Program: Very Simple Shell - Phase b

Brian C. Ladd

Fall 2019

Contents

Overview

Goal: To turn vssh-a from a simple string processor (that extracts words from lines of text) into a shell.

A shell (like sh, bash, csh, etc.)

1. Prompts the user.
2. Keeps track of the current working directory (cwd).
   (a) The prompt often indicates the current working directory.
3. Reads input a line at a time.
   (a) Most (not ours) support some sort of line continuation for longer commands.
4. Extracts the words from the command-line
   (a) The first word is assumed to name a command (executable file) to run.
   (b) Remaining words are command-line parameters passed into the command
5. Search a **path** (list of standard paths) for the named command.
6. Execute the standard command, waiting until the command finishes, and looping back to the prompt.

Practice

Adding new features to an existing program

C/C++ process handling

1. `fork`
2. `wait`
3. `exec` (family)

Thinking about executing processes

NOTE Pull the newest template repository; a slight file rename to make it easier to use.

`vssh-b` Very Simple Shell - phase B

Using `git tag`

Start with `vssh-a`,

1. Before starting, tag the version:
   (a) In the base folder run the `git tag` command:
   ```
   git tag vssh-a
   ```
   (b) This will permit you, in future, to return to this state by checking out the tag
   ```
   git checkout vssh-a
   ```
   (c) You should append `--tags` to any push command so that your tags get copied when you turn stuff in.

Design Notes

The program executable should be named `vssh-b`

1. This means the corresponding source file should be named `vssh-b.cpp`.
2. The `vssh-b.cpp` source file should be in an **executables** folder.
3. The expected folder/file structure (you can have more source code) is
   ```
   ├── Makefile
   ├── README.{md,org,txt}
   ```

```
├── include/
└── src/
    ├── allModule.mk
    ├── executables/
    │     └── executable.mk (<== Notice the name change; get newest template!)
    └── parser/
        ├── module.mk
        ├── shell.h
        └── shell.cpp
```

There should be at least one additional module. In the file structure it is called `parser/`.

1. The module name need not match any files in it.
2. The `.mk` file is always named `module.mk` in a module directory
3. The files for the `Shell` class **should** be named `shell.h` and `shell.cpp`.

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 (or class member functions).
   (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. `src/parser/shell.cpp` contains the implementation of the `class Shell`
   (a) Make sure to include the `.h` interface file in the corresponding implementation file.
3. `src/allModule.mk`
   (a) Must include `executables` (relative name of folder) in the `EXECUTABLE_MODULES` variable

   (b) Must include the name of **all** module directories, (`parser` in our example) in the `MODULES` variable

4. The `Shell` class interface

   Shell the constructor. No obvious parameters.
   ~Shell the destructor. ALWAYS make destructors `virtual`. Ask in office hours if you want an explanation.
   run() the main loop for the program. Could be called with a prompt BUT the prompt this time reflects the cwd so the prompt string is probably not necessary.

User Interaction

When run, the program must loop until the input stream ends (type ^{D} to generate end-of-file) 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 (hopefully that sounds familiar).

   prompt Prompt the user with the name of the current working directory.

   - Use `std::filesystem::current_path`

   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 it is a built-in command, do the right thing; else search for executable.

   - Built-in commands
     <EOF> terminate the shell
     exit terminate the shell
     cd <nwd> <nwd> is the name of the new working directory. Use setting versions of `std::filesystem::current_path` to change it.
   - Search: if the first word on the line
     Begins with a slash ('/') check if the **exact file** named in the first word exists.
       - If it exists: fork, exec the named file with arguments
       - If it does not exist: report that the executable was not found. **Do NOT search the path.**

     Otherwise search for an exact name match for an executable file in each directory in the path **in order**.
       - Path directories . /usr/local/bin /usr/bin /bin
       - `std::filesystem` has features for checking the existence of a file, directory, or device.
       - `std::stat` is a function that gives detailed status information about a file path.

  - This includes whether or not it is marked “executable” and user access rights

(a) Note: searching the path.

   i. The following “trace” shows the search for executable programs # lines are comments on vssh-b; output of executed programs are elided

```
/home/laddbc program-in-home-directory
# /home/laddbc/program-in-home-directory is found (using filesystem).
# The path is passed to stat which returns the executable bit set.
# fork and exec
...

/home/laddbc ls
# find /home/laddbc/ls - FAIL
# find /usr/local/bin/ls - FAIL
# find /usr/bin/ls - FAIL
# find /bin/ls - SUCCEED; stat path - it is executable so run it
...

/home/laddbc sl
# find /home/laddbc/sl - FAIL
# find /usr/local/bin/sl - FAIL
# find /usr/bin/sl - FAIL
# find /bin/sl - FAIL

vssh-b: "sl" is an unknown command.
/home/laddbc
```

   ii. Notice that the last example has no …; no program was run.

Simulating the program you are writing

1. In the assignment folder is a script with some `bash` commands.
   (a) Script is named `set-path`
   (b) The script sets the path to exactly the four directories in the assignment
   (c) The script also sets the prompt to the current directory’s full path
2. Run bash with the following command line (with the relative path to the script) to simulate vssh-b:

```
~$ bash -ic "source set-path"
/home/laddbc
```

Things vssh-b does not need to do:

1. Handle background execution (as with the & character at the end of a command-line)
2. Handle file redirection (as with >, <, and forms of 2>)
3. Handle dynamically set search path.
4. Permit definition of aliases or sourcing of shell scripts

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 Executing 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 ore 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.