Advanced Shell Usage III.B

- The Readline Library
  - Readline Completion
  - Pathname Completion
  - Command Completion
  - Variable Completion
- Aliases
  - Single Quotes Versus Double Quotes in Aliases
  - Examples of Aliases
- Functions
- Where to Define Variables, Aliases and Functions

The Readline Library

- The **Readline** library is a collection of procedures, written in the **C** programming language, that let you edit the command line
- The Readline library was created by the GNU project
- 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

The Readline Library

- There are *two* modes available in the Readline library
  - ***emacs*** mode
  - ***vim*** mode
- **it244a** is configured to use ***emacs*** mode by default
- That is the mode we have been using
- Here are some of the more useful commands for the ***emacs*** version of the Readline library...

Readline Completion: Commands

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 and hit **`Tab`**, 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**
- They all use **`Tab`** to complete something, but they differ in **what** they complete

Pathname Completion

- *Pathname completion* is where you start to type a **pathname** and then hit **`Tab`** to have the Readline fill in the rest
- If you type the first few characters of a pathname and then hit **`Tab`**, then the Readline library will try to supply the rest
- If there is only one pathname that matches, then the Readline library will provide the rest of the pathname

Pathname Completion

- If there is more than one possible completion, then you will hear a 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

```
$ ls hw[Tab][Tab]
hw2/ hw4/ hw5/ hw6/
```

- If the second **Tab** still gives you a beep, then there are no possible completions

Command Completion

- The Readline library will complete the name of a *command* for you
- Begin typing a command, and then hit **Tab**
- If there is more than one possible completion, you will hear a beep
- If you hit **Tab** a second time, you will see a list of possible completions...

Command Completion

```
$ e[Tab][Tab]
e2freefrag                  elfedit                     esac
e2fsck                      elif                        etags
e2image                     else                        etags.emacs23
e2label                     emacs                       ethtool
e2undo                      emacs23                     eval
e4defrag                    emacs23-x                   ex
ebrowse                     emacsclient                 exec
ebrowse.emacs23             emacsclient.emacs23         exit
echo                        enable                      expand
ed                          enc2xs                      expiry
edit                        env                         export
editor                      envsubst                    expr
editres                     eqn                         extcheck
egrep                       erb
eject                       erb1.8
```

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

```
$ bar=BLETCH

$ echo $b[Tab]ar
BLETCH
```

- If there is more than one possibility, you will hear a beep

Variable Completion

- If you then hit **Tab** another time, you will see a list of possible completions

```
$ foo1=FOO; foo2=BAR


$ echo $foo[Tab][Tab]
$foo1   $foo2
```

- If no list appears after the second **Tab**, then there are no possible variable name completions

Aliases

- An **alias** is a string that the shell replaces with *some other string* when you use it on the command line
- Usually, the value assigned to the alias is *a command* or a *part of a command*
- You may want to get a long listing for a directory, and typing ***ls -l*** is quite a few characters
- So, you can define an alias: **ll**

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

Aliases

- Then, if you want a long listing, you can simply type **`ll`** instead of ***`ls -l`***

```
$ ll
total 45
-rwxr-xr-x 1 ghoffmn grad     38 Oct 11 20:05 border.sh
-rwxr-xr-x 1 ghoffmn grad    135 Oct 16 08:35 bother.sh
-rwxr-xr-x 1 ghoffmn grad     13 Oct 29 14:23 cheer.sh
-rwxr-xr-x 1 ghoffmn grad    103 Oct  9 08:53
     command_name.sh
-rwxr-xr-x 1 ghoffmn grad     99 Oct 29 16:15
     comment_test.sh
...
```

Aliases

- To define an alias, you use the ***alias*** command
- ***alias*** uses the following format in Bash

```
alias  ALIAS_NAME=ALIAS_VALUE
```

- There must be **no spaces** on **either** side of the equal sign **=** when defining an alias in Bash
- If the string assigned to the alias has *spaces*, then it must be quoted

```
alias  la='ls -a'
```

