CIT 595 Spring 2022: Project

Due Date: Wednesday April 27 at 11:59pm

NOTE, part of the assignment will be manually graded, and part of it will be autograded. Part C will be entirely manually graded.

For information on partners, see the appropriate section towards the bottom of the write-up.

Goals

- Practice C++, including objects and STL
- Implement network code utilizing the POSIX sockets API
- Write a multi-threaded server to receive HTTP requests from client, and respond to them

Contents

Overview & Instructions

In this assignment you will implement a multi-threaded Web server that provides simple searching and file viewing utilities. This assignment is broken up into three parts. In part A, you will finish implementing some code that will allow the server to read from files, and parse those files to record any words that show up in those files. In part B, you will implement parts of the web server to allow connections and handle Http requests. In Part C, you will combine what you did in part A and B to finish implementing a functioning web server.

We HIGHLY recommend you read through this entire document. At the very least, please read the overview associated with each part before you start working on it.

Part A

In Part A, you will implement four files: `FileReader.cc`, `WordIndex.cc`, `WordIndex.h`, and finally, `CrawlFileTree.cc`.

FileReader.cc

In this file, you will be implementing a simple file reader. The name of the file will be read at the time of construction, and the function read_file will read the entire contents of the file into a singular string. You may use POSIX, the C FILE interface, a stream from C++, or whatever you think works best to implement the reader.

WordIndex.cc and WordIndex.h

In the next part of the assignment, you need to implement WordIndex.h. In these files, you need to implement a data-structure that will allow us to record which documents contain a word and how many times each of those words are contained in each document. You will likely need to use STL containers to implement this, and it is up to you to decide which are most appropriate to use.

CrawlFileTree.cc

Most of the file has already been implemented for CrawlFileTree.cc, you just need to implement the last function at the bottom of the file called HandleFile. This function will take in a file name and a WordIndex. It is up to this function to read the specified file, and record each word that is found in it. This file is the core of our file processing that we will later use for search engine results for the server.

Part B

In Part B, you will implement three files: ServerSocket.cc, HttpConnection.cc and HttpUtils.cc.

ServerSocket.cc

This file contains a helpful class for creating a server-side listening socket, and accepting new connections from connecting clients. We've provided you with the class declaration in ServerSocket.h but you will need to implement it in ServerSocket.cc.

HttpConnection.cc

HttpConnection handles the reading in of an HTTP request over a network connection, parsing such requests into an object, and also the writing of responses back over the connection. This will largely deal with string manipulation to read and parse the HTTP requests.

HttpUtils.cc

HttpUtils provides some utilities that we will need for our search server. In particular, there are two functions that you will need to implement for making sure that our server handles some security concerns. You will only need to implement those two functions (escape_html and is_path_safe). You may still want to take a look at the other functions declared in HttpUtils.h since they will likely help you with implementing part C.

the function is_path_safe is used to make sure that anyone using the server can only access files under the specified static files directory. If we don't implement and use the function, then it is possible that an attacker could request any file on our computer that they would like with something called a directory traversal attack.

The other function escape_html is used to prevent a “cross-site scripting” flaw. See this for background if you're curious: http://en.wikipedia.org/wiki/Cross-site_scripting

Part C

In this part, you will take what you did in part A and part B to implement a web server. Most of the file is filled out for you, but there are a few places that you will need to implement. Particularly, you will need to finish handling the thread function, where each thread would handle a connection, and two helper functions to server the two types of requests the server may get, requests to see a file, and requests to process a search query.

Once you have them working, test your httpd binary to see if it works. Make sure you exercise both the web search functionality as well as the static file serving functionality. You can look at the source of pages that our solution binary serves and emulate that HTML, if you would like to get the same "look and feel" to your server as ours. However, as long as you mimic the same behaviour (have a search bar, process files and queries correctly, and show their results similarly), you are free to modify the look of your site. In the past, some students implemented "dark mode", had a Shrek theme, etc.

To run your completed httpd binary, try runnning the following command from the terminal: `./httpd 3000 ./test_tree/` after it prints "accepting connections…", hit the "Open server on port 3000" button at the top of codio. From here, you can test your server.

Suggested Approach

Below we have provided a suggested approach to this homework. Note that you are not required to follow this ordering if you believe another approach would work better for you. Also note that you can gradually check your progress, and run specific tests. Look at the Gtest section below for more details on running individual tests.

Also note that the only parts of the homeworks that have a direct order you need to follow is that the HttpServer last, and you need to implement FileReader and WordIndex before you implement CrawlFileTree.

1. Start by implementing FileReader::read_file and making sure you pass the provided tests.
2. Implement WordIndex.cc and WordIndex.h and then make sure you pass the word index tests.
3. Implement CrawlFileTree.cc handle_file function, making sure you pass the tests for it.
4. Implement all of ServerSocket.cc, making sure you pass the tests for it.
5. Implement get_request and parse_request from HttpConnection.cc, and then make sure you pass those tests.
6. Implement write_response in HttpConnection.cc and make sure you pass the tests for it
7. Implement both incomplete functions in HttpUtils.cc. At this point, you should have passed the entire test_suite
8. Debug any valgrind errors with the code tested by the test_suite
9. Implement HttpServer.cc and test it as described above.

