Shell redirection and scripting

Complete the exercises below and demonstrate your solutions to a TA.

Part I: Shell redirection of standard output

In output redirection, the shell (not the command) diverts (redirects) most command output that would normally appear on the screen to some other place, either into the input of another command (using a pipe meta-character '|') or into a file (using a file redirect meta-character '>').

The shell meta-character '>' signals that the next word on the command line is an output file (not a program) that should be created or truncated (set to empty) and made ready to receive the standard output of a command:

```
    $ date >outfile
```

The file is always created or truncated to empty *before* the shell finds and runs the command, unless you double up the character like this:

```
    $ date >> outfile   # output is *appended* to outfile; no truncation
```

An example of redirection of output into a file:

```
    $ echo hello                         # output goes to terminal (screen)
hello

    $ echo hello >file                   # erase file; send output to file
$ cat file                           # display what is in the file
hello

    $ echo there >>file                  # append output to end of file
$ cat file                               # display what is in the file
hello     there
```

It is the shell that creates or truncates the file and sets up the redirection, not the command being redirected. The command knows nothing about the redirection - the redirection syntax is removed from the command line before the command is found and executed:

```
    $ echo one two three                 # echo has three arguments
one two three
    $ echo one two three >out            # echo still has three arguments
    $ cat out
one two three
```

Shells handle redirection before they go looking for the command name to run. Indeed, you can have redirection even if the command is not found or if there is no command at all:

```
    $ nosuchcommandxxx >out                # file "out" is created empty
sh: nosuchcommandxxx: command not found
    $ wc out
     0 0 0 out                             # shell created an empty file
    $ >out                                 # file "out" is created empty
    $ wc out
     0 0 0 out                             # shell created an empty file
```

The shell creates or truncates the file "out" empty, and then it tries to find and run the nonexistent command and fails. The empty file remains. Any existing file will have its contents removed:

```
    $ echo hello >out ; cat out
hello
    $ nosuchcommandxxx >out
    sh: nosuchcommandxxx: command not found
    $ wc out
     0 0 0 out                             # shell truncated the file
```

Shells don't care where on or in the command line you do the file redirection. The file redirection is done by the shell, then the redirection syntax is removed from the command line before the command is called. The command actually being run doesn't see any part of the redirection syntax; the number of arguments is not affected.

All the command lines below are equivalent to the shell; in every case the echo command sees only three arguments and the three command line arguments "hi", "there", and "mom" are all redirected into "file":

```
    $ echo hi there mom >file                  # echo has three arguments
    $ echo hi there >file mom                  # echo has three arguments
$ echo hi >file there mom                 # echo has three arguments     $
echo >file hi there mom                  # echo has three arguments     $
>file echo hi there mom                  # echo has three arguments
```

Exercise 1

Answer the following questions. Put your answers in a file named answer1.txt.

a. Explain this sequence of commands:

```
    $ mkdir empty
    $ cd empty
$ cp a b
    cp: cannot stat `a': No such file or directory

    $ cp a b >a
    $                # why is there no error message from cp this time?
```

b. Explain this sequence of commands:

```
    $ date
    Wed Feb  8 03:01:11 EST 2012
    $ date >a
    $ cat a
    Wed Feb  8 03:01:21 EST 2012

    $ cp a b
    $ cat b
    Wed Feb  8 03:01:21 EST 2012

    $ cp a b >a
    $ cat b
    $                               # why is file b empty?
```

c. Explain this sequence of commands:

```
    $ rm
    rm: missing operand

    $ touch file     $ rm >file     rm: missing operand
# why doesn't rm remove "file"?

    $ rm nosuchfile
    rm: cannot remove `nosuchfile': No such file or directory

    $ rm nosuchfile >nosuchfile
    $                            # why is there no rm error message here?
```

Standard output ("stdout") and Standard error ("stderr")

Most commands have two separate output "streams", numbered 1 and 2:
1. stdout - unit 1 - Standard Output (normal output)
2. stderr - unit 2 - Standard Error Output (error and warning messages)

The normal (non-error) "unit 1" outputs on your screen come from the "standard output" ("stdout") of the command. Stdout is the output from "printf" and "cout" statements in C and C++ programs, and from "System.print" and "System.println" in Java. This is the expected, usual output of a command.