Aliases

- If you run ***alias*** with *no arguments*, then it will list all aliases currently defined:

```
$ alias
alias bin='pu $bin'
alias binl='ls $bin'
alias ck755='ls -l *.sh | tr '\''-'\'' '\'' '\'' |
grep '\''rwxr xr x'\'''
alias ckhb='head -1 *.sh | grep  /bin/bash'
alias cl='pu $cl'
alias clhws='pu $clhws'
alias clhwsl='ls $clhws'
alias cll='ls $cl'
alias clr='clear'
...
```

Aliases

- If you follow ***alias*** with the *name* of an alias, then it will display the definition

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

- In Bash, an alias cannot accept an argument, but it can in the TC shell
- Although an alias cannot accept an argument in Bash, *an argument can **follow** it*
- This is a subtle point…

Aliases

- You could use the `ll` alias and follow it with the name of a directory
- So, if you were to type the following at the command line...

```
ll /home/ghoffmn
```

- ...then the shell would substitute "`ls -l`" for `ll`, and the shell would then execute the *changed* command line

```
ls -l /home/ghoffmn
```

- But, what if I wanted to create an alias for a pipe using two commands, and I needed to pass an argument to the first command?

Aliases

- For example, if I wanted to create an alias for something like this

```
ls -l DIRECTORY_NAME | head
```

- In Bash, I **cannot** create an alias like this...

```
alias llh='ls -l $1 | grep txt'
```

  because an alias will not accept an argument

- When Bash comes across an alias, it substitutes the **value** of the alias for the **name** of the alias

`Aliases`

- So, you can't give an argument to an alias in Bash – unless the argument comes **after** the alias
- Instead of allowing aliases to accept arguments, Bash has *functions*
  - Functions in ***`bash`*** can consist of many commands, and you can use arguments with each of these commands
  - We'll discuss functions a little later in this class
- The TC shell has no functions

Aliases

- You can't use the name of an alias inside the value of an alias
- In other words, an alias cannot call itself
- If you defined an alias, and then used the *name* of the alias in the *value* of the alias, how would Bash know when to stop?
- In other words, if you tried to do something like this...

```
alias foo='foo foo foo'
```

Aliases

- ...the alias would try to call itself, and that call to the alias would try to call itself, and you would have an *infinite recursive loop*
- To keep this from happening, an alias *will not work* if it calls itself...

```
$ alias foo='echo foo'

$ foo
foo

$ alias foo='foo  foo  foo'

$ foo
foo: command not found
```

Aliases

- Aliases are **not** global. They *only* work in the shell in which they are defined

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

$ bash

$ ll
ll: command not found
```

- The ***alias*** command is a *built-in*

```
$ type alias
alias is a shell
builtin
```

- This makes sense since an alias *only* works in the shell in which it is defined
- If ***alias*** were not a built-in, it would be defined in the subshell that ran the ***alias*** command, not your current shell.

Single Quotes Versus Double Quotes in Aliases

- There are two types of quotes in ***bash***
  - Single quotes - ' '
  - Double quotes - " "
- Both single and double quotes allow you to assign a variable a value that *contains whitespace*

```
$ name='Glenn Hoffman'

$ echo $name
Glenn Hoffman
```

Single Quotes Versus Double Quotes in Aliases

- The whitespace characters are:
  - Spaces
  - Tabs
  - Newlines (carriage returns)
- Single quotes turn off all special meanings of characters

```
$ echo 'My name is $name'
My name is $name
```

- Variables are **not evaluated** when they are enclosed in single quotes

Single Quotes Versus Double Quotes in Aliases

- But, double quotes allow you to use the **$** in front of a variable, to get the value of the variable:

```
$ echo "My name is $name"
My name is Glenn Hoffman
```

- Usually, when defining aliases, you want to use *single* quotes
- If you use single quotes when *defining* an alias, any variables in the alias value will be evaluated when you *use* the alias, which is usually what you want

Single Quotes Versus Double Quotes in Aliases

- If you use double quotes when defining an alias, then any variable in the alias will be evaluated when it is **defined,** not when it is used
- *Consider the following:* The **PWD keyword variable** is used by the shell to keep track of your current directory