Solution_binaries

With this assignment, we are providing a compiled solution that you can use to implement your search server. You can run the solution from the command line by running `./solution_binaries/httpd 3000 ./test_tree/`. For your implementation, you may want to compare your behaviour to it so that you can ensure you have a correct implementation.

Boost

In this homework, we have installed the external C++ library called boost. Boost is a decently used library and contains many useful functions. In particular, the string functions split(), replace_all() and trim() will likely be the most useful to you. You can take a look at the boost reference for strings here: https://www.boost.org/doc/libs/1_78_0/doc/html/string_algo.html but we also have recitation 11 dedicated to going over the project and getting you Practice with those boost functions.

Hints

- As mentioned above, the boost library is available to you in this homework, and it will make your life a lot easier with this project if you know how to use `replace_all`, `split`, and\or `trim` from boost.
- In particular for the function `split`, it is helpful that you are not required to use `is_any_of` as the predicate for splitting. You can also use functions like `isalpha` which will split on any alphabetic character, or write your own

function that takes in a character and returns a boolean, true iff that character is a delimiter
- We HIGHLY recommend you take inspiration from the provided lecture code `server_accept_rw_close` for your implementation of ServerSocket.cc
- There is a function that will make your implementation of `is_path_safe` a lot easier. Read the comment left for you in HttpUtils.cc and do a little digging online to see if you can find it.
- For FileReader, we need to handle binary files which may hold a 0 byte in it, as a result you may find the 2 argument string constructor to be useful here.

Grading & Testing

Compilation

We have supplied you with a `makefile` that can be used for compiling your code into an executable. To do this, open the terminal in codio (this can be done by selecting Tools -> Terminal) and then type in `make`.

You may need to resolve any compiler warnings and compiler errors that show up. Once all compiler errors have been resolved, if you `ls` in the terminal, you should be able to see an executable called `test_suite`. You can then run this by typing in `./test_suite` to see the evaluation of your code. **Note that the test_suite is not the only way we will be evaluating your code.** See below for more information.

Note that your submission will be partially evaluated on the number of compiler warnings. **You should eliminate ALL compiler warnings in your code**

Valgrind

We will also test your submission on whether there are any memory errors or memory leaks. We will be using valgrind to do this. To do this, you should try running:
`valgrind --leak-check=full ./test_suite`

If everything is correct, you should see the following towards the bottom of the output:

```
==1620== All heap blocks were freed -- no leaks are possible
==1620==
==1620== For counts of detected and suppressed errors, rerun with: -v
==1620== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

If you do not see something similar to the above in your output, valgrind will have printed out details about where the errors and memory leaks occurred.

Note that you should avoid memory errors in your HttpServer.cc but there isn't a good way to test this since we can't make the server exit gracefully. As a result, you do not need to test valgrind on the `./httpd` binary.

Gtest

As with previous homework assignments, you can compile the your implementation by using the `make` command. This will result in several output files, including an executable called `test_suite`.

After compiling your solution with `make`, You can run all of the tests for the homwork by invoking:

```
./test_suite
```

You can also run only specific tests by passing command line arguments into `test_suite`

For example, to only run the HttpConnection tests, you can type in:

```
./test_suite --gtest_filter=Test_HttpConnection.*
```

If you only want to test write_response from HttpConnection, you can type in:

```
./test_suite --gtest_filter=Test_HttpConnection.write_response
```

You can specify which tests are run for any of the tests in the assignment. You just need to know the names of the tests, and you can do this by running:

```
./test_suite --gtest_list_tests
```

These settings can be helpful for debugging specific parts of the assignment, especially since `test_suite` can be run with these settings through `valgrind` and `gdb`!

Partners

Note that the assignment is designed to be completable by one person, but you may work in pairs. **Regardless of if you work in pairs, you MUST fill out the form here to indicate your partner status and how to locate your project grade** https://forms.gle/EpPTTNNEZSbwhMZS9. Additionally, it is up to you to figure out how you want to share code. Setting up a private git repo is generally a good idea, but other ways of file sharing will suffice for this assignment.

Submission:

This is the autograder for the entire assignment. We will be checking for compilation warnings, valgrind errors, and making sure the `test_suite` runs correctly. We will also be checking for use of illegal code.

Note: When you are done with the entire assignment, and ready to submit you must run the autograder below by selecting “Check It!“. You should then verify that you got the points you think you should get, and then at the bottom, you should select the button to mark the assignment as completed.

Note: The test_suite is not the only way we will be evaluating correctness of your code. We will also be checking that your implementation of HttpServer.cc is done correctly by running your httpd program and comparing it to the solution binary. Your html and site doesn't have to look exactly the same, but you should handle queries the same way, print out the correct results in similar format, and load files properly, just as the solution binary does.

{Check It!|assessment}(test-3063119557)