Advanced Shell Usage II

- Separating and Grouping Commands
- | (pipe) and & (ampersand) as Command Separators
- Continuing a Command onto the Next Line
- Using Parentheses ( ) to Run a Group of Commands in a Subshell
- The Directory Stack
  - *dirs* - Displays the Directory Stack
  - *pushd* - Pushes a Directory onto the Stack
  - *popd* - Pops a Directory off the Stack
- Shell Variables
  - Scope
    - Local Variables
    - Global Variables
  - Keyword Shell Variables
  - Important Keyword Shell Variables
  - User-created Variables

Separating and Grouping Commands

- Every time you hit **Enter** , the shell tries to execute your command line entry
- So far, we have only run one command from the command line at a time – except for pipes
- You can enter *more than one* commands on the command line, if you separate them with a semi-colon **;**
- For example…

Separating and Grouping Commands

```
$ cd  ;  echo  "Here are the contents of my home directory:" ; ls  -al  ~
Here are the contents of my home directory:
total 45
drwxr-xr-x 6 it244gh libuuid  1024 2011-10-16 12:47 .
drwxr-xr-x 5 root    root        0 2011-10-16 14:00 ..
-rw-r--r-- 1 it244gh libuuid     0 2011-09-07 11:06 .addressbook
-rw------- 1 it244gh libuuid  2285 2011-09-07 11:06 .addressbook.lu
-rw------- 1 it244gh libuuid  6979 2011-10-15 20:34 .bash_history
-rw-r--r-- 1 it244gh libuuid    44 2011-10-12 15:33 .bash_profile
-rw-r--r-- 1 it244gh libuuid    38 2011-10-11 15:38 .bashrc
-rw-r--r-- 1 it244gh libuuid    16 2011-10-12 14:27 .bash_rc
drwx------ 2 it244gh libuuid   512 2011-09-10 18:30 .cache
-rw-r--r-- 1 it244gh libuuid  1826 2011-09-14 10:00 .cshrc
...
```

- 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 semi-colon **;** is a command separator
- It separates multiple commands on a single command line
- These characters are *also* command separators:
  - The pipe character **|**
  - The ampersand character **&**

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

- When we separate commands with the *pipe* | character…
  - Each command takes its input from the *previous* command
  - Each command is a *separate process*, though the pipeline is a *single job*
- We use an *ampersand* **&** after a command to make the command run in the **background**

```
./bother.sh  &
```

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

- 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 *more than one* command at the same time…

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

- For example:

```
$ ./bother.sh > /dev/null  & ./bother.sh > /dev/null &
./bother.sh > /dev/null & jobs
[1] 1794
[2] 1795
[3] 1796
[1]   Running                 ./bother.sh > /dev/null &
[2]-  Running                 ./bother.sh > /dev/null &
[3]+  Running                 ./bother.sh > /dev/null &
```

- Here, we created *three* jobs with *one* command line

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

- We can kill *all three jobs* using *command substitution*

```
$ kill $(jobs -p)  ;  jobs
[1]   Running                 ./bother.sh > /dev/null &
[2]-  Running                 ./bother.sh > /dev/null &
[3]+  Running                 ./bother.sh > /dev/null &
[1]   Terminated              ./bother.sh > /dev/null
[2]-  Terminated              ./bother.sh > /dev/null
[3]+  Terminated              ./bother.sh > /dev/null
```

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

- Notice that we used the semi-colon to run *two* commands on the same command line
  - Since both these jobs are running in the foreground, they run *sequentially* one right after the other
  - Each command has to wait for the *previous* command to finish before it starts

Continuing a Command onto the Next Line

- Unix will let you type as long a command line as you like
- If you reach the end of your session window while typing, a command your text will *wrap* to the next line:

```
$ echo asdfasdfasdfasdfasdfasdfasdfasdf
asdfasdfasdfasdfasdfasdfasdfasdfasdfasd
fasdfasdfasdfasdfasdfasdfasdfasdfasdfas
dfasdfasdfasfads Done

asdfasdfasdfasdfasdfasdfasdfasdfasdfasd
fasdfasdfasdfasdfasdfasdfasdfasdfasdfas
dfasdfasdfasdfasdfasdfasdfasdfasdfasdfa
sdfasfads Done
```

Continuing a Command onto the Next Line

- If you then *expand* the window, you can see more text on each line:

```
$ echo asdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfas
dfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasd
fasdfasdfasfads Done

asdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfa
sdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfasdfas
dfasfads Done
```

- Unix thinks that the text above is only *two* lines
  - Your command line entry
  - The one line output

Continuing a Command onto the Next Line

- But…sometimes it helps to break a long command into more than one line
- You can do this by typing a backslash **** followed ***immediately*** by the **Enter** key

