Bash (Bourne Again Shell)

- The standard shell of Linux
  - major shell commands include:
  - alias, bg, builtin, case..in..esac, cd, declare, dirs, env, export, fg, for..do..done, function, history, if..then..elif..then..else..fi, jobs, kill, local, popd, pushd, read, readonly, return, select..do..done, set, source, trap, unalias, unset, until..do..done, while..do..done
  - bash is installed at /bin/bash, /bin/sh is link to /bin/bash

Bash (Bourne Again Shell)

- Features of Bash new or different from the previous discussion
  - variable manipulation
  - command-line processing, aliases, and history
  - arithmetic, conditional expressions, control structures
  - directory stack
  - job control
  - shell functions

Bash Startup

- Bash starts just like any program
  - as login shell
    - executes commands in *.bash_profile*
  - as normal invocation
    - executes commands in file *.bashrc*
  - may want to call *.bashrc* from within *.bash_profile*

Variables

- creation and use of shell variables are for:
  - Value assignment and access
  - Defining and using lists of values
  - Testing a value or for existence of a variable
  - Reading or writing a variable's value

Variables

- Simple variable -- creation/assignment
  - {value = name}
  - e.g. gameswon=12
- built-in command *set* displays all variables set in shell

Accessing simple variables

- *$name* Replaced by the value of name.

- *${name}* Replaced by the value of name. This form is useful if the expression is immediately followed by an alphanumeric that would otherwise be interpreted as part of the variable name.

$ verb=sing ...assign a variable.
$ echo I like $verbing ...there's no variable "verbing".
I like
$ echo I like ${verb}ing ...now it works.
I like singing
$ _

- *${name-word}* Replaced by the value of name if set, and word otherwise.

-bash-3.2$ ddd=${ddd- `date`}

-bash-3.2$ echo $ddd

Mon Sep 27 09:31:03 PDT 2010

-bash-3.2$

-bash-3.2$ echo ${asdfasdf-test} (there is no asdfasdf)

test

- *${name+word}* Replaced by word if name is set, and nothing otherwise.

-bash-3.2$ flag=1

-bash-3.2$ echo ${flag+'flag is set'}

flag is set

-bash-3.2$ echo ${flag2+ 'flag2 is set'}

–

-bash-3.2$

- *${name=word}* Assigns word to the variable name if name is not already set. Then it is replaced by the value of name.

-bash-3.2$ echo x = ${x=10}
x = 10
-bash-3.2$ echo $x
10

- *${name?word}* Replaced by name if name is set. If name is not set, word is displayed to the standard error channel and the shell is exited. If word is omitted, then a standard error message is displayed instead.

-bash-3.2$ total=10

-bash-3.2$ value=${total?'total not set'}

-bash-3.2$ echo $value

10

-bash-3.2$ value=${grandTotal?'grand total not set'}

-bash: grandTotal: grand total not set

-bash-3.2$

- Another example for *${name?word}*

```
$ cat script.sh                  ...look at the script.
value=${grandTotal?'grand total is not set'}
echo done                        # this line is never executed.
$ script.sh                      ...run the script.
script.sh: grandTotal: grand total is not set
$ _
```

- ${#name} Replaced by the length of the value of name.
- ${#name[*] } Replaced by the number of elements in the array name.
- ${name:+word } Work like their counterparts that do not contain a :, except that name must be set and non-null instead of just set.
  - ${name:=word }
  - ${name:?word }
  - ${name:+word }

- *${name#pattern}* Removes a leading pattern from name. The expression is replaced by the value of name if name doesn't begin with pattern, and with the remaining suffix if it does. This form removes the smallest matching pattern.

  ${name##pattern} This form removes the largest matching pattern

  -bash-3.2$ echo $PWD

  /home/krings/CS270

  -bash-3.2$ echo $HOME

  /home/krings

  -bash-3.2$ echo ${PWD#$HOME/}

  CS270

  -bash-3.2$ echo ${PWD#$HOME}

  /CS270

${name%pattern} Removes a trailing pattern from name. The expression is replaced by the value of name if name doesn't end with pattern, and with the remaining prefix if it does. This form removes the smallest matching pattern.

