark, # , is a comment ...
- except for the hashbang line

Separating and Grouping Commands

- You can enter many commands on a single command line ...
- if you separate them with a semi-colon, ;

```
$ echo Here are the contents of my home directory  ;  ls  ;  echo
Here are the contents of my home directory
error.txt  foo  it244  work
```

- When you hit Enter each command is executed ...
- in the order it was typed at the command line

| (pipe) and & (ampersand) as Command Separators

- The pipe, | , and ampersand, & , characters are also command separators
- When we separate commands with the pipe character, | ...
- each command takes its input from the previous command
- We use an ampersand, & , after a command ...
- to run that command in the **background**
- When we do this, two things happen
    - The command is disconnected from the keyboard
    - The command will run at the same time as the next command you enter at the terminal
- But the ampersand is also a command separator
- So we can use it to run many commands at the same time

Continuing a Command onto the Next Line

- If you want to coninute a command line entry onto another line ...
- You can typ a backslash, \
- followed **immediately** by the Enter key

```
$ echo A man \
> A plan \
> A canal \
> Panama
A man A plan A canal Panama
```

- After hitting \ and newline ...
- the shell responds with the greater than symbol, >
- This is the **secondary prompt**
- The shell is telling you it expects more input

Using Parentheses, ( ) , to Run a Group of Commands in a Sub-shell

- A group of commands within, a longer command line ...
- can be given a sub-shell of their own ...
- in which to run
- You can do this by putting the commands within parentheses

```
( cd ~/bar ; tar-xvf - )
```

- The shell creates a sub-shell and runs the commands in that sub-shell

Shell Variables

- A **variable** is a place in memory with a name ...
- that holds a value
- To get the value of a variable ...
- put a dollar sign, **$** , in front of its name
- Some variables are set and maintained by the shell itself
- They are called **keyword shell variables** ...
- or just keyword variables
- Other variables are created by the user
- They are called are called user-created variables
- The environment in which a variable can be used is called the **scope**
- Shell variables have two scopes
  - Local
  - Global

Local Variables

- **Local variables** only exist in the shell in which they are defined
- To create a local variable, use the following format

```
VARIABLE_NAME=VALUE
```

- There are **no spaces on either side of the equal sign** ...
- when setting Bash variables
- Variables are local unless you explicitly make them global
- If the value assigned to a variable has spaces or tabs ...
- you must quote it
- If you run a shell script, the local variables will not be visible ...
- because the script is running in a sub-shell ...
- and the local variables are not defined there

Global Variables

- **Global variables** are defined in one shell ...

- and keep their values in all sub-shells created by that shell
- Global variables are defined in Bash using the *export* command
- like .bash_profile
- The *env* command, when used without an argument ...
- displays the values of all global variables

Keyword Shell Variables

- Keyword shell variables have special meaning to the shell
- They have short, mnemonic names
- By convention, the names of keyword variables are **always capitalized**
- Most keyword variables can be changed by the user
- This is normally done in the startup file .bash_profile

Important Keyword Shell Variables

- There are a number of keyword variables that affect your Unix session
- Some of the more important are

Variable	Value
HOME	The absolute pathname of your home directory
PATH	The list of directories the shell will search when looking for the executable file associated with a command you entered at the command line
SHELL	The absolute pathname of your default shell
PS1	Your command line prompt - what you see after entering each command
PS2	The secondary prompt - what you see if you continue a command to a second line

User-created Variables

- User-created variables are any variables you create
- By convention, the names of user-created variables are lower case
- User-created variables can be either local or global in scope

Quoting and the Evaluation of Variables

- Whenever the value of a variable contains spaces or tabs ...
- you must quote the string or escape the whitespace character
- There are three ways this
  - Single quotes (' ')

  - Double quotes (" ")
  - Backslash (\ )
- Everything surrounded by single quotes ...
- appears in the variable exactly as you entered it
- Double quotes also preserve spaces and tabs ...
- in the strings they contain
- But you can use a **$** in front of a variable name ...
- to get the value of a variable ...
- inside double quotes
- Quotes affect everything they enclose
- The backslash, \ , only effects the character immediately following it