```
$ pwd
/home/it244gh

$ echo $PWD
/home/it244gh
```

Single Quotes Versus Double Quotes in Aliases

- This means that the value of **PWD** *changes* as you move about the filesystem
- Let's see what happens if we define an alias using **PWD** inside *double* quotes

```
$ alias where="echo My
current location is
$PWD "
...
```

```
...
$ where
My current location is
/home/it244gh

$ cd /

$ pwd
/

$ where
My current location is
/home/it244gh
```

Single Quotes Versus Double Quotes in Aliases

- The value of **PWD** was evaluated *when the alias was defined*
- If I now move to another directory, the value of **PWD** will be changed, but that *will not affect the alias*.
  - The alias got the value of **PWD**, at the time, when it was defined.
  - Therefore, the current value of **PWD** is irrelevant in terms of how the alias words.
- To define this alias *correctly*, we must use single quotes

```
$ alias where='echo My current location is $PWD '
```

Single Quotes Versus Double Quotes in Aliases

- Now, **PWD** will be evaluated when the alias is used

```
$ pwd
/home/it244gh

$ alias where= 'echo My
current location is
$PWD'

$ where
...
...
My current location is
/home/it244gh

$ cd /

$ pwd
/

$ where
My current location is
/
```

Single Quotes Versus Double Quotes in Aliases

- Here, we get the current value of **`PWD`** **,** at the time the alias is run
- When the shell comes across an alias, it simply substitutes the ***value*** of the alias for the ***name*** of the alias

Examples of Aliases

- I have defined a number of aliases to make my work easier
- One alias makes it easier for me to list the "invisible files" in a directory

```
$ alias la='ls -a'

$ la
.    .addressbook    .bash_profile  .cshrc  .login    .pinerc
..   .bash_history   .cache         it244   .msgsrc   .ssh
```

Examples of Aliases

- Another alias makes it easier to see the *most recently created* files and directories

```
$ alias ltr
alias ltr='ls -ltr'

$ ltr ~/bin/shell
total 4
-rwxr-xr-x 1 ghoffmn grad 107 Oct 13 09:42 border.sh
-rwxr-xr-x 1 ghoffmn grad 339 Oct 13 09:42 hw_setup.sh
-rwxr-xr-x 1 ghoffmn grad 306 Oct 14 00:18 hw_copy.sh
-rwxr-xr-x 1 ghoffmn grad 156 Nov  5 17:09 step.sh
```

Examples of Aliases

- That alias, **`ltr`** , *sorts* the directory listing, with the most recent entries at the bottom
- This make is easier to see the *most recently modified* files
- A user may create aliases, using a variable, to go to directories they visit often

```
$ lnwb=/home/ghoffmn/public_html/it244

$ alias lnwb='cd $lnwb'

$ pwd
/home/ghoffmn/bin

$ lnwb

$ pwd
/home/ghoffmn/public_html/it244
```

Examples of Aliases

- You can follow a standard procedure when defining such aliases
  - First, create a variable holding the absolute address of the directory
  - Next, create an alias to go to that directory, and let the name of the alias be the *same* as the variable
- There is no possibility of confusion since you must put a **$** in front of a *variable* to get its value, but you *don't* do this with an alias

Functions

- A **function** is a collection of shell commands that is given a name
- Functions can accept arguments from the command line using **positional parameters**
- A function can be run anywhere in the filesystem, since it has no pathname
- You simply *type the **name** of the function* to run it

Functions

- 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 its own process*
- For these reasons, functions run much faster than shell scripts

Functions

- Functions should be used sparingly because they take up memory
  - All the functions you define are loaded into the memory of your shell process
  - If you define *too many* functions, it will hurt the performance of the shell
- Functions are local to the shell in which they are defined, so they *do not* work in sub-shells

Functions

- Functions definitions have the following form

