CS 246 Fall 2018 — Tutorial 1

September 19, 2018

Summary

1 Input, Output, and Error

There are three ways we interact with programs once they are running:

- Input: keyboard, typically not read until a newline occurs
- Output: terminal, buffered
- Error: terminal, not buffered

Since both output and error are printed to the terminal, data printed to output and error will appear in the terminal with no visual difference and mixed together.

2 Redirection

Redirection is overriding the default method which for input, output, and/or error.

2.1 Input Redirection

Giving a file as the standard input to a program.

Example: `wc < mywords.txt`

Note that some programs behave differently when input is redirected than when it is given a command-line argument. For example, `wc` prints the name of the file when it is given a command-line argument, but does not when the input is redirected. When redirecting input, the shell opens the file which means the program never sees the name of the file.

2.2 Output/Error Redirection

Sending the output (or error) to somewhere other than the terminal.

- Suppose we have a program (`printer` — prints even numbers to stdout, odd to stderr) that prints to standard output and standard error.
- To redirect stdout to `print.out` and stderr to `print.err`:

  - `./printer > print.out 2> print.err`

- To redirect the output from standard output to standard error:
  - `echo "ERROR" >&2`
- What would be the purpose of redirecting output to `/dev/null`?
  - When we do not care about the actual output of the program but want it to perform some operation (e.g. checking if files are the same, finished successfully).

2.3 Pipelining

Piplining is taking the output of a program and redirecting to be the input of a program.

```
calendar -A 30 -f /usr/share/calendar/calendar.holiday | grep "South Korea"
```

3 Embedded Commands

- We can use a subshell to use the output of commands as command line arguments to scripts.
- This is different from redirection. The output of the embedded command is run first and replaces the embedded command with the output.
- `egrep $(cat file) myfile.txt` allows us to run `egrep` with the contents of a file being the regular expression.
- Note the difference between:
  - `egrep $(cat file) myfile` — the contents of file is used as the first argument to egrep, i.e. the contents of file is being searched for
  - `egrep "cat file" myfile` — searches for the string cat file

4 Bash Variables

- In bash, a variable is assigned a value as follows: `var=42`. You do not need to declare a variable before assigning a value.

  Note: There cannot be spaces on either side of the equals symbol.
- All variables are stored as strings.
- Unlike C variables, bash variables persist outside of the scope of if statements, loops, and scripts.
- Accessing the value in a variable: `$var` or `${var}`.

- `${var%<end>}` removes the suffix `<end>` from the string stored in var. If `<end>` is not at the end of `var`, the string is unchanged.
- In addition to using variables as arguments, we can also treat the value of a variable as a command and run it:

  ```
  greet="echo hello"
  $greet
  ```

5 Bash Scripting

- A bash script is a series of commands saved in a file so that we can accomplish the same task without having to manually type all the commands.
- The first line of every shell script is the "shebang line" — `#!/bin/bash`. This line is telling the shell what program the file should be invoked with.
- To call a bash script, give the file executable permission using `chmod` and call the file by giving either an absolute path or a relative path to it.

  Note: if the relative path consists of only the file name (e.g. `script_name`), we need to add `./` before the path to call it: `./script_name`.
- Command line arguments are `$1`, `$2`, etc. The number of command line arguments is stored in `$#`.

5.1 Subroutines in Bash Scripts

- Format:

  ```
  subroutine() {
      ...
  }
  ```
- A subroutine is a series of commands which can be called at any time in a bash script.
- They can be given command line arguments the same way a program would be given command line arguments. A subroutine cannot access the command line arguments to the script. All other variables can be accessed.
- **Exercise:** Write a bash script which takes in two arguments, `ext1` and `ext2`. For each file (not directory) in the current directory which ends with an `.ext1`, rename the file to end with `.ext2`.

5.2 Debugging

- Debugging mode can be activated when running a bash script by placing `-x` at the end of the shebang line, or calling it using `bash -x`.
- When running the script, each command is printed to the screen with variables expanded.
- If a script is not doing what you expect it to do, using this debugging mode can be an easy way to see what is happening in the script.

6 Bash Loops and If Statements

- For the condition in both if statements and while loops, the result is checked, and if it's `true`, the program will go into the body of the if statement or while loop.
  - Form of an if statement in bash:

    ```
    if [ <cond1> ]; then
        ...
    elif [ <cond2> ]; then
        ...

    ......
    else
        ...
    fi
    ```

  - Form of a while loop in bash:

    ```
    while [ <cond> ]; do
        ...
    done
    ```

  - Form of a for loop in bash:

    ```
    for <var> in <words>; do
        ...
    done
    ```

  Where `<words>` is a list of whitespace separated strings. The body of the loop runs once for each string in `<words>`.

- Note: `[ <cond> ]` can be replaced by any command and the exit code will be checked. For example:

  ```
  # prints "foune" if grep succeeded
  if grep "hello" file.txt; then
      echo found
  fi
  ```

6.1 Test Command

- `test` is a bash command. The program is implicitly referred to using `[` (though it can also be explicitly referred to using `test`) and is called in the form `[ cond ]` whose exit code is 0 if `cond` is true and 1 if `cond` is false. It may be useful to review the `man` page for `test` (`man [` brings up the same page).
- A few conditions you can use for test:

`num1 -gt num2`	`num1` > `num2`
`str1 = str2`	`str1` == `str2` (string equality)
`-d path`	checks that `path` is a path a directory which exists

7 Program Exit Codes

- When a program completes, it always returns a status code to signify if the program was a success.
- This is true of any C program you have written before now. The exit code is the value returned from main, hence the contract `int main();`. In C and C++, if you do not explicitly return from main, the exit code is 0.
- In bash, if a program is successful, the exit code is 0. Otherwise, the exit code is non-zero. The exit code is stored in the variable `$?`.

