Korn Shell Script Writing

Science & Technology Support Group
High Performance Computing
OSC (Ohio Supercomputer Center)
1224 Kinnear Road
Columbus, OH 43212

OSC
Innovations in computing, networking, and education

Basic Shell Programming

- Creating a Shell Script
- Executing a Shell Script
- Functions
- Precedence of Commands
- eval
- Shell Variables
- Assigning Variable Names
- Null Variables
- Special Shell Variables
- Parameter Substitution
- Special Pattern Matching Features
- Command Substitution
- Exercises

Creating a Shell Script

- **A shell script is an executable file which is executed by the shell line-by-line. It can contain the following:**
  - UNIX commands
  - shell programming statements
  - comments

- **Create using editor of choice**

- **Can include a #! construct in first line of script to override login shell**
  - `#!/bin/sh` uses Bourne shell to execute script
  - `#!/bin/csh` uses C shell to execute script
  - etc...

Executing a Shell Script

There are 3 ways to execute a shell script:

1."dot" method

```
$ . scriptname
```

2."just the name" method

```
$ scriptname
```

3.in the background

```
$ scriptname &
```

Executing a Shell Script (cont.)

- **Method 1 runs the command as if you typed them in on the command line**

- **Note that methods 2 and 3 require:**
  - execute permission for scriptname

    ```
    chmod +x scriptname
    ```
  - current directory (.) must be in PATH or else must use `./scriptname`

Executing a Shell Script (cont.)

cat list

```
# a simple little shell script
   print "A listing of $PWD \n" > list.out
   ls -l >> list.out
```

- **For methods 2 and 3 the shell runs another copy of itself as a subprocess. Pictorially, the difference between Methods 1, 2, and 3 is:**

Functions

- **Improves shell's programmability**
  - already in memory (unless autoloaded)
  - modular programing

- **Syntax:**

```
function functname {
        shell commands
   }
```

or,

```
functname ()   {
       shell commands
}
```

Functions (cont.)

- **Delete a function definition with**

```
$ unset -f functname
```

- **Display all defined functions:**

```
$ functions
```

Functions (cont.)

- **Two important differences between functions and shell scripts run by name (Method 2):**
  - functions do not run in a subprocess; behave more like a script run by Method 1
  - functions have precedence over scripts

- **Where to put them?**
  - enter on command line (no longer available if you log out and log back in)
  - `.profile` (only available in login shell)
  - `$ENV file` (available to all child shells)
  - in `$FPATH` directory (Korn shell will use autoload feature)

- **Function names should begin with a letter or underscore**
- **The closing brace (}) must appear on a line by itself**

Functions (cont.)

Example: Run a program and view the output

```
$ myrun ()
> {
>cd $WORKDIR
> a.out < data.in
> more data.out
> /bin/rm data.in
> mv data.out data.in
> }
```

Precedence of Commands

Now that we have discussed the various sources of commands, let us show their order of precedence:

1. Keywords such as function, if and for (see later)
2. Aliases
3. Shell built-ins
4. Functions
5. Scripts and executable programs, for which the shell searches in the directories listed in the `PATH` environment variable.

eval

- **You can think of quoting as a way of getting the shell to skip some of the steps of command-line processing.**

- **Consider:**

```
$ listpage="ls | more"
    $ $listpage
    ls: |: No such file or directory
    ls: more: No such file or directory
```

The shell evaluates variables after it has looked for metacharacters like |

eval

- **The command**

```
$ eval $listpage
```

  works because the line is "rescanned" by the shell.