Removing a Variable's Value

- There are two ways of removing the value of a variable
- You can use the *unset* command
- Or you can set the value of the variable to the empty string

Processes

- A **process** is a running program
- Unix is a multitasking operating system
- Many processes can run at the same time
- The shell runs in a process like any other program
- Every time you run a program ...
- a process is created ...
- except when you run a **built-in**
- When you run a shell script ...
- your current shell creates a sub-shell to run the script
- This sub-shell runs in a new process
- When each command in the script is run ...
- a process is created for that command

Process Structure

- Processes are created in a hierarchical fashion
- When the machine is started, there is only one process
- This process is called init
- init then creates other processes
- These new processes are child process of init
- These child processes can create other processes
- init has PID (Process ID) of 1
- init is the ancestor of every other processes ...
- that ever runs on the machine

Process Identification

- Each process has a unique Process ID (PID) number
- *ps -f* displays a full listing of information about each process running for the current user

```
$ ps -f
UID         PID  PPID  C STIME TTY          TIME CMD
it244gh  26374 26373  0 13:41 pts/5    00:00:00 -bash
it244gh  27891 26374  0 13:57 pts/5    00:00:00 ps -f
```

- The UID column shows the user's Unix username
- The PID column is the process ID of the process
- The PPID column is the process ID of the parent process ...
- the process that created this process
- The CMD column gives the command line that started the process

Executing a Command

- When you run a command from the shell ...
- the shell asks the operating system to create a process ...
- to run the command
- Then it sleeps ...
- waiting for the child process to finish
- When the child process finishes ...
- it notifies its parent process of its success or failure ...
- by returning an **exit status** ...
- and then it dies

History

- The history mechanism maintains a list of the command lines you type
- These command line entries are called events
- To view the history list, use the *history* command
- If you run *history* without an argument ...
- it will display all the events in this history list
- By default, this list contains 500 values
- To restrict how many lines are displayed ...
- run *history* followed by a number
- You cannot have a **-** in front of the number ...
- as there **must** be when using *head* or *tail*

Using the History Mechanism

- If you know the event number of a previous command ...
- you can run it again by using an exclamation mark, **!** ...

- followed by the event number

```
$ !517
echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

- The history mechanism prints out the old command line ...
- before running it
- There must be **no space between the ! and the number** ...

The Readline Library

- The readline library is a collection of procedures ...
- which let you edit the command line
- When you use Control key combinations on the command line ...
- you are using the readline library
- Any program running under Bash and written in C can use the readline library
- Here are some of the more useful commands for the emacs version of the readline library

Command	Meaning
Control A	Move to the beginning of the line
Control E	Move to the end of the line
Control U	Remove everything from the text entry point to the beginning of the line
Control K	Remove everything from the text entry point to the end of the line
→	Move the text entry point one character to the left
←	Move the text entry point one character to the right
↑	Recall the previous command line entry in the history list
↓	Recall the following command line entry in the history list

Readline Completion

- The readline library provides a completion mechanism
- Type a few letters of something ...
- and readline completion will try to supply the rest
- There are three forms of completion provided by the readline library
  - **Pathname completion**
  - **Command completion**

- Variable completion

Pathname Completion

- The readline library provides pathname completion
- You begin typing a pathname, then hit Tab
- If there is only one pathname that matches ...
- the readline library will provide the the rest of the pathname
- If there is more than one possible completion ...
- the readline library will beep
- You can then enter more characters ...´
- before hitting Tab again ...
- or you can hit Tab right after the first beep ...
- and the readline library will give you a list of possible completions
- If the second Tab still gives you a beep ...
- there are no possible completions

Command Completion

- The readline library will complete the name of a command for you
- Begin typing a command ...
- then hit Tab ...
- and the readline library will try to supply the rest of the command
- If there is more than one possibility ...
- you will hear a beep
- If you hit Tab a second time ...
- you will see a list of possible completions

Variable Completion

- When you type a dollar sign, **$** , followed immediately by some text ...
- you are entering a variable name
- The readline library knows this
- and will attempt to complete the name of the variable
- If there is more than one possibility, you will hear a beep
- If you then hit Tab another time
- you will see a list of possible completions
- If no list appears after the second Tab ...
- there are no possible variable name completions

Aliases

- An **alias** alias is a string ..
- that the shell replaces with some other string
- Usually, the value assigned to the alias ...

- is a command or part of a command
- To define an alias, you use the *alias* command
- *alias* uses the following format in Bash