  Remember: this is opposite from the definition of true in C. In C a non-zero integer represents true, while in bash zero represents success.
- The exit code cannot be larger than 255. In bash if you return some return code larger than 255, you will get the code modulo 256.

8 Extra Topics

The next subsections are review and will likely not be covered but may be useful to read.

8.1 Types of Quotes

- Note that does not affect the way `egrep` evaluates regular expressions.

8.1.1 Double Quotes

- Suppresses globbing, but allows variable substitutions and embedded commands:
  - `echo *` — prints names of all files in the current directory
  - `echo "*"` — prints `*`
  - `echo "$HOME"` — prints the absolute path to the user's home directory

8.1.2 Single Quotes

- No substitution or expansion will take place with anything inside of single quotes.
- Suppresses globbing, variable substitution, and embedded commands:
  - `echo '*'` — prints `*`
  - `echo '$(wc word.txt)'` — prints `$(wc word.txt)`

Both single and double quotes can be used to pass multiple words as one argument. This is useful for e.g. passing file names with spaces in them.

8.2 `egrep` and Regular Expressions

- Recall that `egrep` allows us to find lines that match patterns in files / standard input.
- Some useful regular expression operators are:

<table>
<tr><td><code>^</code></td><td>matches the beginning of the line</td></tr>
<tr><td><code>$</code></td><td>matches the end of the line</td></tr>
<tr><td><code>.</code></td><td>matches any single character</td></tr>
<tr><td><code>?</code></td><td>the preceding item can be matched 0 or 1 times</td></tr>
<tr><td><code>*</code></td><td>the preceding item can be matched 0 or more times</td></tr>
<tr><td><code>+</code></td><td>the preceding item can be matched 1 or more times</td></tr>
<tr><td><code>[...]</code></td><td>matches any <b>one</b> of the characters in the set</td></tr>
<tr><td><code>[^...]</code></td><td>matches any one character not in the set</td></tr>
<tr><td><code>\</code></td><td>the character after this will be regarded as a character not an operator.
i.e. <code>\.</code> matches the <code>.</code> character, instead of any single character.</td></tr>
<tr><td><code>expr1|expr2</code></td><td>matches <code>expr1</code> or <code>expr2</code></td></tr>
</table>

- Recall that concatenation is implicit.
- Parentheses can be used to group expressions.
- The option `-n` will print line numbers.
- Give a regular expression to find lines starting with ‘a’ or ending with ‘z’:
  - `^a|z$`
- Give a regular expression to find lines with more than one occurrence of the characters a,e,i,o,u:
  - We may try `[aeiou](.*[aeiou])+`
  - But `[aeiou].*[aeiou]` would also suffice. Why?
- `egrep` can be especially useful for finding occurrences of variable / type names in source files. To find all lines containing the name `count` in all files ending in `.cc`:
  - `egrep "count" *.cc`
- **Remember:** regular expressions **are not the same as** globbing patterns.

8.3 Bash Example

- Create a Bash script called `mean` that is invoked as follows:

  ```
  ./mean filename
  ```

  The argument `filename` is the name of a file containing a list of whitespace-separated numbers, from which the mean will be calculated.

9 Tips of the Week: Vim Basics

- You'll quickly notice that vim has a few basic modes. The one you are likely familiar with are the normal, insert, and command mode.
- If you get stuck and don't know what mode you are in, pressing `Esc` key a few times usually brings you back to normal mode.

9.1 Normal Mode

- In normal mode, most keys are hotkeys for various actions.
- For moving around:

  `C-f` (Ctrl + F) moves cursor one screen down.

  `C-b` (Ctrl + B) moves cursor one screen up.

  `w` moves cursor to the next word.

  `b` moves cursor to the previous word.

  `/` starts searching in the file. Enter the text to search and press `Enter` moves the cursor to the first match after cursor. To find the next match, press `n`.

- For editing text:

  `i` enters insert mode at the current position.

  `a` enters insert mode at the position after the current location.

  `o` creates a new line after the current line, and enter insert mode.

  `u` undoes last change.

9.2 Insert Mode

- This is the mode where you can write text. Anything you type will go into the file contents.
- Pressing `Esc` when you are in insert mode switches to normal mode.

9.3 Command Mode

- This is the mode that you enter by pressing `:` (colon) in normal mode.
- A colon will be shown on the bottom of the editor to indicate that you are in command mode.
- Similar to entering commands in a shell, you can use up / down arrow keys to go through the history, and press `Enter` to run a command.
- These are the most commonly used commands:

  `:q` closes vim if no changes have been made to the file.

  `:q!` closes vim without saving change which have been made to the file (since the last save).

  `:w` saves changes to the current file without quitting.

  `:wq` saves changes to the current file and closes vim.

  `:x` like `:wq`, but only save if changes have been made.