- **`eval` is an advanced command that requires some cleverness to be used effectively**

Shell Variables

- **The Korn shell includes the following types of variables:**
  - User-defined variables
  - Special shell variables

- **User-defined variables can be declared, read and changed from the command line or from within a shell script.**

- **A variable name can consist of the following:**
  - letters
  - digits
  - underscore character
  - first character of a variable name must be a letter or an underscore character

A variable can be made read-only using the `readonly` command: `$ readonly variable_name`

Assigning Variable Names

= Operator:

Enter the name that you have chosen for the variable followed by an equal sign and then

the value that you want to store in the variable.

```
$ my_card=ace
$ print $my_card
ace
$ my_name="Garth of Izar"
$ print $my_name
Garth of Izar
```

OSC
Innovations in computing, networking, and education

Assigning Variable Names

`read` command:

Reads a line from standard input and stores the value(s) entered in the variable name(s) following the `read` command.

```
$ read fname lname
Bela Okmyx
$ print $fname
Bela
$ print $fname $lname
Bela Okmyx
```

Null Variables

A variable can be set to a null value, even if previously assigned, using any of the following methods. There should be no spaces preceding or following the equal sign.

```
$ name=

$ name=''

$ name=""

$ unset varname
```

Null Variables (cont.)

All variables that don't exist are assumed null unless set -o nounset is used. Then the shell will indicate an error when an undefined variable is encountered

```
$ unset name
$ set -o nounset
$ print $name
ksh: name: parameter not set
$ set +o nounset
```

Special Shell Variables and Positional Parameters

- **Other variables defined by the shell for the user are special shell variables and "positional parameters."**

- **These variables are set automatically by the shell. Their values cannot be changed, but they may be referenced.**

- **The special shell variables are as follows:**

```
$#
$-
$?
$$
$!
$0
$*
$@
```

OSC
Innovations in computing, networking, and education

Special Shell Variables and Positional Parameters

- **The variable $# contains the number of arguments typed on the command line.**

```
$ cat numargs
```

print The number of arguments is $#

```
$ numargs
```

The number of arguments is 0

```
$ numargs 1 2 3 4 5
```

The number of arguments is 5

```
$ numargs "Hello World"
```

The number of arguments is 1

Special Shell Variables and Positional Parameters

- **The variable `$-` contains the shell flags (options) of the current shell.**

```
$ print $-
isxumh
$ set +u
isxmh
```

Special Shell Variables and Positional Parameters

- **The variable $? contains the exit status of the last command.**

```
$ ls
  file1     data     account.txt
  $ rm file1
  $ print $?
  0
  $ rm dataa
  dataa: No such file or directory
  $ print $?
  2
```

Special Shell Variables and Positional Parameters

- **The variable $$ contains the process ID of the current shell process.**

```
$ cat pid.test
print $$
$ print $$
454
$ pid.test
846
$ pid.test &
847
```

Special Shell Variables and Positional Parameters

- The variable $! contains the process ID number of the last command sent to the background.

```
$ compress hugefile.tar &
[1]     8834
$ kill -9 $!
```

- The variable $0 contains the name of the command (process) currently being executed.

```
$ cat cmd_name
```

print The name of this script is $0

```
$ cmd_name
```

The name of this script is **cmd_name**

```
$ mv cmd_name new_name
$ new_name
```

The name of this script is **new_name**

OSC
Innovations in computing, networking, and education

Special Shell Variables and Positional Parameters

- **The variable `$*` contains the string of all arguments (positional parameters) on the command line.**

```
$ cat args
```

print The arguments are: `$*`

```
$ args bob dave
```

The arguments are: `bob dave`

Special Shell Variables and Positional Parameters

- The variable **$@** is the same as **$*** except when enclosed in double quotes. Then, each argument contained in **$@** is double quoted.

```
$ cat args
print The arguments are: "$@"
```

```
$ args bob dave
The arguments are: bob dave
```

```
$ args "bob     dave"
The arguments are: bob       dave
```

Special Shell Variables and Positional Parameters

- **Positional parameters refer to the individual arguments on the command line. The positional parameters available are referenced as follows:**

```
$1 $2 $3 $4 $5 $6 $7 $8 $9
```

- **The parameter $1 contains the first argument, $2 the second argument, and so on.**

```
$ cat parms
print Arg 1 is: $1
print Arg 2 is: $2
print Arg 3 is: $3
print Arg 4 is: $4
print Arg 5 is: $5
```