The error message "unit 2" outputs on your screen come from the "standard error output" ("stderr") of the command. Stderr is the output from "fprintf(stderr" and "cerr" statements in C and C++ programs, and from "System.err.print" and "System.err.println" in Java. Programs print on this output only for error messages.

The stdout and stderr mix together on your terminal screen. They look the same on the screen, so you can't tell by looking at your screen what comes out of a program on stdout and what comes out of a program on stderr.

To show a simple example of stdout and stderr both appearing on your screen, use the "ls" command and give it one file name that exists and one name that does not exist (and thus causes an error message to be displayed):

```
  $ ls -l /etc/passwd nosuchfile   ls: nosuchfile: No such file or
directory            # standard error   -rw-r--r-- 1 root root 2209 Jan
19 20:39 /etc/passwd  # standard output
```

The stderr (error messages) output often appears first, before stdout, due to internal I/O buffers used by commands for stdout.

Normally, both stdout and stderr appear together on your terminal. The shell can redirect the two outputs individually or together into files or into other programs. The default type of output redirection(whether redirecting to files or to programs using pipes) redirects only standard output and lets standard error go, untouched, to your terminal.

Below are some examples all using the shell file redirect meta-character '>':

```
  $ ls /etc/passwd nosuchfile                # no redirection used   ls:
nosuchfile: No such file or directory  # this on screen from stderr
/etc/passwd                               # this on screen from stdout
  $ ls /etc/passwd nosuchfile >out           # shell redirects only stdout
ls: nosuchfile: No such file or directory  # only stderr appears on screen
  $ cat out
  /etc/passwd
```

You can redirect stdout and stderr separately into files using unit numbers before the '>' meta-character: - stdout is always unit 1 and stderr is always unit 2 (stdin is unit 0) - put the unit number immediately (no blank) before the '>' meta-character.

">foo" (no preceding unit number) is a shell shorthand for "1>foo"

">foo" redirects the default unit 1 (stdout) only, not stderr
">foo" and "1>foo" are identical

You can also tell the shell to redirect standard error, unit 2, to a file:

```
  $ ls /etc/passwd nosuchfile 2>errors       # shell redirects only stderr
  /etc/passwd                                # only stdout appears on screen
  $ cat errors
  ls: nosuchfile: No such file or directory
```

You can redirect stdout into one file and stderr into another file:

```
  $ ls /etc/passwd nosuchfile >out 2>errors  # shell redirects each one
  $                                          # nothing appears on screen
  $ cat out
  /etc/passwd
  $ cat errors
  ls: nosuchfile: No such file or directory
```

You needed a special syntax "2>&1" to redirect both stdout and stderr safely together into a single file in the Bourne shells. Read the syntax "2>&1" as "send unit 2 to the same place as unit 1":

```
  $ ls /etc/passwd nosuchfile >both 2>&1     # redirect both into same file
$                                          # nothing appears on screen
  $ cat both
  ls: nosuchfile: No such file or directory
  /etc/passwd
```

The order of >both and 2>&1 on the command line matters!

The ">both" stdout redirect must come first (to the left of) stderr "2>&1" because you must set where stdout (unit 1) goes before you send stderr(unit 2) to go "to the same place as unit 1". Don't reverse these!

You must use the special syntax ">both 2>&1" to put both stdout and stderr into the same file. Don't use the following, which is not the same:

```
  $ ls /etc/passwd nosuchfile >wrong 2>wrong # WRONG! DO NOT DO THIS!

  $ cat wrong
/etc/passwd
  ccess nosuchfile: No such file or directory
```

This above WRONG example will cause stderr and stdout to overwrite each other and the result is a mangled output file; don't do this.

