Miscellaneous Scripting Tools

- **_read_** Command
- Here Documents
- Using Braces **{ }** with Variables
- Array Variables
- Special Parameters
    - **$** - The PID
    - **!** - The PID of the Last Process Put into the Background
    - **?** The Exit Status
- Positional Parameters
    - **#** - The Number of Command Line Arguments
    - **0** - The Pathname of the Script
    - **1** - **n** - The Command Line Arguments
    - ***** and **@** - All the Command Line Arguments
    - **_shift_** - Promotes Command Line Arguments
    - **_set_** - Initialize Command Line Arguments

***read* Command**

- In order to get input from the user without using command line arguments, we need to use the ***read*** command
- 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 following the *read* command
- Let's look at an example…

read Command

```
$ cat read_1.sh
#! /bin/bash
#
# demonstrate use of the read command

echo -n "Please enter a word: "
read reply
echo You entered: $reply

$ ./read_1.sh
Please enter a word: foo
You entered: foo
```

***read* Command**

- When ***read*** takes in a value from the terminal, it grabs everything the user types until they hit Enter

```
$ ./read_1.sh
Please enter a word: foo bar bletch
You entered: foo bar bletch
```

- In the script above, we used ***echo*** to print a prompt for the user, but we can also use the **-p** option to have ***read*** issue the prompt…

read Command

```
$ cat read_2.sh
#! /bin/bash
#
# demonstrate use of the read command
# using the prompt option

read -p "Please enter a word: " reply
echo You entered: $reply

$ ./read_2.sh
Please enter a word: foo
You entered: foo
```

***read* Command**

- By default, the ***read*** command **will not** allow you to edit the text using the readline library commands

```
$ ./read_2.sh
Please enter a word: foooo^?^?^?
You entered: foooo
```

- Here we have hit the backspace key three times
- But, the ***read*** command normally ignores this; however, it can be *forced* to allow **command line editing**, if you use the **-e** option to ***read***

Here Documents

- Scripts can read data from a file like any Unix command
- But sometimes, you want the data to be contained *inside the script itself*
- You might want to do that to make a script easier to deploy since you would not have to distribute a second (data) file along with the script
- This can be accomplished with a **here document**
- A here document is a feature of Bash that allows a file to be included within a script, which the script can read for its input

Here Documents

- Here is an example…

```
$ cat here.sh
#! /bin/bash
#
# demonstrates how here documents work

read -p "Please enter a city: " city
grep $city << EOF
Boston Red Sox
New York Yankees
Toronto Blue Jays
Baltimore Orioles
Tampa Bay Rays
EOF
```

Here Documents

```
$ ./here.sh
Please enter a city: Boston
Boston Red Sox
```

- The script contains a list of all American League Eastern Division baseball teams, along with the cities in which they play
- The here document begins with the two less than symbols `<<` followed **immediately** by a string
- In the script above, we have chosen "EOF" as the string
- This string will serve to mark the beginning and end of the here document

Here Documents

- This string can be anything you like, as long as it contains no whitespace
- Bash will look for another occurrence of the same string on a line by itself and read that as the end of the here document
- **Nothing** must appear on the line after the first string, and the second appearance of the sting **must** be on a line **by itself**

Using Braces { } with Variables

- Normally, when we use the value of a variable, any text following the variable name is separated by **whitespace**

```
$ team=Bruins

$ echo Go $team
Go Bruins
```

- But, what if you needed to use the value of the variable as part of a larger string?
- If the following string starts with a period **.** or a slash **/** there is no problem
- Here is an example…

Using Braces { } with Variables

```
$ filename=test

$ echo Creating $filename.txt
Creating test.txt

$ dirname=test_dir

$ echo Creating $dirname/test.txt
Creating test_dir/test.txt
```

- But with any other text you run into difficulty

```
$ echo Creating $filename_1.txt
Creating .txt
```

- Bash did not see the variable `filename` next to the string "`_1.txt`"

Using Braces { } with Variables

- Instead, it saw a *new* variable, **filename_1** , which was not defined. Since this variable was not defined, it has no value
- In order 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 Creating ${filename}_1.txt
Creating test_1.txt
```

- The braces surround the variable name and the opening brace comes after the dollar sign **$**

Array Variables

- ***bash*** supports one-dimensional array variables
- An array can hold many individual values
- An array variable is defined as follows
  ```
  VARIABLE_NAME=(ELEMENT1 ELEMENT2 ... )
  ```
- For example
  ```
  $ cities=(Boston Chicago Philadelphia Cleveland)
  ```
- Notice that there are **no spaces** on either side of the equal sign **=** but spaces **are required** between individual array values

Array Variables

- To access the array values we must
  - Follow the name of the variable with square brackets
  - Have a number inside the square bracket indicating the desired value
  - Enclose the variable name and the square bracket inside braces
- Here is an example