Special Shell Variables and Positional Parameters

```
$ parms Space, the final frontier
```

Arg 1 is: Space,

Arg 2 is: the

Arg 3 is: final

Arg 4 is: frontier

Arg 5 is:

```
$ parms "Space, the final frontier"
```

Arg 1 is: Space, the final frontier

Special Shell Variables and Positional Parameters

Use the set command to change positional parameters. It replaces existing positional parameters with new values.

```
$ cat newpos
 print starting args are $*
 print number of args is $#
 print arg 1 is $1
 print arg 2 is $2
 set 3 4
 print new args are $*
 print number of args is $#
 print arg 1 is $1
 print arg 2 is $2

 $ newpos 1 2
 starting args are 1 2
 number of args is 2
 arg 1 is 1
 arg 2 is 2
 new args are 3 4
 number of args is 2
 arg 1 is 3
 arg 2 is 4
```

OSC
Innovations in computing, networking, and education

Special Shell Variables and Positional Parameters

- **What if there are more than 9 arguments?**

```
$ cat ten_args
print arg 10 is $10

print arg 10 is ${10}



$ ten_args a b c d e f g h i j
arg 10 is a0
arg 10 is j
```

- **Use the shift command to perform a shift of the positional parameters n positions to the left.**
  (Default n=1).

Special Shell Variables and Positional Parameters

- **The variables `$#`, `$*` and `$@` also change with the shift command.**

- **Once a shift is performed, the first parameter is discarded. The `$#` variable is decremented, and the `$*` and `$@` variables are updated.**

Special Shell Variables and Positional Parameters

Example of shift:

```
$ cat shift_it
 print $#: $0 $*
 shift
 print $#: $0 $*
 shift
 print $#: $0 $*
 shift
 print $#: $0 $*

 $ shift_it 1 2 3 4 5 6 7 8 9 0 a b
 12: shift_it 1 2 3 4 5 6 7 8 9 0 a b
 11: shift_it 2 3 4 5 6 7 8 9 0 a b
 10: shift_it 3 4 5 6 7 8 9 0 a b
 9: shift_it 4 5 6 7 8 9 0 a b

 $ cat ten_args
 arg1=$1
 shift
 print $arg1 $*

 $ ten_args 1 2 3 4 5 6 7 8 9 10
 1 2 3 4 5 6 7 8 9 10
```

OSC
Innovations in computing, networking, and education

Parameter Substitution

- **Notes on Variable Syntax**
  - Syntax `$name` is not quite accurate
  - `${name}` is more general

```
$ file=new;cp $file ${file}copy
```

  - Curly braces can be omitted as long as name is followed by a character that isn't a letter, digit, or underscore
  - This syntax allows for parameter substitution: replacement of the value of a variable with that of another based on specific conditions and events.

- **Parameter substitution formats:**

```
${parameter:-value}
${parameter:=value}
${parameter:+value}
${parameter:?value}
```

Parameter Substitution (cont.)

`${parameter:-value}`

- If parameter exists and isn't null, return its value; otherwise return value
- Purpose: returning a default value if the variable is undefined.
- Examples:

```
$ count=""
$ print You have ${count:-5} cards
You have 5 cards
$ print $count

$ FOO=/usr/local/bin/xx
$ print edit file ${FOO:-$HOME/xx}
edit file /usr/local/bin/xx
$ print $FOO
/usr/local/bin/xx
```

Parameter Substitution (cont.)

`${parameter:=value}`

- If parameter exists and isn't null, return its value; otherwise set it to value and then return its value
- Purpose: Setting a variable to a default value if it is undefined.
- Examples:

```
$ count=""
$ print You have ${count:=5} cards
You have 5 cards
$ print $count
5
$ FOO=/usr/local/bin/xx
$ print edit file ${FOO:=$HOME/xx}
edit file /usr/local/bin/xx
$ print $FOO
/usr/local/bin/xx
```

Parameter Substitution (cont.)