${name%%pattern} This form removes the largest matching pattern.

-bash-3.2$ testfile=menu.sh

-bash-3.2$ echo ${testfile%.sh}.bak

menu.bak

-bash-3.2$

List Variables

- List variables (arrays) are created with *declare*. Simply using the variable in an array format will also work.

- Shell command: ***declare*** *[-ax] [listname]*
  - If the named variable does not already exist, it is created.
  - If an array name is not specified when **-a** is used, declare will display all currently defined arrays and their values.
  - If the **-x** option is used, the variable is exported to subshells. *declare* writes its output in a format that can be used again as input commands. This is useful when you want to create a script that sets variables as they are set in your current environment.

List Variables

- example for *declare*

```
$ declare -a teamnames
$ teamnames[0]="Dallas Cowboys"
$ teamnames[1]="Washington Redskins"
$ teamnames[2]="New York Giants"
```

  - if you omit the declare command, the other lines will still work as expected.

Accessing List Variables

- When accessing array values, you can always put braces around the variable name to explicitly distinguish it from other text that might be around it.

Figure 6-6. Accessing the value(s) of a list variable.

${*name*[*index*]}	Access the *index*th element of the array $*name*.
${*name*[*]} or ${*name*[@]}	Access all elements of the array $*name*.
${#*name*[*]} or ${#*name*[@]}	Access the number of elements in the array $*name*.

```
$ echo "There are ${#teamnames[*]} teams in the NFL"
There are 32 teams in the NFL
$ echo "They are: ${teamnames[*]}"
...
```

Building Lists

- You can build an array in one of two ways.
  - If you know how many elements you will need, you can use the declare built-in command to define the space and assign the values into specific locations in the list.
  - If you don't know, or don't care, how many elements will be in the list, you can simply list them and they will be added in the order you specify.

Building Lists: example

```
$ declare -a teamnames
$ teamnames[0]="Dallas Cowboys"
$ teamnames[1]="Washington Redskins"
$ teamnames[2]="New York Giants"
  ...
$ teamnames[31]="Houston Texans"
```

- This can also be done in a single (long) command:

```
$ declare -a teamnames
$ teamnames=([0]="Dallas Cowboys" \
             [1]="Washington Redskins" \
  ...
             [31]="Houston Texans")
```

- or simply type

```
$ teamnames = ("Dallas Cowboys" "Washington Redskins" \
               "New York Giants" "New York Jets" \
  ...
                 "Houston Texans")
```

Building Lists: example

- How many elements are counted in the array?
  - As many as are assigned

```
? mylist[0]=27
? mylist[5]=30
? echo ${#mylist[*]}      ...number of elements in mylist[]
2
? declare -a              ...display defined element values
declare -a mylist='([0]="27" [5]="30")'
? _
```

  - it shows 2 and not 6 elements

Destroying Lists

- List variables are deallocated by using built-in command *unset*

- Shell command: **unset** *name*

unset *name[index]*

  - deallocates specific variable or element in list variable

```
? unset teamnames[17]
```

Reading a Variable from stdin

- Shell Command: **read** *{ variable }+*
  - read one line from standard input and then assigns successive words from the line to the specified variables.
  - Any words that are left over are assigned to the last-named variable.

```
$ cat script.sh                          ...list the script.
echo "Please enter your name: \c"
read name                                # read just one variable.
echo your name is $name                  # display the variable.
$ bash script.sh                         ...run the script.
Please enter your name: Joe Blow
your name is Joe Blow                    ...whole line was read.
$ _
```

Reading a Variable from stdin

- Reading multiple variables

```
-bash-3.2$ cat script.sh
echo "Enter your name: "
read firstname lastname
echo your first name is $firstname
echo your last name is $lastname

-bash-3.2$ ./script.sh
Enter your name:
Joe Blow
your first name is Joe
your last name is Blow
-bash-3.2$

-bash-3.2$ ./script.sh
Enter your name:
Jane
your first name is Jane
your last name is                    ... note that last name was not entered
-bash-3.2$
```

Exporting Variables

- In a shell variables are local to that shell, i.e., they are not passed to subshells. Command *export* allows you to export a variable to the environment
  - Shell Command: **export** *{ variable }+*
    - marks the specified variables for export to the environment. If no variables are specified, a list of all the variables marked for export during the shell session is displayed.
  - Utility: **env** *{ variable=value }* [command]*
    - env assigns values to specified environment variables, and then executes an optional command using the new environment. If variables or command are not specified, a list of the current environment is displayed.