```
alias ALIAS_NAME=ALIAS_VALUE
```

- In Bash, there must be **no spaces on either side of the equal sign,** =
- If the value assigned to the alias has spaces, it must be quoted
- If you follow *alias* with the name of an alias, it will display the definition

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

- In Bash, an alias cannot accept an argument
- Instead of allowing Bash to have aliases that accept an argument ...
- Bash has functions
- Aliases are **not** global
- They only work in the shell in which they are defined

Single Quotes Versus Double Quotes in Aliases

- Usually you want to use single quotes ...
- when defining alias
- If you use single quotes any variables in the alias ...
- will be evaluated **when you use the alias**
- If you use double quotes, any variable in the alias ...
- will be evaluated **when it is defined**

Functions

- A function is a collection of commands that is given a name
- Functions **can** accept arguments from the command line
- Functions can be run anywhere you happen to be in the filesystem ...
- because function exist in memory ...
- not on the disk
- Functions, unlike aliases, can have arguments
- They use the same positional arguments that shell scripts use
- Functions differ from shell scripts in a number of ways
  - They are stored in memory (RAM), rather than in a file on disk
  - The shell preprocesses the function so it can execute more quickly
  - The shell executes the function in it's own process
- Functions are local to the shell in which they are defined
- They do not work in subshells
- Functions definitions have the following form

```
FUNCTION_NAME ()
{
```

```
        COMMANDS
    }
```

- For clarity you can precede the function name ...
- with the keyword *function*
- The keyword *function* is optional

Shell Modification of the Command Line

- But before the shell executes the commands ...
- if first looks to see if it needs to make changes ...
- to the **tokens** on the command line
- The shell actually rewrites the command line before executing it
- It does this to implement features of the shell ...
- like **command substitution** and **pathname expansion**
- These are features that make the shell more powerful ...
- but they require the shell to change what you typed on the command line ...
- before executing it
- There are 10 different ways in which the shell can modify the command line

History Expansion

- History expansion occurs when you use the exclamation mark, **!** ...
- to run again a previous command using the history mechanism

```
$ 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
- The shell replaces the name of the alias ...
- with the value of the alias
- Aliases allow you to run complicated commands ...
- by typing only a few characters

Brace Expansion

- Braces, **{ }** , allow you to specify several strings ...

- all at once
- Braces can appear with strings of characters in front ...
- or behind
- The braces contain strings of characters **separated by commas**
- The shell expands a brace by creating many strings ...
- one for each string contained within the braces
- If I wanted to create 5 foo files 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
```

~ Expansion

- Whenever *bash* sees a tilde, **~** , by itself ...
- it substitutes the absolute pathname of your home directory
- Whenever *bash* sees a tilde, **~** , followed by a Unix user name,
- it substitutes the absolute pathname of the home directory ...
- of that account

Parameter and Variable Expansion

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

```
$ echo $SHELL
/bin/bash
```

- bash notices the **$** in front of a string ...
- and looks to see if that string is the name of a variable
- If the string is a variable, bash subsitutes the value of the variable ...
- for the dollar sign and variable name

Arithmetic Expansion

- Unix treats everything on the command line as text
- except in a few situations
- **Arithmetic expansion** is where the text inside **$(( ))** ...
- is treated as an arithmetic expression ...
- and the result of evaluating that expression replaces **$(( ))** ...
- and everything inside it

Command Substitution

- In command substitution, a command is run in a sub-shell ...
- and the output of that command ...

- replaces the command itself
- Command substitution uses the following format