`${parameter:+value}`

- **If parameter exists and isn't null, return value; otherwise return null.**
- **Purpose: Testing for the existence of a variable**
- **Examples:**

```
$ trace=y
$ print ${trace:+"Trace mode on"}
Trace mode on
$ trace=""
$ print ${trace:+"Trace mode on"}
```

OSC
Innovations in computing, networking, and education

Parameter Substitution (cont.)

`${parameter:?message}`

- If parameter exists and isn't null, return its value; otherwise print parameter: followed by message and abort the current command or script. Omitting message produces the default message parameter null or not set.
- Purpose: Catching errors that result from variables being undefined.
- Examples:

```
$ count=""
$ print card '#'${count:?"deal!"}
ksh: count: deal!
$ count=5
$ print card '#'${count:?"deal!"}
card #5
```

NOTE: In all forms, the colon (:) is optional. If omitted, the operator tests for existence only, i.e., change "exists and isn't null" to "exists" in each definition.

Special Pattern Matching Features

- We have seen the use of the wildcards *,? and []

- Korn shell adds to their capabilities with special pattern matching operators:

Pattern	Matches
x	x
*(x)	Null string, x, xx, xxx, ...
+(x)	x, xx, xxx, ...
?(x)	Null string, x
!(x)	Any string except x
@(x|y|z|...)	x or y or z or ...

Special Pattern Matching Features

- **Example: list all files ending in .ps or .eps**

```
$ ls *.?(e)ps
```

- **Example: list all files not ending in .c, .h or not named `README` or `Makefile`**

```
$ ls !(*.c|*.h|README|Makefile)
```

Command Substitution

- **Two ways seen thus far for getting values into variables:**
  - assignment statements
  - command-line arguments (positional parameters)

- **Command substitution is a third way:**

  `$(UNIX command)` or `` `UNIX command` `` (archaic)

- **Examples:**

```
$ DAY=$(date +%A)
$ FILE_LIST=$(ls)
$ LOGGED_ON=$(who|cut -f1 -d' ')
$ FILE=$(< filename)
```

Also useful interactively,

`$ print $(who|wc -l)` users logged in

Exercises

1. Write a script called `lsc` which executes the command `ls -C`. To execute this command you must give the full path name to your `lsc` script. Make the `lsc` shell script executable and run it.

2. Write a script called `b` which changes your current directory to `/bin`, displays the current directory, and then executes the `lsc` script created above. Make the `b` script executable and run it. What is your current directory when the `b` script is finished executing? Why?

3. Write a script called `ex_on`, which turn on execute permission on the first argument on the command line.

4. Modify ex_on to turn on execute permission on all arguments on the command line.

5. Write a script called `12` that prints the twelfth argument on the command line.

Korn Shell Flow Control

- Flow Control
- Exit Status
- Logical Command Grouping
- `if / else`
- Condition Tests
- String Comparison Operators
- File Attribute Checking
- Integer Conditionals
- Examples
- `for`
- `case`
- `select`
- `while` & `until`
- Command-line Options
- Integer Variables & Arithmetic
- Exercises

OSC
Innovations in computing, networking, and education

Flow Control

- **Flow control gives a programmer the power to specify that only certain portions of a program run, or that certain portions run repeatedly.**

- **Korn shell supports the following flow control constructs:**

`if/else`	**Execute a list of statements if a certain condition is/is not true**
`for`	**Execute a list of statements a fixed number of times**
`while`	**Execute a list of statements repeatedly while a certain condition holds true**
`until`	**Execute a list of statements repeatedly until a certain condition holds true**
`case`	**Execute one of several lists of statements depending on the value of a variable**
`select`	**Allow the user to select one of a list of possibilities from a menu**

Exit Status

- At the end of its execution, every UNIX command returns a status number to the process that invoked it.
- Indicates whether or not the command ran successfully.
- An exit status of zero is used to indicate successful completion. A nonzero exit status indicates that the program failed. Some exceptions exist (diff).
- The shell sets the $? variable to the exit status of the last foreground command that was executed.

