Assignment 3

Writing a Basic Command Shell

Author: Brian Crites

This project must be done in a group of two

Coding Requirements

The Test Command

In this assignment, you will add the `test` command to the rshell you have developed as well as its symbolic equivalent `[ ]`. The square brackets `[ ]` are actually interpreted as the `test` command in bash which you can read about here and try it out on your own. This command returns 0 (TRUE) if the test succeeds and 1 (FALSE) if the test fails. This command is very useful for writing conditions that can be combined with the `&&` and `||` connectors to write more complex bash command structures.

Note: You can not use execvp to run the test command as you did in Assignment 2. You must implement this functionality yourself and it should be compatible with all the functionality that you developed in Assignment 2.

Your subset of the test command is only responsible for a subset of file testing and should allow the user to run the command using the keyword test

```
$ test -e test/file/path
```

Additionally, your rshell should allow the user to use the symbolic version of the command

```
$ [ -e test/file/path ]
```

You command should also allow the user to run tests using the following flags

`-e`	checks if the file/directory exists
`-f`	checks if the file/directory exists and is a regular file
`-d`	checks if the file/directory exists and is a directory

If a user does not specify a flag, then the `-e` functionality should be used by default.

You will also add an extra feature that the current bash `test` command currently does not have. Your `test` command will need to print to the terminal if it evaluated to `True` or `False`

If the command `test -e /test/file/path` evaluates to `True`, then print display the following (including the parentheses)

```
(True)
```

And likewise, if the above command evaluates to `False`, then print `False` in the same manner

```
(False)
```

Additionally, your `test` command should work with all the connectors and other functionality that you have developed for Assignment 2.

```
$ test -e test/file/path && echo “path exists”
```

- or -

```
$ [ -e test/file/path ] && echo “path exists”
```

This will check if the file or directory `/test/file/path` exists and if path does exist will print “`path exists`”.

When your input requires you to have multiple outputs, simply print the `(True)` or `(False)` labels as they are evaluated, so the above command (assuming the path exists) would print

```
(True)
path exists
```

Your `test` function should be designed to work with both full directory paths and relative directory paths. In order to create this command, you will need to use the `stat()` function, which you can read about here along with the `S_ISDIR` and `S_ISREG` macros, which you can read more about here, in order to implement all the flags required for this program.

The Precedence Operators

Additionally, you will implement parentheses `( )` as precedence operators in your rshell. The parentheses operators are used to change the precedence of the execution of commands, connectors, and chains of connectors. For example

```
$ echo A && echo B || echo C && echo D
```

Would print the following

```
A
B
D
```

However, we can add parentheses to change the precedence of the connectors

```
$ (echo A && echo B) || (echo C && echo D)
```

which would print

```
A
B
```

This is because the parentheses change the precedence, so that when `(echo A && echo B)` returns true (since it executes correctly) to the `||` operator then it will not run the entire group `(echo C && echo D)` since the parentheses have now grouped these commands together. Note that if for some reason `echo A` failed, `echo B` would not execute, and `(echo C && echo D)` would be executed instead. If you are unsure about how the parentheses will change how a set set of commands and connectors run, or want to check how a specific set of command and connectors should execute, run the example on the hammer terminal for validation.

Note: While the above example only shows a single parenthesis, your program must account for multiple sets of parentheses as well as nested parentheses. If there are an uneven number of parentheses given as input you can either fail gracefully or ignore the input and prompt for a new input, printing some type of error message about the failure.

Project Structure

You must have a directory called `src/` which contains all the source (.cc/.cpp) files for the project. For header files you may either have a folder `header/` which contains your header files or you may keep them in the `src/` directory. You must have a `unit_tests/` directory which should contain all the unit tests you've written using the Google Unit Test framework (the main for the unit tests can be located either in `unit_tests/` or `src/`). You must also have an `integration_tests/` directory which should contain all the integration tests you've written using bash. We recommend using IO redirection for your bash integration tests, and automated validation is not necessary.

Your root directory must have a `CMakeLists.txt` with two targets. The first target is named `rshell` and should build the main executable and the second should be `test` which runs the unit tests that you have created using the Google Unit Test framework.

Note: The file/directory names above are a standard convention. You must use the exact same names in your project, including capitalization. We utilize these names when performing steps of our automated grading process, so any deviation may result in missing points.