```
$(COMMAND)
```

- Where COMMAND is any valid Unix command

Pathname Expansion

- 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 meta-characters are
    - **?**
    - *****
    - **[ ]**

Shell Script Control Structures

- Control structures are Unix statements that change the order of execution ...
- of commands within a program or script
- There are two basic types of control structures
    - Loops
    - Conditionals

The *if ... then ...* Construct

- The first conditional is the *if* ... *then* statement ...
- which has the format

```
if COMMAND
then
    COMMAND_1
    COMMAND_2
    ...
fi
```

- COMMAND is **any** Unix command
- COMMAND_1, COMMAND_2, ... are a series of Unix commands
- The most common comman used with *if* is *test*
- which must be followed by arguments that form a logical expression
- It is used to test the truth or falsity of a condition
- The keyword *fi* must close the conditional statement
- The statements between *then* and *fi* are executed ...
- depending on the status code ...
- given by the command that follows *if*

- If the command following *if* runs without error ...
- then it will return an exit status of 0 ...
- which the *if* ... *then* statment treats as true

test

- The *test* command evaluates a logical expression ...
- given to it as an argument ...
- and returns a status code of 0 ...
- if the expression evaluates to true
- It returns a status code of 1 ...
- if the expression evaluates to false
- In an *if* statement, a status code of 0 means true ...
- and a status code greater than 0 means means false

The *test* operators

- *test* has a number of operators
- The operators test for different conditions
- When used with two arguments, the operators are placed between
- Some operators work only on numbers

Operator	Condition Tested
-eq	Two numbers are equal
-ne	Two numbers are not equal
-ge	The first number is greater than, or equal to, the second
-gt	The first number is greater than the second
-le	The first number is less than, or equal to, the second
-lt	The first number is less than the second

- *test* uses the different operators when comparing **strings**

Operator	Condition Tested
=	When placed between strings, are the two strings the same
!=	When placed between strings, are the two strings not the same

- Note that *test* uses symbols (=) when comparing strings
- But letters preceded by a dash (**-eq**) when comparing numbers
- There are two additional operators ...
- that *test* uses when evaluating two expressions
- It is placed between the two expressions

Operator	Condition Tested
-a	Logical AND meaning both expressions must be true
-o	Logical OR meaning either of the two expressions must be true

- The exclamation mark, **!** is a negation operator
- It changes the value of the following logical expression
  - It changes a false expression to true
  - And a true expression to false

Using *test* in Scripts

- Bash provides a synonym for *test* ...
- a pair of square brackets, **[ ]**
- To test whether the value of the variable number1
- is greater than the value of the variable number2
- you could write either

```
test $number1 -gt $number2
```

or

```
[ $number1 -gt $number2 ]
```

- Whenever you use this construction
- there **must** be a space ...
- before and after each square bracket

The *if ... then ... else ...* Construct

- The another conditional is the *if ... then ... else ...* construct ...
- which has the following format

```
if  COMMAND
then
  COMMAND_1
  COMMAND_2
  ....
else
  COMMAND_A
  COMMAND_B
  ...
fi
```

- If COMMAND returns an exit status of 0 ...
- COMMAND_1, COMMAND_2, ... will be executed ...
- otherwise COMMAND_A, COMMAND_B, ..., will be run

The *if ... then ... elif ...* Construct

- The *if ... then ... elif ...* construct allows you to create nested *if* statements

```
if COMMAND
then
  COMMAND_1
  COMMAND_2
  ...
elif ANOTHER_COMMAND
then
  COMMAND_A
  COMMAND_B
  ...
...
else
  COMMAND_N1
  COMMAND_N2
  ...
fi
```

- *elif* stands for "else if"
- Notice that *elif* must be followed by *then*
- The *then* must either be on the next line ...
- or the same line separated by a semi-colon, **;**
- The *else* statement must be terminated by a *fi*
- *elif* only requires a single *fi* at the end

Debugging Scripts

- If you run a script using *bash* with the -x option ...
- the shell will print each line of the script ...
- just before it executes that line
- Before *bash* prints a line from the script ...
- it prints a plus sign, +
- to let you know that the line is not output from the script

for ... in ... Loops

- Bash provides many kinds of loops,
- but we'll start with the *for ... in* loop