Exit Status

- The constructs `if`, `while`, `until` and the logical `AND` (`&&`) and `OR` (`||`) operators use exit status to make logical decisions:

  0 is a logical "true" (success)

  1 is a logical "false" (failure)

- There are built-in `true` and `false` commands which you can use.

Exit Status (cont.)

A shell, like any other process, sets an exit status when it finishes executing. Shell scripts will finish in one of the following ways:

`Abort` - If the script aborts due to an internal error, the exit status is that of the last command (the one that aborted the script).

`End` - If the script runs to completion, the exit status is that of the last command in the script

`Exit` - If the script encounters and exit or return command, the exit status is that set by those commands.

Logical Command Grouping

- Two special symbols enable you to execute a command based on whether or not the preceeding command was successful:

  `&&` is a logical AND `||` is a logical OR

- Usage:

  `statement1 && statement2`

  `statement2` will execute only if `statement1` was successful (returned an exit status of 0)

  `statement1 || statement2`

  `statement2` will execute only if `statement1` was not successful (returned a nonzero exit status).

Logical Command Grouping (cont.)

Example:

```
$ cat on
# a simple shell script to check
# if certain users are logged on
who | grep $1 > /dev/null \
&& print $1 logged on \
|| print $1 not logged on
$ on jason
jason logged on
$ on phantom
phantom not logged on
```

The logical constructs `&&`/`||` are commonly used with `if/else` constructs

if / else

- **Simplest type of flow control construct is the conditional embodied in Korn shell's `if` statement.**

- **Syntax:**

```
if condition
   then
            statements
   [elif condition            <--  can use multiple elif clauses
      then statements . . .]
   [else
            statements]
   fi                         <--  must be by itself on the final
                                   line of the construct
```

if / else (cont.)

- The `if` statement uses an exit status to determine whether or not to execute the commands.
- Statements are executed only if the given condition is true.
- If one or more `elifs` are used, the last else clause is an "if all else fails" part.

Condition Tests

- **The if construct can only test exit status but that doesn't limit you to checking only whether commands ran properly or not.**

- **Using the `[[ ]]` construct, many different attributes can be tested:**
  - pattern matching on strings
  - file attributes
  - arithmetic conditionals

- **`[[ condition ]]` just returns an exit status that tells whether condition is true or not (fits within if construct's syntax of if statements).**

- **`[[ ]]` surround expressions that include various types of operators.**

OSC
Innovations in computing, networking, and education

Condition Tests (cont.)

- Conditional expressions inside `[[ ]]` can be combined using the logical `&&` and `||` operators.

- Can also combine shell commands with conditional expressions using `&&` and `||`.

Condition Tests (cont.)

String comparison operators:

Operator	True if ...
`str = pat`	`str` matches `pat`.
`str != pat`	`str` does not match `pat`.
`str1 < str2`	`str1` is less than `str2`.*
`str1 > str2`	`str1` is greater than `str2`.*
`-n str`	`str` is not null (has length greater than 0).
`-z str`	`str` is null (has length 0).

*based on the ASCII value of their characters

`str` refers to an expression with a string value, and `pat` refers to a pattern that can contain wildcards.

Condition Tests (cont.)

File attribute checking:

Operator	True if ...
`-a file`	file exists
`-d file`	file is a directory
`-f file`	file is a regular file
`-m file`	file is migrated (UNICOS extension)
`-r file`	there is read permission on file
`-s file`	file is not empty
`-w file`	you have write permission on file
`-x file`	you have execute permission on file, or directory search permission if it is a directory
`-O file`	you own file
`-G file`	your group id is same as file's
`file1 -nt file2`	file1 is newer than file2
`file1 -ot file2`	file1 is older than file2

There are 21 such operators in total.