Part III: Shell redirection of standard input

The shell meta-character '<' signals that the next word on the command line is an input file (not a program) that should be made available to a command on standard input. Using the shell meta-character '<', you can tell the shell to use input redirection to change from where standard input comes, so that it doesn't come from your keyboard but instead comes from an input file.

You can only use standard input redirection on a command that would otherwise read your keyboard. Here are examples using the shell to attach files to commands that are all reading standard input:

```
  $ cat food              # reads from file "food"
  $ cat                   # reads from stdin (from your keyboard)
  $ cat <food             # reads from stdin (now from the file "food")
  [...etc. for all commands that can read from stdin...]
```

The shell does not know which commands will actually read input from standard input; you can attach a file on standard input to any command. A command that ignores standard input will ignore the attached file.

If a command is not reading from standard input, redirecting input into the command will be ignored and do nothing. The shell cannot force a command to read from standard input. For example, the date command and the sleep command never read from standard input, and you can't force them by adding redirection:

```
  $ date                          # date never reads stdin
  Thu Feb 16 05:48:13 EST 2012
  $ date <file                    # date never reads stdin and ignores
<file
  Thu Feb 16 05:48:15 EST 2012
  $ echo 30 >file                   # first, put the number 30 into a file
$ sleep 10                        # sleep never reads stdin
  $ sleep 10 <file                  # sleep never reads stdin; ignores <file
$ sleep    <file                  # sleep never reads stdin; ignores <file
sleep: too few arguments
```

Commands never read both pathnames and standard input; it's one or the other, and command argument pathnames are always used instead of stdin. So if a file can be supplied as a command line pathname or attached to a command via standard input, what is the difference? Note the difference between "cat food" and "cat <food":

```
  $ cat food
```

- the cat command has a pathname argument, which means it ignores stdin
- the "cat" command is opening the file argument "food", not the shell
- any errors will come from the "cat" command and will mention the file
- name "food", e.g. cat: food: Permission denied
- the cat command reads data from the file it opened itself

```
$ cat <food
```

- the cat command has no arguments, which means it will read standard input • the shell is performing standard input redirection from file "food", which means standard input for "cat" will come from file "food"
- the shell itself is opening the file "food", not the "cat" command
- any errors will come from the shell, not from the "cat" command, e.g. bash: food: Permission denied
- the cat command reads data from standard input, opened by the shell

For commands that display their input pathnames in their output, the above difference is more significant. If no pathnames are supplied on the command line and all the data comes from standard input, there is no file name available to the command to indicate in the output:

```
  $ wc -l /etc/passwd
  44 /etc/passwd
```

- wc was passed the file name "/etc/passwd" as a command line pathname argument and so wc had to open the file itself
- wc knows the file name, so it prints the name in the output

```
   $ wc -l </etc/passwd
     44
```

- the shell opens the standard input redirection file "/etc/passwd" and attaches it to standard input for the command, which is "wc"
- wc was given no file arguments and so reads the data from standard input; wc doesn't know the file name; only the shell knows the name, so wc does not print any file name. wc cannot know the file name!

Part IV: Redirection into programs (Pipes)

Since the shell can redirect both the output of programs and the input of programs, it can connect (redirect) the output of one program into the input of another program. This is called "piping" and uses the "pipe" meta-character '|' (shift-'\'), e.g. $ date | wc

Rules for Pipes:

1. Pipe redirection is done by the shell, first, before file redirection.
2. The command on the left of the pipe must produce some standard output.
3. The command on the right of the pipe must want to read standard input.

The shell meta-character "|" ("pipe") signals the start of another command on the command line. The standard output (only stdout; not stderr) of the command on the immediate left of the "|" is attached/connected("piped") to the standard input of the command on the immediate right:

```
  $ date
  Mon Feb 27 06:37:52 EST 2012
  $ date | wc
    1 6 29
```

(Note that the newline character at the end of a line is counted by wc.)