```
for LOOP_VARIABLE in LIST_OF_VALUES
do
    COMMAND_1
    COMMAND_2
    ...
done
```

- The block of loop commands start after the *do* keyword ...
- and end just before the *done* keyword
- The *do* keyword is like the *then* keyword in an *if* statement
- With a *for ... in* loop, Bash
  - Assigns the first value in the LIST_OF_VALUES ...
    to the variable specified by LOOP_VARIABLE
  - Executes the block of commands between *do* and *done*
  - Assigns the next value in the LIST_OF_VALUES
    to the LOOP_VARIABLE
  - Executes the commands between *do* and the *done* again
  - And so on until each value in LIST_OF_VALUES has been used

for Loops

- The *for* loop has a simpler structure than the *for ... in ...* loop

```
for LOOP_VARIABLE
do
    COMMAND_1
    COMMAND_2
    ...
done
```

- The difference between the two *for* loops ...
- is where they get the **values** for the loop variable
- The *for ... in ...* loop gets its values ...
- from the list that appears right after the *in* keyword
- These values are "**hard coded**" into the script
- They never change
- The simple *for* loop gets its values from the command line
- This *for* loop can have different values each time it is run

Three Expression *for* loops

- The first two *for* loops are totally different ...
- from the for loops in programming languages
- In programming languages, the *for* statement
  - Initializes a loop variable
  - Tests the current value of the loop variable ...
    to determine whether the loop should continue
  - Changes the loop variable at the end of the loop code
- The *for* statements in programming languages ...
- **create** the values used in the loop
- The two *for* loops we just have studied ...
- **must be given** the values used in the loop
- But there is a third form of *for* loop in Bash
- This form creates loop values the same way ...

- as the for loop in programming languages
- It has the following form

```
for (( EXP1; EXP2; EXP3 ))
do
    COMMAND_1
    COMMAND_2
    ...
done
```

- Notice that the three expressions after the *for* keyword ...
- are inside double parentheses
- That means that anything inside them will be treated as numbers
- The first expression sets the inital value of the loop variable
- The second is a logical expression
- As long as it is true, the loop will continue
- The third expression changes the value of the loop variable ...
- after each pass through the loop

while Loops

- The first two *for* loops keep running ...
- until all values supplied to them ...
- have been used in the loop
- A *while* loop continue running ...
- as long as the command following the keyword *while*
- returns a status code of 0
- *while* loops have the form

```
while COMMAND
do
    COMMAND_1
    COMMAND_2
    ...
done
```

- As long as COMMAND is returns and exit status of 0 ...
- the code block between *do* and *done* will be executed

until Loops

- The *until* loop is similar the *while* loop
- Except that the *until* loop ends ...
- when the command following *until* returns an exit status of 0
- The *while* loop stops ...
- when the exit status is **not** 0
- The *until* loop has the form

```
until COMMAND
do
    COMMAND_1
    COMMAND_2
    ...
done
```

- In practice, the *while* loop is used much more often than the *until* loop

continue

- Normally, a loop will execute all the commands ...
- between *do* and *done* ...
- in each pass through the loop
- But sometimes, you want to skip all or part ...
- of the commands in the loop block ...
- under specific conditions
- *continue* causes the shell to stop running the rest of the code ...
- between the *do* and *done* keywords
- The script then returns to the top of the loop ...
- and begins the next iteration
- *continue* does not cause the script to break out of the loop
- It merely stops the execution of the loop code ...
- for one iteration

break

- When you start a loop, you specify the conditions ...
- which will cause the loop to end
- With *for ... in* and simple *for* loops, the code leaves the loop ...
- when every value in the argument list has been used
- In the *while*, *until* and three expression *for* loops ...
- the code exits the loop when a logical condition is met
- But what if you encountered some unusual condition ...
- and wanted to break out of the loop entirely?
- To do this, you would have to use *break*
- When bash comes across the *break* keyword ...
- it jumps out of the loop

***case* Statement**

- Sometimes you want to write code that takes one of several different paths ...
- depending on the value of a single variable
- You could do this with an *if ... elsif* statement ...
- but a *case* statement is easier to use ...
- in this situation

- The *case* statement has the following format

