Week3 - Assigning Files As Descriptors; Piping; Multiple Commands; Lists; Looping

Assigning Files As Descriptors

- any of file descriptors 0 through 255 can be used
- exec 4> filename - output or errors redirected to file descriptor 4 will be written to filename
- file is opened until explicitly closed, so appending is automatic
- can be closed with command exec 4>&-
    - example of use:

```
exec 4> myFiles
echo "Here are my current files" >&4
ls -l >&4
exec 4>&-
```

    - this is functionally equivalent to:

```
echo "Here are my current files" > myFiles
ls -l >> myFiles
```

- assigning a file descriptor is more efficient, because the file is opened and closed only once, instead of during each redirection
- exec 4< filename - input redirected from file descriptor 4 will be read from filename
    - example of use:

```
exec 4< myFiles
read line1 <&4
read line2 <&4
exec 4<&-
echo $line1
echo $line2
```

- assigning a file descriptor keeps the file open until explicitly closed, allowing the reading of one line at a time
    - in the following example, the first line of the file would be read twice because the file is opened and closed for each read:

```
read line1 < myFiles
read line2 < myFiles
echo $line1
echo $line2
```

- exec 4<> filename - the file will be opened for both input and output
    - example of use:

```
exec 4<> myFiles
read line1 <&4
read line2 <&4
echo "This will be a new third line" >&4
exec 4<&-
```

    - the file remains open until explicitly closed - two lines are read, then the output of the "echo" gets written to the file at that location

- open file descriptors can be found using ls /proc/$$/fd
    - note that bash uses fd 255 internally to connect to the terminal
    - for example, try the following:

```
exec 255>&-
cat cars | more
exec 255>&1
cat cars | more
```

- the standard file descriptors can also be closed and re-opened
    - for example, try the following, one command at a time:

```
exec 6>&1
exec 1>&-
ls
ls > /dev/tty
exec 1>&6 6>&-
ls
```

Piping

- | (pipe) will connect the standard output of the command to its left, to the standard input of the command to its right:

```
find . -name "*.c" 2> /dev/null | more
find . -name "*.c" 2> /dev/null | cut -c3- | more
```

- tee command will take standard input from a pipe, and send it as output to one or more files and to its standard output:

```
ls -al | tee file1 file2
```

- can redirect (or tee) to the file that represents the display unit:

```
ls -al | tee /dev/tty | wc -l
```

- xargs command will take standard input from a pipe, and send it as arguments to the following command:

```
find . -size +150k
find . -size +150k | xargs ls -ld
```

    - note that if any of the filenames contains whitespace, the command after "xargs" will misinterpret as multiple files

- we can tell "find" to use nulls (instead of newline) to separate the filenames using the -print0 option
    - this allows programs processing the "find" output to correctly interpret filenames containing whitespace
    - then we can tell "xargs" to expect null-delimited input by using the -0 option
    - for example:

```
find . -size +150k -print0 | xargs -0 ls -ld
```

- some commands can't handle a null separator
    - the following will not work if filenames contain whitespace:
        find . -size +150k -print0 | cut -c3- | xargs -0 ls -ld

    - instead, we'll feed "cut" the expected newline characters
    - then we can use "tr" to translate newlines to nulls
    - for example:
        find . -size +150k | cut -c3- | tr "\n" "\0" | xargs -0 ls -ld

- to pipe both stdout and stderr:
    find . -size +1M 2>&1 | more
    find . -size +1M |& more
    - only works with the latest versions of the bash shell

- to pipe stderr only:
    find . -size +1M 2>&1 >/dev/null | more
    - note that all piped commands will be executed in a subshell, so any variable assignments will not be seen outside of the subshell
    - for example, try the following on the command line:

    echo "Hello there" | read greeting
    echo $greeting

- <<<'string' - "here string", can help alleviate some of these piping problems:

```
==> name="Josephine Smith"==> echo $name | read first last==> echo "First name:
$first, Last name: $last"First name: , Last name:==> read first last <<< $name==>
echo "First name: $first, Last name: $last"First name: Josephine, Last name: Smith==>
_
```

Multiple Commands

- besides piping, there are other ways that multiple commands may be placed in one line
    - commands may be separated by semi-colons
    - each command will be executed when the previous command has terminated
    - for example:

```
sleep 5; ls
```

- commands may be grouped by using brace brackets, and redirected as a group:

```
{ echo "Files in $PWD"; ls -l; } > current_files
```

    - will execute the grouped commands and redirect all output to current_files

- the grouped commands will be executed within the current shell, so any variable assignments will be seen outside of the grouping
    - for example, try:

```
num=5
{ ((num = num + 7)); echo $num; }
echo $num
```

- commands may be grouped by using parentheses, and redirected as a group:

```
(date; echo "Who is on:"; who) > current_users
```

    - will execute the grouped commands and redirect all output to current_users

- the grouped commands will be executed in a subshell, so any variable assignments will not be seen outside of the grouping
    - for example, try:

```
num=5
( ((num = num + 7)); echo $num )
echo $num
```

- commands may also be split over multiple lines, making it easier (for humans) to interpret a long command
    - quote or "escape" the newline character at the end of a line, to get rid of the special meaning of newline (to end a command line)
    - for example:

```
echo "This will be split over multiple \
lines. Note that the shell will realize \
that a pipe requires another command, so \
it will automatically go to the next line" |
tr '[a-z]' '[A-Z]'
```