Recognizing pipes and splitting a command line into piped commands is done first, before doing file redirection. File redirection happens second (after pipe splitting), and if present, has precedence over pipe redirection. (The file redirection is done after pipe splitting, so it always wins, leaving nothing for the pipe.)

```
  $ ls -l      | wc          # correct - output of ls goes into the pipe
2 11 57

  $ ls -l >out | wc          # WRONG! - output of ls goes into the file
    0 0 0                    # wc reads an empty pipe and outputs zeroes
```

- shell first splits the line on the pipe, redirecting the output of the command on the left into the input of the command on the right, but: • then the shell processes the standard output file redirection on the "ls" on the left and changes the "ls" standard output into the file "out"
- finally, the shell finds and runs both commands simultaneously
- all the standard output from "ls" goes into the file "out"; nothing is available to go into the pipe
- wc counts an empty input from the pipe and outputs: 0 0 0

Note that many Unix commands can be made to act as "filters" – reading from stdin and writing to stdout, all supplied by the shell, without opening any pathnames themselves. With no file names on the command line, the commands read from standard input and write to standard output. The shell provides redirection for both standard input and standard output:

```
    $ grep "/bin/sh" /etc/passwd | sort | head -5
```

The "grep" command above is reading from the filename argument /etc/passwd given on the command line. (When reading from files, commands do not read from standard input. File names take priority over standard input.)

The "sort" and "head" commands have no file names to read; this means they read from standard input, which is set up to be pipes by the shell. Both "sort" and "head" are acting as filters; they are reading from stdin and writing to stdout. (The "grep" command is technically not a filter - it is reading from the supplied argument pathname, not from stdin.)

Remember: if file names are given on the command line, the commands ignore standard input and only operate on the file names. Look at this small change to the above pipeline:

```
    $ grep "/bin/sh" /etc/passwd | sort | head -5 /etc/passwd    # WRONG!
```

Above is the same command line as the previous example, except the "head" command is now ignoring standard input and is reading directly from its /etc/passwd filename argument. The "grep" and "sort" commands are doing a lot of work for nothing, since "head" is not reading the output of sort coming down the pipe. The head command is reading from the supplied file name argument /etc/passwd instead. File names take precedence over standard input.

Exercise 2

Answer the following questions. Put your answers in a file named answer2.txt. Hint: the commands head, tail, ls, and sort are your friends.

a. Create a command line that displays only lines 6-10 of the password file.
b. Create a command line that displays only the second-last line of the password file.
c. Create a command line that lists the five largest files in current directory.

Part V: Shell scripting

Shell scripts are text files with programming style commands (variables, assignments, conditionals, loops, etc) arranged in top down order of execution, unless a control structure is used to change this. Typical names for shell script files end in .sh such as script.sh but the extension is not required. You invoke a shell script file by the sh command followed by the script file name:

```
$ sh script.sh
```

Or you can just use the file name if you have executable permission on the file. If you have trouble running your shell script, use the *change mode* command to give yourself permission to execute it and everyone else to read it if desired:

```
$ chmod 744 myscript
```

Bash shell script files should start with the first line being

#!/bin/bash

to indicate we are calling the bash shell to interpret the shell script. The symbol # usually starts a comment so the first line is a special code that is not to be interpreted as a command but a directive instead. Bash shell script commands are fussy about spaces or lack of spaces so be sure to try variants on your script commands if they are being flagged as errors. Also note that strings with embedded blanks need quotes "This string needs quotes" around it to be parsed as a string. Enclosing strings in 'single quotes' or even `back quotes` are sometimes needed to be parsed correctly. Also remember that regular expressions are used in some shell commands so we may need to *escape* characters by using backslash such as *, etc.