```
$ echo ${cities[0]}
Boston

$ echo ${cities[1]}
Chicago
```

Array Variables

- Notice that the array elements are numbered starting with **`0`**
- If you don't use the square brackets, you will simply get the first value of the array

```
$ echo $cities
Boston
```

- If you don't use the braces, you will not get the results you expect

```
$ echo $cities[2]
Boston[2]
```

- Since I did not put braces around **`cities`**[2], ***bash*** simply appended the string "[2]" to the value of the first entry

Array Variables

- There are two symbols that can be used to get all the values in an array

```
$ echo ${cities[*]}
Boston Chicago Philadelphia Cleveland

$ echo ${cities[@]}
Boston Chicago Philadelphia Cleveland
```

- The asterisk * turns all the elements of the array into a **single string** with the values separated by spaces
- The at sign @ reproduces the original array. That is, it creates a new array with the same elements in the same order

Array Variables

- Each element of the array remains distinct
- We can see this if we use the ***declare*** built-in
- ***declare*** is normally used to set the attributes of a variable, and one of those attributes is *whether the variable is an array*
- You can also use ***declare*** to list every variable which is an array
- To do this run ***declare*** with a dash **-** followed by an attribute, but no argument

Array Variables

- So if we run ***declare*** with the **-a** option, it will display all array variables

```
$ c1=("${cities[*]}")

$ echo $c1
Boston Chicago Philadelphia Cleveland

$ c2=("${cities[@]}")

$ echo $c2
Boston

$ declare -a
...
```

Array Variables

```
declare -a c1='([0]="Boston Chicago Philadelphia
   Cleveland")'
declare -a c2='([0]="Boston" [1]="Chicago" [2]=
   [3]="Cleveland")'
declare -a cities='([0]= [1]="Chicago"
   [2]="Philadelphia" [3]="Cleveland")'
```

- The variable `c1` is an array with only one element
- That element is a string composed of all the values in the original array variable
- `c2` is an array variable with exactly the same entries as `cities` in the same order

Array Variables

- You can use the hash mark # to get the length of an array value.
- Place the # within the curly braces, immediately in front of the array name, followed by the index number, inside square brackets

```
$ echo ${cities[0]}
Boston
$ echo ${#cities[0]}
6
```

- We can use an array variable to present a list of arguments to a script…

Array Variables

```
$ cat print_args.sh
#! /bin/bash
#
# prints the arguments given on the command line

for arg
do
    echo $arg
done

$ ./print_args.sh ${cities[*]}
Boston
Chicago
Philadelphia
Cleveland
```

Array Variables

- You assign a *new* value to an element of an array the same way you assign a value to *an ordinary variable*

```
$ cities[3]=Akron

$ echo ${cities[@]}
Boston Chicago Philadelphia Akron
```

- Array variables will **not** be on the final

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

$ - PID of Current Shell Process

- The special parameter **$** contains the *process ID* (PID) of the current shell
- If you ***echo*** the **$** at the command line, you will get the process ID of your current shell

```
$ echo $$
6834

$ ps
  PID TTY          TIME CMD
 6834 pts/1    00:00:00 bash
 7024 pts/1    00:00:00 ps
```

- Notice that the value returned is the *same* as that of the shell

$ - PID of Current Shell Process

- Notice also that $ is the **name** of the parameter, so we had to put a $ in front of it to get its value
- The $ special parameter is very useful when creating temporary files
- When you create a temporary file you want to be sure the name is unique
- Otherwise, you might overwrite a temp file created by another process

$ - PID of Current Shell Process

- You can create a unique temp file using the special parameter like this

```
$ tmp=$$tmp

$ echo $tmp
6834tmp
```

! - The PID of the Last Process Put into the Background

- ! contains the process ID (PID) of the *last* process put into the background

```
$ sleep 60 &
[1] 7347

$ echo $!
7347
```

! - The PID of the Last Process Put into the Background

- You can use ! to kill a job that you just ran