The google test code for your unit tests can have any name as long as the files reside in the `unit_tests/` directory and an executable is generated named test in the root directory. The integration test shell scripts that you develop and place into the `integration_tests/` directory must have the following names:

`test_literal_tests.sh`	tests primarily for the test commands execution
`test_symbolic_tests.sh`	tests primarily for the symbolic tests commands execution
`precedence_tests.sh`	tests primarily for the precedence being respected

Note: the above tests should not only validate that the newly created functionality works correctly but also that it works with the features that were developed during Assignment 2.

Your project should not contain any files not necessary for the project. This includes things like CMake temporary build files used for building the project on your machine such as `CMakeCache.txt` and the `CMakeFiles/` directory as well as executables. We have provided a `.gitignore` in the template repository which will stop may of these files from showing as untracked when you run git status, and you should extend this file with additional temporary and machine specific files.

Submission Instructions

You will also need to add an **annotated** `hw3` tag to the commit that you want to be graded. Annotated tags are described in lab 2 and in the git documentation. `git push` will not automatically push tags to your repository. Use `git push origin hw3` to update your repository to include the `h3` tag. If you need to update your tag, you can remove the old tag using `git push --delete origin hw3` and push an updated one to your repo.

Note: We will test your project on the hammer server, so while it is not required that you develop on hammer you should verify that your project builds and functions properly on hammer.cs.ucr.edu.

Your project must also contain a names.txt file in the root directory which contains the name, SID, and email of each of the partners in your group. It should have the following format.

```
Brian Crites, 860XXXXXX, bcrit001@ucr.edu
Andrew Lvovsky, 860XXXXXX, alvov001@ucr.edu
```

Do not wait to push your assignment to Github until the project due date. You should be committing and uploading your assignment continuously. If you wait until the last day and can't figure out how to use git properly, then you will get a zero on the assignment or be forced to use the course late policy. No exceptions.

For late assignment submissions, a penalty of 10% will be deducted for every day the assignment is late up to a maximum of 3 days, with exceptions only for documented emergencies. For clarity, this means that an assignment that is 3 days late would incur a 30% penalty to the grade **assessed**. Please submit this form along with your late submission. Otherwise, the team will receive a zero on that assignment.

In addition to the above submission requirements you must update your `README.md` file so that it specifies the current structure of your program. This is especially important since your final program will likely be much different than what you originally designed in Assignment 1.

Testing Your Submission

Your assignment 3 should have approximately the following structure, see the project structure section for more details:

example-project-directory/

```
├── CMakeLists.txt
├── README.md
├── googletest
├── header
    ├── header_file_1.h
    └── header_file_2.h
├── src
    ├── src_file_1.cpp
    └── src_file_2.cpp
└── unit_tests
    ├── test_file_1.cpp
    └── test_file_2.cpp
└── integration_tests
    ├── test_command_tests.sh
    ├── test_symbolic_tests.sh
    └── precedence_tests.sh
```

You should also have a .gitignore and a .gitmodules hidden file in your root directory. Your root directory should contain no C++ header or source files and your googletest directory should be added to your project as a submodule. While the googletest directory is not required to be present in the root directory it is suggested.

When testing your project we will run (approximately) the following commands on hammer:

```
git clone <assignment-repo-url>
cd <assignment-repo-url>
git checkout tags/hw3
git submodule init
git submodule update
cmake3 .
make
test -e rshell || echo "rshell executable missing, check submission instruction
section of the specifications"
test -e test || echo "test executable missing, check submission instruction
section of the specifications"
test -d unit_tests || echo "unit_tests/ directory missing, check submission
instruction section of the specifications"
test -d integration_tests || echo "integration_tests/ directory missing, check
submission instruction section of the specifications"
```

These commands should generate a `test` executable as well as an `rshell` executable in the root directory, otherwise will print that they are missing. **Please run these commands on hammer with a brand new clone** of your GitHub repository to ensure your shell compiles and builds properly before submission.

Collaboration Policy

- You **may not** copy any code found online, doing so will result in failing the course.
- You **may not** look at the source code of any other student.
- You **may** discuss with other students in general terms how to use the unix functions.
- You are **encouraged** to talk with other students about test cases and are allowed to freely share ideas in this regard.

Grading

10	Sufficient Unit Test Cases
10	Sufficient Integration Test Cases
10	Updated README.md

15	Literal Test Command Execution
20	Symbolic Test Command Execution
35	Precedence Implementation
100	**Total**

Note: Your project structure and executable names are not explicitly listed in the grading schedule above but not following the specified file names and location may result in losing points because the automated grading system is unable to process your repository. Loss of points due to an issue specified above are final and failure to build and compile will result in a zero for the assignment.