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

Continuing a Command onto the Next Line

- Here, we are escaping the newline character at the end of the line
  - Escaping turns off the special meaning of a character
  - The backslash above *turns off* the special meaning of newline – which is *normally* for the shell to run the command when it sees newline
- The newline character is sent when you hit
  - **Enter** on a PC
  - **Return** on a Mac

Continuing a Command onto the Next Line

- Backslash only escapes the character **immediately** following it
  - This trick *won't* work if you put a space before the newline
  - Then the backslash only operates on the *space*, not the newline
- After hitting **** and newline, the shell responds with the greater than symbol **>**
- This is the **secondary prompt** , which means that the shell is telling you it expects more input

Continuing a Command onto the Next Line

- The normal prompt is your primary prompt
  - You get the *primary* prompt when the shell is waiting for a command
  - You get the *secondary* prompt when the shell is waiting for the continuation of a command *already started*

Use () to Run a Group of Commands in a Subshell

- Sometimes, you want to run a group of commands in a shell of its own
  - 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
  - *Why* would you want to do this?

Use () to Run a Group of Commands in a Subshell

- Consider the following command line entry

```
cd ~/foo ; tar -cf - . | ( cd ~/bar ; tar-xvf - )
```

- It tells the shell to…
  - Go to a certain directory
  - Run ***tar*** on the files you find there
  - Send the results to standard input **-**
  - Go to another directory
  - Recreate the files from standard input **-**

Use () to Run a Group of Commands in a Subshell

- When using **`tar`**, **-** means either standard *input* or standard *output*, depending upon the context
- Without the sub-shell…
  - the output of the first **`tar`** would go to **`cd`**, and…
  - **`cd`** would *ignore* it, since it already has the only parameter it needs
- But…the second **`tar`** is waiting for something from standard input

The Directory Stack

- Moving back to a previous directory can be a pain
  - You might have to type a *long pathname*
  - Even worse, you might *forget* where you were
- Bash provides the *directory stack* mechanism to make this easier
  - The directory stack keeps tracks of each directory you enter by putting them onto a stack
    - The stack operates on the principle of last in, first out
    - You can go back to a previous directory by using, and removing, the last directory from the stack
  - You can keep doing this until you get where you want to be

The Directory Stack

- There are *three* commands that use the directory stack:
  - ***dirs***
  - ***pushd***
  - ***popd***
- ***cd*** keeps no record of where you have been
- ***pushd*** and ***popd*** use the directory stack to change your location and update the directory stack

dirs **- Displays the Directory Stack**

- A stack is a list with unusual properties
- It is a **LIFO** data structure, which stands for

  Last **I**n **F**irst **O**ut

- Stacks are a well-known data structure in programming
  - They allow you to go back in time to previous values of some important variable
  - A physical example of a stack can be found in some cafeterias in the dish stack…

dirs **- Displays the Directory Stack**

- The dish stack:
  - There is a circular hole in the counter which opens onto a metal cylinder with a spring at the bottom
    - An attendant puts a bunch of dishes into the cylinder
    - The next customer takes a dish from the top of the stack
    - That dish was put in last
  - *Last in, first out*

dirs **- Displays the Directory Stack**

- ***dirs*** displays the *current* contents of the directory stack
- If the stack is empty, ***dirs*** simply displays the current directory

```
$pwd
~/it244/hw5

$ dirs
~/it244/hw5
```

- ***dirs*** always uses a tilde **~** when referring to *your home directory*

pushd **- Pushes a Directory onto the Stack**

- In programming, putting something onto a stack is called a *push*
- ***pushd*** changes your current directory, just like ***cd***, but it also *adds* your new directory to the directory stack
- When used **with** an argument, ***pushd***
  - *Places* the new directory on the stack
  - *Displays* the current contents of the directory stack
  - *Moves* to the new directory

pushd - Pushes a Directory onto the Stack

- Let's look at an example:

```
$ pwd
/home/it244gh/it244/hw5

$ dirs
~/it244/hw5

$ pushd ~ghoffmn

$ pwd
/home/ghoffmn

$ dirs
/home/ghoffmn ~/it244/hw5
```

pushd **- Pushes a Directory onto the Stack**

- When used **without** an argument ***pushd***
  - *Swaps* the positions of the first two directories on the directory stack
  - *Displays* the current contents of the directory stack
  - *Moves* to the new top directory the directory stack
- Let's look at an example…

```
$ pushd examples_it244/
~/examples_it244  ~
```