Condition Tests (cont.)

Integer Conditionals

- Necessary if you want to combine integer tests with other types of tests within the same conditional expression
- There is a separate, more efficient syntax for conditional expressions involving integers only

Test	Comparison
-lt	less than
-le	less than or equal
-eq	equal
-ge	greater than or equal
-gt	greater than
-ne	not equal

OSC
Innovations in computing, networking, and education

Condition Tests (cont.)

Examples

```
$ [[ -z "" ]]
$ print $?
0
$ [[ -z foo ]]
$ print $?
1

$ who | grep joe && write joe || \
print joe not logged in

$ if [[ 6 > 57 ]] ; then print huh?;fi
huh?
```

Condition Tests (cont.)

The following script sets user execute permission on an ordinary, non-migrated file:

```
if [[ -f $1 && ! -m $1 ]]; then
        chmod u+x $1
fi
```

Condition Tests (cont.)

Examples

- **The following script removes the first file if it's older than the second file and the variable KEY is non-null:**

```
if [[ $1 -ot $2 && -n $KEY ]]
   then
       /bin/rm $1
   fi
```

Condition Tests (cont.)

The following script compares two files and if no differences are found, removes the second file:

```
USAGE="Usage:\t$(basename $0) file1 file2"
      if [[ $# -ne 2 ]]; then
        print -u2 ${USAGE}\\n
        exit 1
      fi

      diff $1 $2 > /dev/null
      if [[ $? -eq 0 ]]; then
        /bin/rm $2
        print $2 removed, $1 kept
      else
        print $1 and $2 differ
      fi
```

for

- Previous tests only allow reporting on single files since tests like `-f` and `-x` only take single arguments
- The `for` loop allows you to call a section of code a fixed number of times, e.g., once for each file given on the command line.
- During each time through the iteration, the loop variable is set to a different value.

for

- Previous tests only allow reporting on single files since tests like `-f` and `-x` only take single arguments
- The `for` loop allows you to call a section of code a fixed number of times, e.g., once for each file given on the command line.
- During each time through the iteration, the loop variable is set to a different value.

for (cont.)

Syntax:

```
for name [in list]
do
        statements that can use $name
done
```

- **`name` is a variable which can be called anything (commonly called i)**
- **`list` is a list of names (defaults to "$@")**
- **`name` is set to each name in `list`, in order, for each iteration; the number of iterations equals the number of names in `list`.**

for (cont.)

Examples

```
$ cat simple
   for i in This is a test
   do
      print $i
   done
   $ simple
   This
   is
   a
   test
```

<-- Omitting the in This is a test clause and instead running the script as:
$ simple This is a test
would yield the same output.

for (cont.)

Examples

Check to see who is logged on the machines listed in the variable SYSTEMS="myrtle gull sandy newport daytona":

```
for sys in $SYSTEMS x
   do
       finger @$sys
       print
   done
```

for (cont.)

`list` can contain shell wildcards and command substitution as well:

```
$ cat file1
 for i in *
 do
     print $i
 done

 $ file1
 file1
 file2

 $ cat file2
 for i in $(ls)
 do
     print $i
 done

 $ file2
 file1
 file2
```

case

- **Provides a multiple choice decision structure.**
- **Lets you test strings against patterns that can contain wildcard characters.**
- **Syntax:**

```
case expression in
      pattern1)
          statements ;;
      pattern2)
          statements ;;
      ...
  esac
```

Case (cont.)

- If expression matches one of the patterns, its corresponding statements are executed
- If there are several patterns separated by pipes, the expression can match any of them in order for the associated statements to be run
- Patterns are checked in order for a match; if none is found, nothing happens

Case (cont.)

Here's a simple script which moves C and fortran source files to one directory and object code files to another:

```
for file in $*; do
      case $file in
        *.c|*.f)
          /bin/mv $file ${HOME}/src ;;
        *.o    )
          /bin/mv $file ${HOME}/obj;;
        *      )
          print $file not moved ;;
      esac
   done
```

(Could have also used `*.[cf]` wildcard construct above)

OSC
Innovations in computing, networking, and education

Case (cont.)