```
$ ./bother.sh > /dev/null &
[1] 5274

$ jobs
[1]+  Running                 ./bother.sh > /dev/null &

$ echo $!
5274

$ kill $!
[1]+  Terminated              ./bother.sh > /dev/null

$ jobs

$
```

? - The Exit Status

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

```
$ pwd
/home/it244gh/it244/work

$ echo $?
0

$ ls asdfasd
ls: cannot access asdfasd: No such file or directory

$ echo $?
2
```

? - The Exit Status

- The *first* command succeeds and returns an exit status of **0**, while the *second* fails and returns a non-zero exit status
- You can set the exit code in a shell script by using the ***exit*** built-in

```
$ cat exit.sh
#! /bin/bash
#
# demonstrates the use of the exit command with a status code

exit 2

$ ./exit.sh

$ echo $?
2
```

- The exit status is also called the *condition code* or the *return code*

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

```
$ cat arg_count.sh
#!/bin/bash
#
# Prints the number of arguments sent to this script

echo This script received $# arguments

$ ./arg_count.sh foo bar bletch
This script received 3 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

```
$ cat command_name.sh
#!/bin/bash
#
# prints the pathname by which called this script

echo This sript was called using the pathname $0

$ ./command_name.sh
This sript was called using the pathname
./command_name.sh

$ /home/ghoffmn/examples_it244/command_name.sh
This sript was called using the pathname
/home/ghoffmn/examples_it244/command_name.sh
```

0 - The Pathname of 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

n - The Command Line Arguments

- The numbered positional parameters are used to give *command line arguments* to a script or to a function
- Here is an example…

```
$ cat print_positionals.sh
#!/bin/bash
#
# Prints the value of the first four positional arguments

echo 0: $0
echo 1: $1
echo 2: $2
echo 3: $3
```

n - The Command Line Arguments

```
$ ./print_positionals.sh foo bar bletch
0: ./print_positionals.sh
1: foo
2: bar
3: bletch
```

- If there is no corresponding command line argument, then the parameter will have no value

```
$ ./print_positionals.sh foo bar
0: ./print_positionals.sh
1: foo
2: bar
3:
```

- You cannot run ***test*** on a positional parameter if there is no corresponding command line argument

n - The Command Line Arguments

- If you do, you will get an error

```
$ cat greater_than_zero.sh
#! /bin/bash
#
# this script tests whether its first command line
# argument is greater than 0

if [ $1 -gt 0 ]
then
    echo $1 is greater than 0
else
    echo $1 is not greater than 0
fi
```

n - The Command Line Arguments

```
$ ./greater_than_zero.sh
./greater_than_zero.sh: line 5: [: -gt: unary operator
expected
is not greater than 0
```

- This is why you need to *check* for the correct number of arguments whenever your script takes arguments from the command line

@ - All the Command Line Arguments

- There are two special parameters that can be used to return **all** arguments from the command line
- They are * and @
- Both return all the arguments from the command line

```
$ cat special_param_test_1.sh
#! /bin/bash
#
# demonstrates some properties of the special
# parameters $* and $@

echo 'Here is $*: ' $*
echo 'Here is $@: ' $@
```

@ - All the Command Line Arguments

```
$ ./special_param_test_1.sh 1 2 3 4 5
Here is $*:  1 2 3 4 5
Here is $@:  1 2 3 4 5
```

- You can use them in ***for ... in*** loops

```
$ cat special_param_test_2.sh
#! /bin/bash
#
# demonstrates some properties of the special
# parameters $* and $@

echo 'The $* loop'
for arg in $*
do
  echo $arg
done
....
```

@ - All the Command Line Arguments

```
....

echo 'The $@ loop'
for arg in $@
do
   echo $arg
done
```

- So, why does ***bash*** give us two different parameters that *appear* to do the same thing?
- Because they are subtly different

```
$ ./special_param_test_2.sh 1 2 3 4 5
The $* loop
1
2
3
4
5
The $@ loop
1
2
3
4
5
```

@ - All the Command Line Arguments

- When you enclose `$*` within **double quotes** it's value is a single string
- That single string consists of *all* command line arguments, concatenated together with a space between them
- But, when you enclose `$@` within double quotes, its value is a *list* of separate strings – one for each command line argument
- Let's look at an example...

