Due before:	11:59 PM (before Midnight) on Tue 23 2018

CSE-381: Systems 2

Homework #5

Due: Tuesday October 23 2018 before 11:59 PM

Due to travel: No office hours on Tue Oct 23 2018

Email-based help Cutoff: 5:00 PM on Sun, Oct 21 2018

Maximum Points: 45

Submission Instructions

This homework assignment must be turned-in electronically via Canvas CODE plug-in. Ensure your C++ source code is named `MUID_hw5.cpp`, where `MUID` is your Miami University Unique ID. Ensure your program compiles without any warnings or style violations. Ensure you thoroughly test operations of your program as indicated. Once you have tested your implementation, upload the following onto Canvas:

1. The 1 C++ source file developed for this homework.

General Note: Upload each file associated with homework individually. Do not upload archive file formats such as zip/tar/gz/7zip/rar etc.

Objective

The objective of this homework is to:

- Understand the use of HTTP protocol for communication
- Develop a C++ program that can process HTTP GET requests
- Develop program to run a given command and process its outputs
- Continue to gain familiarity with I/O streams & string processing

Grading Rubric:

The program submitted for this homework **must pass necessary base case test(s) in order to qualify for earning any score at all**. Programs that do not meet base case requirements will be assigned zero score!
Program that do not compile, have a method longer than 25 lines, or just some skeleton code will be assigned zero score.

- See points set for each command and overall structure, organization, **conciseness**, variable-names, etc. in the next page. **If your methods are not concise points will be deducted**.
- **-1 Points**: for each warning generated by the compiler (warnings are most likely sources of errors in C++ programs)

NOTE: Violating CSE programming style guidelines is a compiler error! Your program should not have any style violations.

Background

A web-server is an application layer program that processes HTTP requests and generates responses. Recollect that HTTP is a multi-line text protocol that is used by web-browsers to interact with web-servers. In this homework you will be developing a program to respond to HTTP requests. **Overall this program is essentially just string processing from I/O streams with most of the required code copy-pasted from different exercises/lecture slides.**

Test files

The following files are supplied for this homework:

1. You are supplied with several image, html, and text files for testing.
2. For functional testing, test files are in `test_files` directory. These are the same input and output files used by the CODE plug-in for checking.

Chunked Response Requirements [Review from lecture slides]

The response/output from the program/server should be in the following format, with each line terminated with a `"\r\n"`:

1. The first line should be `HTTP/1.1 200 OK` or `HTTP/1.1 404 Not Found` (if a specified file was not found). See example outputs
2. The second line should be `Content-Type: MIMEtype`, where `MIMEtype` is determined based on file extension as discussed further below. By default the `MIMEtype` is `text/plain` as in: `Content-Type: text/plain`
3. The next line must be `Transfer-Encoding: chunked` (a fixed constant string)
4. The next line must be `X-Client-Header-Count: num`, where *num* is the number HTTP headers sent by the client.
5. The next line must be `Connection: Close` (a fixed constant string)
6. The header section must be terminated by `"\r\n"`

Rest of the contents of a file or output of a program are printed line-by-line with line size (in hex on a separate line) followed by the data as shown below:

```
// Send chunk size to client.
os << std::hex << line.size() << "\r\n";
// Write the actual data for the line.
os << line << "\r\n";
```

The end of data must be sent to the client via the following trailer:

```
// Send trailer out to denote end of data
os << "0\r\n\r\n";
```

Request Processing Requirements

Your program should process multiple lines of an HTTP GET request (read via `std::cin`) in the following manner:

1. Base case -- Return contents of a file or error [**15 points**]: If the first line of the HTTP request does not start with `GET /cgi-bin/exec`, then assume that the GET request is in the format: `GET /file HTTP/1.1`, where **`file`** corresponds to the path of the file whose contents is to be returned back to the user using chunked transfer as discussed below. The **`file`** should be processed as:

   a. If the **`file`** is simply / the server should respond with contents of given `index.html`

   b. If the **`file`** path is valid (that is file is readable, see `ifstream's good()` method), the server should respond with the **`file`**'s contents and appropriate HTTP headers & MIME type for the **`file`** deduced from the **`file`** name extension. The extension and MIME types are determined (and set in `Content-Type` output/response header) as tabulated in the table below:

Extension	MIME type
`.html`	`text/html`
`.png`	`image/png`
`.jpg`	`image/jpeg`
`.txt`	`text/plain`
Default (*i.e.*, in all other cases)	`text/plain`

   The contents of the file should be sent to the client line-by-line using chunked encoding as discussed earlier, along with necessary headers. See following sample input and output files in `test_files` directory for details: `file_test_inputs.txt`, `file_test_expected_outputs.txt`, `img_test_inputs.txt`, `img_test_expected_outputs.txt`, `html_test_inputs.txt`, and `html_test_expected_outputs.txt`.

> **!** The base case is relatively straightforward. Consequently, if the base case does not operate as expected, as per the course policy, the program will be assigned zero score.

2. **Unreadable file [7 points]**: If the **`file`** is not readable (i.e. `std::ifstream::good()` method returns `false`) the server should respond with a suitable HTTP 404 response/output. For example, see example output shown in `test_files/err_test_expected_outputs.txt`.

3. Run program & return output [**23 points**]: If input line starts with "`GET /cgi-bin/exec`" then this line corresponds to the case where a program is to be run. In this case, this line will be in the format: `GET /cgi-bin/exec?cmd=ls&args=1.txt "hello.cpp" HTTP/1.1` (one line), where `cmd` contains the command to run and `args` has list of space separated command-line arguments to the program. Use `std::quoted` to correctly handle double quoted words in `args`. Note that you need to `url_decode` `cmd` and `args` prior to using them as they will be encoded (for an example see request in `test_files/ cgi_bin_test_inputs1.txt`). URL decoding method is copy-paste from lecture slides.

   Appropriately process the input, run the specified command, obtain its outputs using pipes (see slide #19, #20 in `07_IPC.pdf`, you will need to suitably use simple pipes example but run 1 process and read its outputs using a pipe) and route the outputs back to the client using chunked encoding as discussed earlier.

Functional testing

Several test inputs and expected output files are also supplied to streamline your testing as shown below:

```
$ ./hw5 < test_files/base_case1_inputs.txt my_base_case1_output.txt
$ diff my_base_case1_output.txt test_files/base_case1_expected_outputs.txt
```

Note: If your solution is correct then the above `diff` command will not print any output.

Ensure you test with all input files and corresponding expected outputs to check all features of your program.

Submit to Canvas

This homework assignment must be turned-in electronically via Canvas CODE plug-in. Ensure your C++ source files are named appropriately. Ensure your program compiles (without any warnings or style errors) successfully. Ensure you have tested operations of your program as indicated. Once you have tested your implementation, upload the following onto Canvas:

- The 1 C++ source file you developed for this homework

Upload the C++ source files to onto Canvas. Do not submit zip/7zip/tar/gzip files. Upload each file independently.