- The case statement is often used to specify options for a shell script.
- Here is a shell script called `dt_fmat` that allows the user to enter options that affect the way the date is displayed:

```
case $1 in
  -d) print -n Date:
      date +"%a %h %d" ;;
  -t) print -n Time:
      date +"%T" ;;
  -w) print -n Weekday:
      date +"%a" ;;
  -n) print -n Date:
      date +"%D" ;;
  -y) print -n Year: 19
      date +"%y" ;;
  -m) print -n Month:
      date +"%h" ;;
```

Case (cont.)

```
*)  print -n Date:
          date +"%a %h %d" ;;
          print -n Time:
          date +"%T" ;;
      esac

      $ dt_fmat -y
      Year: 1996
      $ dt_fmat *
      Date:Thu Oct 17
      Time:14:33:36
```

select

- `select` allows you to generate simple menus easily.

- Syntax:

```
select name [in list]
   do
       statements that can use $name
   done
```

- This is the same syntax as the for loop except for the keyword select. As with `for`, in list defaults to `"$@"` if omitted.

- A menu is generated for each item in list, formatted with number for each choice.

Select (cont.)

- The selected choice is stored in name and the selected number in **REPLY**
- Executes the statements in the body
- Repeats the process forever; exit loop with break statement (or user can issue ctrl-d)

Select (cont.)

The following script **termselect** allows you to select a terminal setting:

```
PS3='terminal? '
   select term in vt100 vt220 xterm
   do
       if [[ -n $term ]]; then
          TERM=$term
          print TERM is $term
          break
       else
          print invalid choice
       fi
   done
vt100 vt220 xterm
```

<--- the list can be expanded for clarity using continuation lines and quotes

OSC Innovations in computing, networking, and education

Select (cont.)

```
$ termselect
   1) vt100
   2) vt220
   3) xterm
   terminal? 4
   invalid choice
   terminal? 3
   TERM is xterm
```

while & until

- Allows a section of code to be run repetitively while a certain condition holds true.
- Syntax:

```
while condition
   do
       statements ...
   done
```

- As with `if`, the condition is really a list of statements that are run; the exit status of the last one is used as the value of the condition.
- `[[ ]]` can be used here as with `if`.
- Beware of creating an infinite loop (condition must become false at some point).

while & until (cont.)

Example: print out arguments

```
$ cat args
  while [[ $# -ne 0 ]]; do
     print $1
     shift
  done

  $ args "a dog" '$x' "$x" a 1
  a dog
  $x

  a
  1
```

- Until allows a section of code to be run repetitively as long as a certain condition is false.

- Just about any until can be converted to a while by simply negating the condition.

OSC
Innovations in computing, networking, and education

Command-line Arguments

- We want to expand on our ability to use command-line options to shell scripts

- Typical UNIX commands have the form

```
command [-options] arguments
```

  meaning that there can be 0 or more options.

- A piece of code that handles a single option called `-o` and arbitrarily many arguments would be:

```
if [[ $1 = -o ]]; then
        process the -o option
        shift
   fi

   normal processing of arguments ...
```

Command-line Arguments (cont.)

Example:

- **Suppose you keep a list of your home coin collection that keeps track of how many coins you have in a given category. Lines in the list look like:**

```
62 U.S. proofs
11 U.S. pennies (1850-1908)
36 U.S. pennies (1909-1950)
9   U.S. nickels (1861-1938)
```

- **You want to write a program that prints the N types of coins of which you have the most. The default for N is 10. The program should take one argument for the name of the input file and an optional argument for how many lines to print.**

- **A simple implementation would be:**

```
filename=$1
   howmany=${2:-10}
   sort -nr $filename | head -$howmany
```

OSC
Innovations in computing, networking, and education

Command-line Arguments (cont.)