@ - All the Command Line Arguments

```
$ cat special_param_test_3.sh
#! /bin/bash
#
# demonstrates some properties of the special
# parameters $* and $@

echo 'The $* loop'
for arg in "$*"
do
  echo $arg
done

echo 'The $@ loop'
for arg in "$@"
do
  echo $arg
done
```

@ - All the Command Line Arguments

```
$ special_param_test_3.sh 1 2 3 4 5
The $* loop
1 2 3 4 5
The $@ loop
1
2
3
4
5
```

- The difference between `$*` and `$@` only appears when they are placed inside double quotes
- `$*` and `$@` will not be on the final

***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...
- Let's look at an example...

shift: Promotes Command Line Arguments

```
$ cat shift_1.sh
#! /bin/bash
#
# demonstrates the use of
# the shift command

echo '$1: ' $1
echo '$2: ' $2
echo '$3: ' $3
echo '$4: ' $4

echo
echo shifting arguments
shift
echo
```

```
echo '$1: ' $1
echo '$2: ' $2
echo '$3: ' $3
echo '$4: ' $4

$ ./shift_1.sh foo bar
bletch bling
$1:  foo
$2:  bar
$3:  bletch
$4:  bling

shifting arguments

$1:  bar
$2:  bletch
$3:  bling
$4
```

***shift*: Promotes Command Line Arguments**

- After ***shift*** is called, all arguments move *up* one position, and the first argument value is lost
- It is not possible to get the value of the first parameter after ***shift*** has been called
- If ***shift*** is called with a numeric argument, all arguments are moved up that number of positions
- Let's look at an example...

shift: Promotes Command Line Arguments

```
$ cat shift_2.sh
#! /bin/bash
#
# demonstrates the use of the
# shift command with an
# integer argument

echo '$1: ' $1
echo '$2: ' $2
echo '$3: ' $3
echo '$4: ' $4

echo
echo shifting arguments by 2
shift 2
echo
```

```
echo '$1: ' $1
echo '$2: ' $2
echo '$3: ' $3
echo '$4: ' $4

$ ./shift_2.sh foo bar bletch
bling
$1:  foo
$2:  bar
$3:  bletch
$4:  bling

shifting arguments by 2

$1:  bletch
$2:  bling
$3:
$4:
```

***shift*: Promotes Command Line Arguments**

- 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_3.sh
#! /bin/bash
#
# demonstrates the use of the shift in a while loop

while [ ! -z $1  ]
do
     echo 'The value of $1 is ' $1
     shift
done
```

shift: Promotes Command Line Arguments

```
$ ./shift_3.sh foo bar bletch bling blam
The value of $1 is  foo
The value of $1 is  bar
The value of $1 is  bletch
The value of $1 is  bling
The value of $1 is  blam
```

- ***shift*** keeps promoting arguments until there are no more left
- The **-z** option to ***test*** returns true if the length of what follows is zero
- But the not operator **!** makes the test true when the string is *greater than zero*

***set*: Initialize Command Line Arguments**

- The ***set*** command can create positional arguments from *within* a script
- Normally, the positional parameters take their value from the command line arguments
- If you call ***set*** and follow it with a list of values, then ***set*** will assign each of those values to a positional parameter – and any values from the command line are lost
- Each of the values following ***set*** are loaded into the corresponding positional parameter, starting with **1**

set: Initialize Command Line Arguments

```
$ cat set.sh
#! /bin/bash
#
# demonstrates using the set command to assign values
# to positional parameters

echo This script received $# arguments from the command line
echo '$1: ' $1
echo '$2: ' $2
echo '$3: ' $3
echo '$4: ' $4

echo
echo 'After  set bloo blah blim blak'
set bloo blah blim blak
echo '$1: ' $1
echo '$2: ' $2
echo '$3: ' $3
echo '$4: ' $4
```

***set*: Initialize Command Line Arguments**

```
$ ./set.sh foo doo ewe goh
This script received 4 arguments from the command line
$1:  foo
$2:  doo
$3:  ewe
$4:  goh

After  set bloo blah blim blak
$1:  bloo
$2:  blah
$3:  blim
$4:  blak
```

- This script was run with one set of positional parameters
- But, after I ran ***set***, the positional parameters had different values

***set*: Initialize Command Line Arguments**

- ***set*** is a built-in command
- ***set*** has a completely different behavior when called with the **-o** or **+o** options
- With those options, the ***set*** command sets or unsets a shell option that alters the way ***bash*** behaves
- ***set*** will not be on the final