Lists

- AND list
  - list of statements separated by &&
  - statements will be executed till one fails, giving a non-zero exit status
  - some examples:
    - [$# != 2] && echo "This command requires two arguments" >&2
    - echo $1 | grep "[^0-9]" && echo "First argument is not numeric" >&2
    - ["$1" -gt 0] && ["$1" -le 26] && echo abcdefghijklmnopqrstuvwxyz | cut -c$1
      - produces a very compact "if-then" type structure

- OR list
  - list of statements separated by ||
  - statements will be executed till one succeeds, giving a zero exit status
  - for example:
    - [! -f "$1"] || [! -r "$1"] || [! -d "$2"] || [! -w "$2"] || [! -x "$2"] || cp $1 $2
  - && and || can be combined, they just check the exit status of the previously executed command
  - for example:

```
$ xxx=3$ [ $xxx -gt 5 ] && echo '$xxx is > 5' || echo '$xxx is <= 5'xxx is <= 5$
xxx=7$ [ $xxx -gt 5 ] && echo '$xxx is > 5' || echo '$xxx is <= 5'xxx is > 5
```

- but this is tricky logic, and assumes that the second command will succeed if executed
- for example, assuming "file1" doesn't exist:

```
$ xxx=3$ [ $xxx -gt 5 ] && echo '$xxx is > 5'; grep -qs 'some-string' file1 || echo
'$xxx is <= 5'xxx is <= 5$ xxx=7$ [ $xxx -gt 5 ] && echo '$xxx is > 5'; grep -qs
'some-string' file1 || echo '$xxx is <= 5'xxx is > 5xxx is <= 5
```

Looping

for-in

- for is used to execute statements for a specifed number of repetitions
- a loop variable takes the values of a specified list, one at a time
- for example, to process a list of strings:

```
for animal in lion tiger beardoecho $animaldone
```

- to process space-delimited strings within a variable:

```
animals="lion tiger bear"for animal in $animalsdoecho $animaldone
```

- to process a list created by command substitution:

```
animals="lion tiger bear"for animal in $(echo $animals | tr ' ' '\n' | grep
"i")doecho $animaldone
```

- to process filenames in a directory, using command substitution:

```
for file in $(ls $1)doecho $filedone
```

- note that a path is not included, try the following:

```
for file in $(ls $1)dols -ld $filedone
```

- to make this work, the path needs to be specified:

```
for file in $(ls $1)dols -ld $1/$filedone
```

- note that we need some checking in case $1 is missing or invalid
- or, to process filenames in a directory using filename expansion, which includes path information:

```
for file in $1/*doecho $filedone
```

- to execute a loop 4 times:

```
for count in 3 2 1 "BLAST OFF!!!"doecho $countsleep 1done
```

- another way to execute a loop 4 times, using the seq command with command substitution:

```
for count in $(seq 3 -1 1) "BLAST OFF!!!"doecho $countsleep 1done
```

- "seq 20" produces the numbers 1 to 20, incremented by 1
- "seq 5 15" produces the numbers 5 to 15, incremented by 1
- "seq -10 2 10" produces the numbers -10 to 10, incremented by 2

- another way to execute a loop 4 times, using brace expansion:

```
for count in {3..1} "BLAST OFF!!!"doecho $countsleep 1done
```

  - "echo {1..20}" produces the numbers 1 to 20, incremented by 1
  - "echo {5..15}" produces the numbers 5 to 15, incremented by 1
  - "echo {-10..10..2}" produces the numbers -10 to 10, incremented by 2

  - a way to execute a loop 3 times, using C-style shell arithmetic:

```
for (( count = 3; count >= 1; count-- ))doecho $countsleep 1doneecho "BLAST OFF!!!"
```

for

- for without the "in" keyword - loop variable takes value of arguments $1, $2, $3, etc.

```
for args     # Note that "args" is a user-defined variabledoecho $argsdone
```

- another example:

```
for filedoif [ -f "$file" ]thenecho "$file is an ordinary file"elif [ -d "$file"
]thenecho "$file is a directory"elseecho "$file is not an ordinary file or
directory"fidone
```

while

- while control structure, loop while condition remains true (0 exit status)
  - condition testing is similar to the "if" statement
- to read from the keyboard:

```
input=while [ "$input" != end ]doecho -n "Type something: "read input[ "$input" !=
end ] && echo "You typed: '$input'"done
```

- the loop condition can be several statements, the exit status of the last statement determines loop termination:

```
while echo -n "Type something: "read input[ "$input" != end ]doecho "You typed:
'$input'"done
```

- to read from a file, the following would NOT work, the first line of the file would be displayed continuously:

```
while read input < carsdoecho "Input line is: $input"done
```

  - this is because by redirecting to "read", the file is opened and closed by "read" at each iteration

- to read from a file, the file has to be redirected or piped to the while loop, not to the read statement
    - for example:

```
cat cars |while read inputdoecho "Input line is: $input"done
```

- note that "read" is successful if it can read a line, and fails on end-of-file
- however, a pipe creates a child process, so any variable changes are local:

```
lines=0cat cars |while read inputdo((lines++))echo "Input line #$lines is:
$input"doneecho "$lines lines were read"
```

- another way to read from a file, without the local variable problem:

```
lines=0exec 3< carswhile read input <&3do((lines++))echo "Input line #$lines is:
$input"doneexec 3<&-echo "$lines lines were read"
```

- and another way:

```
lines=0while read inputdo((lines++))echo "Input line #$lines is: $input"done <
carsecho "$lines were read"
```

until

- until control structure, loop until test becomes true (0 return code), the opposite of while

```
input=until [ "$input" = end ]doecho -n "Type something: "read input[ "$input" != end
] && echo "You typed: '$input'"done
```