- This script is usable but if no arguments are given to the script it will appear to hang
- No useful error messages
- Doesn't conform to typical UNIX command syntax

- An improvement (script is named highest):

```
if [[ $1 = -+([0-9]) ]]; then
      howmany=$1
      shift
   elif [[ $1 = -* ]]; then
      print usage: highest [-N] filename
      return 1
   else
      howmany=-10
   fi

   filename=$1
   sort -nr $filename | head $howmany
```

Command-line Arguments (cont.)

For multiple options, a general technique would be (assume script named **exo**):

```
while [[ $1 = -* ]]; do
    case $1 in
      -a) process option -a ;;
      -b) process option -b ;;
      -c) process option -c ;;
      * ) print usage: exo [-a] [-b] [-c] args; return 1 ;;
    esac
    shift
  done
```

normal processing of arguments ...

Command-line Arguments (cont.)

Suppose option b takes an argument itself:

```
while [[ $1 = -* ]]; do
     case $1 in
       -a) process option -a ;;
       -b) process option -b
           $2 is the option's argument
           shift ;;
       -c) process option -c ;;
       * ) print usage: exo [-a] [-b option] [-c] args; return
1 ;;
     esac
     shift
   done
```

normal processing of arguments ...

OSC
Innovations in computing, networking, and education

Integer Variables & Arithmetic

- The shell interprets words surrounded by `$ (( and ))` as arithmetic expressions.
- Variables in arithmetic expressions do not need to be preceded by dollar signs.

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division with truncation
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
==	equal
!=	not equal
&&(||)	logical and (or)

Integer Variables & Arithmetic (cont.)

- No need to backslash escape special characters within `$((...))` syntax.
- Parentheses can be used to group subexpressions.
- Relational operators have true values of 1 and false values of 0.
- Examples:

```
$ print $ ((3 > 2 ))
  1
  $ print $ (( (3 > 2 ) || (4 <= 1) ))
  1
  $ print $ (( 2#1001 ))
  9                    <--using base 2
```

Integer Variables & Arithmetic (cont.)

- **Can also construct arithmetic condition tests; these set an exit status of 0 if true and 1 if false**

```
$ ((3 > 2 )) ;print $?
  0
  $ ((3 < 2 )) ;print $?
  1
```

<-- note no leading $ sign!

Integer Variables & Arithmetic (cont.)

- **"Truth" values:**

```
$ (( 14 )) ;print $?
  0
  $ (( 0 )) ;print $?
  1
```

- **Assigning expressions to integer variables with the let command:**

```
let x=                          $x
1+4                             5
'1 + 4'                         5
'(2+3) * 5'                     25
17/3                            5

$ let x=5; let y=6
$ let z=x+y
$ print $z
11
```

OSC
Innovations in computing, networking, and education

Exercises

1. Write a script called `lis` that uses a for loop to display all files and directories in the current directory.

2. Write a script called `char` that checks a single character on the command line, c. If the character is a digit, digit is displayed. If the character is an upper or lowercase alphabetic character, letter is displayed. Otherwise, other is displayed. Have the script print an error message if the argument c is more than one character in length.

3. Write a script called `mycopy` that copies a source file to multiple destinations. Add a check to see if the source file exists. If the source file does not exist, print an error message.

4. Write a script called `mydir` that prints the message File is a directory if the file is a directory.

Exercises (cont.)

5. Write a script called `ver` that accepts one argument on the command line indicating the name of a file. The file is copied to another file with the same name with the addition of the extension `.v2`. Also, the line #Version 2 is added to the top of the file.

6. Execute the `ver` script on itself, creating a new version of the file called `ver.v2`.

7. Rewrite `ver.v2` to accept a possible second argument. If two arguments are entered, the file specified by the first argument is copied to a file with the name of the second argument. If no second argument is entered, the file is copied to another file with the same name, adding the extension `.v2`. In either case, the line `#Version2` is added to the top of the file.