```
FUNCTION_NAME ()
{
        COMMAND
        COMMAND
        . . .
}
```

- Where **COMMAND** is anything you can enter at the command line.

Functions

- *Example:*

```
$ whoson ()
> {
>   date
>   finger | grep 'pts/'$1
> }

$ whoson 14
Sun Nov 10 20:18:33 EST 2013
thamerfa  Thamer AlTuwaiyan   pts/14       2  Nov
  10 18:44 (c-174-63-86-44.hsd1.ma.comcast.net)
```

Functions

- Once you type the final `}`, the definition is complete, and you get a command prompt back
- You can define a function on a single command line

```
$ echo3 () { echo $1; echo $1; echo $1;}

$ echo3 foo
foo
foo
foo
```

Functions

- But, you **must** use a semi-colon after each command. *Including the last command*
- For clarity, you can precede the function name with the **keyword** ***function***

```
$ function cheer ()
> {
>   echo Go $1'!'
> }

$ cheer 'Red Sox'
Go Red Sox!
```

Functions

- ***function*** is not a command

```
$ type function
function is a shell
   keyword
```

- So it is **optional** when defining a function
- You can pass *command line arguments* to a function using the *positional parameters*

```
$ print_args ()
> {
>   echo "arg1: $1"
>   echo "arg2: $2"
> }

$ print_args foo bar
arg1: foo
arg2: bar
```

```
$ type pu
pu is a function
pu ()
{
    echo;
    pushd $1 > /dev/null;
    ls --color=auto
}

$ pwd
/home/it244gh

$ pu ~ghoffmn

assignments_submitted  it114  mail     public_html  vp
bin                    it244  Mail     scans
course_files           it341  News     test
html                   it441  nsmail   test_taken

$ pwd
/home/ghoffmn
```

- To see a function's definition, you can use the ***type*** command
- Consider the function **pu** (*defined by Prof. Hoffman*), which can be used in place of ***cd***

Functions

- The function **pu** first prints a blank line, which makes things easier to read; then, it calls ***pushd***
- But, ***pushd*** prints the directory stack, which can be distracting, so the function sends this output to **/dev/null**
- Next, run ***ls*** to see the contents of the new directory
- **pu** is a good function name because it is shorter than "push"

Functions

- With **pu** , the directory stack remembers the last directory
- To return to this directory, there is **po** (*also by Prof. Hoffman*), which is short for "pop"
- But, **po** does not require an argument, so it can be an *alias*

```
$ alias po
alias po='popd > /dev/null; echo; ls'
```

- **po** uses ***popd*** to return to the previous directory, redirecting the printing of the directory stack to **/dev/null**

Functions

- Then, it prints a blank line and prints the contents of the directory
- Another example: Perhaps you would like to be able to both *create* and *change to* a new directory, at once.
- Here is a function, called **godir**, that will do this…

```
$ function godir ()
> {
>
>       mkdir $1
>       cd $1
>
> }

$ pwd
/home/ckelly

$ godir new_test_dir

$ pwd
/home/ckelly/new_test_dir
```

Functions

- To remove a function, use the ***unset*** command

```
$ cheer 'Red Sox'
Go Red Sox!

$ unset cheer

$ cheer
cheer: command not found
```

- Functions, like aliases, only work in the shell in which they are defined

Where to Define Variables, Aliases, and Functions

- Global variables are visible in all sub-shells
- Global variables should be defined in `.bash_profile` in your home directory
- That will make them available when you login, since the commands in `.bash_profile` are run *after* your password is accepted
- Aliases and functions cannot be made global, which means that if you define them in `.bash_profile`, they will *not* be available in interactive sub-shells.

Where to Define Variables, Aliases, and Functions

- But, since few people use interactive sub-shells, they should probably be put in **`.bash_profile`**, just to keep things simple.
- If you *need* your aliases and functions in interactive sub-shells, then you should define them instead in **`.bashrc`** and then add the following in your **`.bash_profile`**

```
source .bashrc
```

....or they won't be defined in your login shell