Exporting Variables

```
$ export                    ...list my current exports.
export TERM                 ...set in my ".profile" startup file.

$ DATABASE=/dbase/db        ...create a local variable.
$ export DATABASE           ...mark it for export.
$ export                    ...note that it's been added.
export DATABASE
export TERM

$ env                       ...list the environment.
DATABASE=/dbase/db
HOME=/home/strammsack
LOGNAME=strammsack
PATH=:/bin:/usr/bin
SHELL=/bin/bash
TERM=xterm
USER=glass

$ bash                      ...create a subshell.
$ echo $DATABASE            ...a copy was inherited.
/dbase/db
$ ^D                        ...terminate subshell.
$ _
```

Exporting Variables

- The bash shell command *set* tells the shell that all shell variables be exported to any subshells created
- Shell command: **set** *-o allexport*
  - Tell the shell to export all variables to subshells.

Read-Only Variables

- To protect a variable against modification use:
- Shell Command: **readonly** *{ variable }**
  - makes the specified variables read-only, protecting them against future modification.
  - If no variables are specified, a list of the current read-only variables is displayed.
  - Copies of exported variables do not inherit their read-only status.

Predefined Variables

- figure 6-13

Name	Value
$-	The current shell options assigned from the command line or by the built-in set commanddiscussed later.
$$	The process ID of this shell.
$!	The process ID of the last background command.
$#	The number of positional parameters.
$?	The exit value of the last command.
$@	An individually quoted list of all the positional parameters.
$_	The last parameter of the previous command.
$BASH	The full pathname of the Bash executable.
$BASH_ENV	Location of Bash's startup file (default is ~/.bashrc).
$BASH_VERSINFO	A read-only array of version information.
$BASH_VERSION	Version string.
$DIRSTACK	Array defining the directory stack (discussed later).
$ENV	If this variable is not set, the shell searches the user's home directory for the ".profile" startup file when a new login shell is created. If this variable is set, then every new shell invocation runs the script specified by ENV.
$EUID	Read-only value of effective user ID of user running Bash.

$HISTFILE	Location of file containing shell history (default ~/.bash_history).
$HISTFILESIZE	Maximum number of lines allowed in history file (default is 500).
$HISTSIZE	Maximum number of commands in history (default is 500).
$HOSTNAME	Hostname of machine where Bash is running.
$HOSTTYPE	Type of host where Bash is running.
$IFS	When the shell tokenizes a command line prior to its execution, it uses the characters in this variable as delimiters. IFS usually contains a space, a tab, and a newline character.
$LINES	Used by *select* to determine how to display the selections.
$MAILCHECK	How often (seconds) to check for new mail.
$OLDPWD	The previous working directory of the shell.
$OSTYPE	Operating system of machine where Bash is running.
$PPID	The process ID number of the shell's parent.
$PPID	Read-only process ID of the parent process of Bash.

$PPID	The process ID number of the shell's parent.
$PPID	Read-only process ID of the parent process of Bash.
$PS1	This contains the value of the command-line prompt, and is $ by default. To change the command-line prompt, simply set PS1 to a new value.
$PS2	This contains the value of the secondary command-line prompt that is displayed when more input is required by the shell, and is > by default. To change the prompt, set PS2 to a new value.
$PS3	The prompt used by the *select* command, #? by default.
$PWD	The current working directory of the shell.
$RANDOM	A random integer.
$REPLY	Set by a *select* command.
$SHLVL	Level of shell (incremented once each time a Bash process is started, it shows how deeply the shell is nested).
$UID	Read-only value of user ID of user running Bash.