Program: Very Simple Shell - Phase a

CIS 310: Operating Systems

Brian C. Ladd

Fall 2019

Overview

Goal: Write a program that reads lines from standard input, taking each line apart into its words and listing the words, until the user halts the program.

Think about the shell program (`bash` on the lab machines)

1. It reads each line you type
2. It determines what command to execute by finding the words in the command line
3. It then **executes** the commands (not part of this program).

Think about how you would do this in `Java`

1. Hopefully you thought of `Scanner`.
   (a) You will not be using any C++ `Scanner` equivalent <sad face>
2. C++ input streams **do** support `getline`
   (a) Which can be used directly in a `while` loop to stop on EOF

Practice

Compiling C++

1. Using the standard `Makefile`
2. Using the conventional directory structure

Sentinel-controlled loop

C++ Stream I/O

1. insert, »
2. extract, «
3. `getline`

C++ `string` class

1. Take a string apart into words

`vssh-a` Very Simple Shell - phase A

Design Notes

The program executable should be named `vssh-a`

1. There will be two modules (and two `.cpp` files and one `.h` file): `executable` and `parser`
   (a) The source files you are writing: `src/executables/vssh-a.cpp`, `src/parser/shell.h`, and `src/parser/shell.cpp`
2. The name of the executable comes from the name of the `.cpp` file **and** the `module.mk` file in the executables directory.
   (a) Make sure `src/executable/module.mk` sets the Make variable `TOOLS` (`TOOLS` is the list of executables to build)
   (b) Make sure `src/allModule.mk` includes `executable` (the *relative* name of the directory) in the list of `MODULES`
      i. `MODULES` list is **space separated**

The work of the program will be done in the `Shell` class

1. `src/parser/shell.h` contains the declaration of `class Shell`
   (a) C++ does not enforce the name match; the grader **does**
   (b) A header file must be protected with macros to make sure it is only included once.
   (c) A C++ class header will **not** include the body of any class methods
   (d) **IMPORTANT** A C++ class declaration *must* end with a semicolon (;) AFTER the closing curly brace.
      i. Error messages generated for this error are sometimes opaque

(e) The header is in the `src` directory tree because it is an **internal** header

(f) It is only used **inside** the module

(g) External headers, to be used in other programs, are put in the `include` folder of the project.

2. `allModule.mk` must also include the name of the module directory, `parser` in the `MODULES` variable

   (a) `src/parser/module.mk` should set the `SRC` variable, **not** the one for tools given above

User Interaction

When run, the program must loop until the input stream ends or the user exits the program

1. The loop is to **prompt** the user for input, **read** a line from the user, and **process** the input line into words.

   prompt Prompt the user with the string "vssh-a$ ".

   - Put "vssh-a" into a constant or variable; it will change next phase

   read Read a whole line from the user into a `string` variable.

   - Use the `getline(istream, string)` function with `cin` and your variable

   process If the input is not the exit command, break the line into words and print out the words, one per line

   - Words are just whitespace separated groups of non-whitespace characters (no need to handle escaped characters or quotes)
   - Use the `stringstream` class (note that standard headers do **NOT** use our class naming convention; look up the header file name)
   - From the `stringstream`, `>>`, the *extractor operator*, behaves like `Scanner.next` in Java
   - Use the `vector` class to hold the `string` words extracted; `vector` is the C++ equivalent of `ArrayList` in Java
     - `vector::push_back` inserts at the tail
     - Your process method should return the `vector` of `string`
     - Another method should take a `vector` of `string` and print it.

Built-in commands

`exit` If the command is the word "exit" with arbitrary white space before and/or after, terminate the program

<EOF> If `getline` is given the end-of-file character, it will return false and the loop should terminate

Example

```
vssh-a$ alpha
word[0] = alpha

vssh-a$ one fish two fish
word[0] = one
word[1] = fish
word[2] = two
word[3] = fish

vssh-a$ ! 0098 golden pastry aspect?
word[0] = !
word[1] = 0098
word[2] = golden
word[3] = pastry
word[4] = aspect?

vssh-a$ exit
```

Documentation

Note that these requirements, repeated or not, apply to *all* programming assignments in CIS 310.

Do not forget the `README.org` or `README.txt` file

The `README` document goes in the root directory of the project (where the `Makefile` lives)

It is in plain text or Org mode formatting

It must contain (at least) the following:

Identification Block Much as described in the next section, the `README` must identify the programmer (with e-mail address) and the problem being solved. No ID block is the same as no `README`.

Problem Restatement Restate the problem being solved to make the project self-contained. Restating the problem is also good practice to check that you understand what you are supposed to do.

Testing Criteria You know by now that "it must be right, it compiles" is a silly statement. So, how do you know that you are done? You must document exactly how you tested your program with

Test Input Files or descriptions of what to give as input

Test Execution Commandlines and answers to prompts to execute your program with each set of test data.

Expected Output How to find the output and what the output is supposed to be. This should refer back to the input data and the assignment to establish that the expected output matches the problem being solved.

Compiling and Exectuitg Instructions Give clear *commandline specifications* for compiling and running your program. What folder should the user be in to run the commands? What tool(s) does the process require? What do the commandline arguments *mean*?

The `README` must accompany every program you turn in.

Do not forget ID blocks in each C++ file and `README`

Example header block for a Java/C++ file

Taken from Departmental Coding Standards

```
/**
 * Gargoyle draws a random ASCII art monster on standard output.
 *
 * Gargoyle has all static methods (and no constructor) including
 * main. It is run with a single integer on the command-line that
 * is used to randomize the monster that is generated.
 *
 * @author Jimmy A. Student
 * @email studeja199@potsdam.edu
 * @course CIS 203 Computer Science II
 * @assignment p004
 * @due 04/25/2018
 */
```

Function comments must document intent.

Why is this computation broken out into a function?

What does it do?

1. This is in the language of the *caller*.
   (a) A function is the **interface** between two levels of *abstraction*.
      i. The header documentation is written for the *higher level* of abstraction.
      ii. The code (and its included documentation) is for the lower level of abstraction.

What are the *parameters*?

1. Document expected range of values, checks done on parameters, etc.

What errors/exceptions can happen?

1. Document both what exceptions and what they mean (to the *caller*).

What *preconditions* must pertain for this function to perform correctly?

What *postconditions* will this function put in place when run?

Deliverables/Submission Method

Submission is through `git`

The project is to be developed in a project directory structured as in the `cppProjectTemplate`

Name the directories under `src/` according to the names of modules in the project

Make sure there is one executables module using the `src/executables/module.mk` file

Copy (recursively) the `src/module/` directory to start new modules as necessary

1. You need to make sure the names of all modules are in `src/allModule`.
2. The recursive copy gets the module version of `module.mk` into the directory

The base directory of the project is to be a `git` repository

Make sure it has an appropriate C++ (and your editor) `.gitignore` file.

Make sure to delete the `.git` folder (the whole history of the template) and run `git init` to create a new history

Log in to `GitTea` at https://cs-devel.potsdam.edu

Create a new, empty repository on `cs-devel`.

1. The name **must** begin with `p001`.
2. The name of the repository of *each* program you turn in will begin with `p###` where "###" is the number of the assignment.

After the `GitTea` will guide you to connect your *local* repository (where your solution lives) to the *remote* repository.

1. Notice that the instructions differ on when you create the new repository on `cs-devel`.

   *

Submit using git to push to git@cs-devel.potsdam.edu in the repo you made.