```
case TEST_VARIABLE in
  PATTERN_1)
    COMMAND_1A
    COMMAND_1B
    COMMAND_1C
    ...
    ;;
  PATTERN_2)
    COMMAND_2A
    COMMAND_2B
    COMMAND_2C
    ...
    ;;
  PATTERN_3)
    COMMAND_3A
    COMMAND_3B
    COMMAND_3C
    ...
    ;;
...
esac
```

- When *bash* encounters a *case* statement it
  - Finds the first pattern that matches the test variable
  - Runs the statements for that pattern
  - Leaves the *case* statement
- Notice
  - There is a right parenthesis, **)** , after each pattern
  - The statements for each pattern end with two semi-colons, ;;
  - *esac* marks the end of the *case* statement
- *esac* is *case* spelled backwards
- This ***** will match anything that has not matched a previous pattern
- When creating patterns you can use the meta-characters and the logical OR

*****	Matches any string of characters
?	Matches any single character
[ ]	Every character within the brackets can match a single character in the test string
|	Logical OR separates alternative patterns

read **Command**

- The *read* command sets a variable ...
- to a value entered by the user at the terminal
- When *bash* comes across a *read* command it

  - Waits for the user to enter some text at the terminal
  - Assigns the text entered by the user to the variable ... whose name follows the *read* command
- When *read* takes in a value from the terminal ...
- it grabs everything the user types ...
- until they hit Enter
- I can use the -p option to *read*
- to have *read* issue a prompt
- By default, the *read* command does not allow you to edit text at the terminal ...
- the way you can a the command line
- But you can enable the readline library to make it possible to edit the line ..
- if you use the -e option to *read*

Using Braces, { } , with Variables

- If we try to concatenate the value of a variable with a string, we run into problems

```
$ dir=hw

$ echo The directory is $dir11

The directory is
```

- *bash* did not see the variable dir next to the string "11"
- Instead, it saw a new variable, dir11, which was not defined
- Since this variable was not defined ...
- it has no value
- To concatenate the value of a variable with a string ...
- we need to use braces, **{ }**
- The braces surround the name of the variable
- They set off the name of the variable from surrounding text

```
$ echo The directory is ${dir}11
The directory is hw11
```

- The opening brace comes right after the dollar sign, **$**

Special Parameters

- Special parameters are **shell variables** whose values ...
- are automatically set by Bash
- The parameters contain information about the current shell environment
- They are very useful when writing shell scripts
- Bash sets the values of these parameters ...
- based on the state of current environment

? - The Exit Status

- The ? special parameter returns the exit status of the last command

Positional Parameters

- Positional parameters give the value of the command line arguments ...
- to a shell script
- They can also be used with functions

- The Number of Command Line Arguments

- The # positional parameter contains the number of command line arguments
- This parameter allows you to check ...
- if your script has received all the arguments it needs

0 - The Pathname of the Script

- The 0 positional parameter contains the full pathname used to call the script
- You should use this parameter when creating a usage message
- But you should use it with the *basename* command ...
- to remove the path part of the pathname

1 - n - The Command Line Arguments

- The numbered positional parameters are used to give command line arguments ...
- to a script ...
- or to a function
- If there is no corresponding command line argument ...
- the parameter will have no value

shift: Promotes Command Line Arguments

- The *shift* built-in promotes command line arguments
- This means that the value of positional parameter 2 ...
- is assigned to positional parameter 1 ...
- and the value of positional parameter 3 ...
- is assigned to positional parameter 2 ...
- and so on
-
- After *shift* is called, all arguments move up one position ...
- and the first argument value is lost
- If *shift* is called with a numeric argument ...
- all arguments are moved up that number of positions
- The first two command line arguments are lost ...
- after *shift* is called ...
- and every positional parameter moves up 2 positions

- *shift* comes in handy when you want to write a script ...
- that loops over all command line arguments
- *shift* keeps promoting arguments until there are no more left