Look online for a couple of Bash shell script tutorials or references you find useful and use them to help you work through this part. Two useful references are:

tldp.org/LDP/Bash-Beginners-Guide/html/chap_01.html

tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-10.html

Shell Variables

Shell variables are assigned by equality = but accessed by prefixing a $ symbol. Shell variables follow the same conventions as program variables. Some examples follow. Type or copy these commands into a file testing.sh and try it out.

```
#!/bin/bash # This
is a comment
myVariable="A long string variable here." echo
$myVariable
echo myVariable # This will print myVariable since we aren't using the $
prefix
```

Just to appreciate the fussy nature of shells, put an extra space between the = and " in the assignment and rerun the shell script. Now you know.

You can also read and make assignments from stdin using the read command. Here we read input from the user and echo the resulting variable assignment:

```
#!/bin/bash
myName="A long string variable here."
echo "Please enter your name: " read
myName
echo Your name is $myName
```

or several variables can be scanned:

```
#!/bin/bash
echo "Please enter your full name: "
read firstName middleName lastName
echo Your first name is $firstName
echo Your middle name is $middleName
echo Your last name is $lastName
```

You can use the local variable $@ to get the full list of command line parameters and $# holds the number of such parameters.

```
#!/bin/bash echo You
entered $# words echo The
whole list is: $@
```

String vs. Arithmetic

One major difference between bash, which is a scripting language, and C, is that you don't need to identify the type of the data you're storing to a variable. Instead, you have to provide some context for how you want the data to be interpreted, otherwise, by default, it will be treated like strings and + is concatenation.

```
n=1 n=1+$n echo $n   #<-- print
"1+1" not 2!
```

To perform arithmetic operations, you use the `let` operations.

```
n=1
let n=1+$n #<-- use quotes if you need white space echo
$n    #<-- prints 2!
```

There is also no such things as floats in bash. Everything is a numeric integer.

Control Structures

Check out in your tutorial how to use the if...fi control structure to do branching. This often requires the test conditions enclosed in square brackets such as

```
echo Enter a file name and I will check to see if it exists
read myFile if [ -r $myFile ]; then
      echo There is a readable file called $myFile else
      echo $myFile does not exist fi
```

Use your tutorial for a full reference on the [test] conditions using -r, -w, =, -eq, etc. Some take filenames, others use integer values. CAUTION: the blank spaces inside the [test] format are NOT optional.

Here is an example script that checks if the script was run from the home directory: `echo`

```
"Hello $USER"


if [ $HOME == $PWD ] then
    echo "Good, you're in your home directory: $HOME" else
    echo "What are you doing away from home?!?"
    cd $HOME

    echo "Now you are in your home directory: $PWD" fi
```

There are two things of note here. First, the [$HOME = $PWD] is a command that succeeds only when $HOME is the present working directory. Also, when you change directories within a script, you are not changing the directory of the shell you ran the script from, just the present working directory of the script itself.

And of course there is a lot more: other kinds of loops, for...do...done, case...esac constructions, sleep, arrays, etc. Consult your favorite shell programming tutorial for more examples and other functions beyond those briefly indicated here. Below are some exercises to try your hand at for doing shell programming.

Exercise 3

Write a script, `getsize.sh`, which takes a path as an argument and prints out the size of the file/dir at that path. Your script **must do error checking**. Here is some sample output:

```
#>./getsize.sh
ERROR: Require file
#>./getsize.sh adadf
ERROR: File adadf does not exist
#>./getsize.sh empty.txt
0
#>./getsize.sh larger.txt
5000
```

You should be able to use a cut, ls, and/or wc to get the information you need. All errors should be written to stderr such that:

```
#> ./getsize badfile > /dev/null
ERROR: File badfile does not exist
```

You may also find it useful to use the tr command. Consult the man page for more details, but tr is used for translation, substituting one string for another. The *squeeze* option -s, in particular, could be useful to get rid of extra whitespace so that your cut fields are more consistent.

Useful references:

http://www.thegeekstuff.com/2013/06/cut-command-examples
http://www.thegeekstuff.com/2012/12/linux-tr-command/