pushd - Pushes a Directory onto the Stack

```
$ pushd examples_it244/
~/examples_it244  ~

$ pushd ~it244gh
/home/it244gh  ~/examples_it244  ~

$ pushd it244/work/
/home/it244gh/it244/work /home/it244gh  ~/examples_it244  ~

$ pushd
/home/it244gh  /home/it244gh/it244/work  ~/examples_it244  ~

$ dirs
/home/it244gh  /home/it244gh/it244/work  ~/examples_it244  ~
```

pushd **- Pushes a Directory onto the Stack**

- You can also give ***pushd*** a plus sign followed by a *number*
- If you do this, it will take you to the directory *at that position* in the stack
- The directory on the top of the stack has the number **0**

popd **- Pops a Directory off the Stack**

- In programming, removing a value from a stack is called a *pop*
- ***popd*** changes your current directory to another directory, but it also *removes* a directory from the stack
- When used **without** an argument ***popd***
  - *Removes* the top directory from the stack
  - *Prints* the current stack
  - *Goes* to the directory it removed from the stack

popd - Pops a Directory off the Stack

- Here is an example:

```
$ pwd
/home/it244gh/it244/hw5

$ dirs
~/it244/hw5

$ pushd ~ghoffmn
/home/ghoffmn  ~/it244/hw5

$ pwd
/home/ghoffmn
...
```

```
...

$ popd
~/it244/hw5

$ pwd
/home/it244gh/it244/hw5

$ dirs
~/it244/hw5
```

popd **- Pops a Directory off the Stack**

- You can also give ***popd*** a plus sign followed by a number
- The directory with that number will be removed from the stack, but you will *stay in the current directory*

Shell Variables

- A variable is a name given to a place in memory that *holds a value*
- **Shell variables** are variables that are defined inside a shell and can be used inside the shell
- To get the value of a shell variable, put a dollar sign **$** in front of the variable name

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

Shell Variables

- Some variables are set and maintained by the shell itself
- They are called **keyword shell variables**
  - Or just **keyword variables**
  - A **keyword** is a word with special meaning to the shell
  - Some of these variables are defined in `/etc/profile`
  - Many of these keyword variables can be changed by the user
- Other variables can be created by the user. They are called are called *user-created* variables

Shell 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 have meaning in the shell in which they are defined
- To create a local variable, use the following format

```
VARIABLE_NAME=VALUE
```

- There cannot be **any** spaces – on **either** side of the equal sign – when setting Bash variables. ***Example:***

```
$ foo=bar

$ echo $foo
bar
```

Local 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

```
$ hello='Hello there'

$ echo $hello
Hello there
```

- Local variables only exist in the shell in which they are created

Local Variables

- If you run a shell script, that script cannot see your local variables because:
  - the script is running in a sub-shell, and...
  - the local variables are only defined in the shell that *launched* the script

```
$ foo=bar

$ echo $foo
bar

$ cat print_foo.sh
#!/bin/bash
#
# Prints the value of the
variable foo

echo foo = $foo

$ ./print_foo.sh
foo =
```

Local Variables

- Notice that the script printed no value for `foo`

```
$ ./print_foo.sh
foo =
```

- The variable `foo` is defined **only** in the shell which **calls** `print_foo.sh`
- It **does not exist** in the sub-shell that runs the script so it has no value in the sub-shell

Local Variables

- Bash allows you to assign a value to a variable *used* in a script on the command line that *calls* the script:

```
$ echo $foo
bar

$ foo=bletch ./print_foo.sh
foo = bletch

$ echo $foo
bar
```

- Notice that the value of `foo` is different in the *running* script from its value in the shell that *launched* the script
- A variable defined at the command line, before running a script, ***only*** exists in the *sub-shell* that runs the command

Global Variables

- **Global variables** are variables that are
  - defined in *one* shell and…
  - have meaning in all sub-shells created *from* that shell
- In ***Bash***, you define a *global* variable by preceding the variable name with the keyword **`export`**

```
$ echo $foo
bar

$ export foo=bletch

$ echo $foo
bletch

$ ./print_foo.sh
foo = bletch
```

Global Variables

- Usually, global variables are declared in a startup file like **.bash_profile**
- If you run the ***env*** command *without* an argument, it prints the values of global variables

```
$ env
TERM=xterm-color
SHELL=/bin/bash
SSH_CLIENT=66.92.76.
9 53785 22
OLDPWD=/home/it244gh
SSH_TTY=/dev/pts/8
USER=it244gh
...
```

Keyword Shell Variables

- *Keyword shell variables*, also called *keyword 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 ones 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
MAIL	The absolute pathname of the file that holds your mail
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 that *you* create
- By convention, the names of user-created variables are *lower case*

```
$ foo=bling

$ echo $foo
bling
```

- User-created variables can be either *local* or *global* in scope