Url Parser Project

URL's are used to represent resource locations on the internet. Understanding URL's is one of the fundamental requirements of any program that interacts with online resources, including web browsers and mobile applications. For this project, we will build a set of objects in C++ that can help us disect a url string into it's different parts. We will be working with a slightly simplified version of valid *relative* URL's.

Parts of a URL

There are many types of URL's. We will be working with a subset most commonly used in online applications. Here are a few example URL's:

- http://example.com/path/to/file.txt?test=true&value=three#fragment-string
- file:///local/path/on/machine/file.cpp
- https://www.example.com/?test=true
- /new/path/file2.c

In general, a URL can be broken down into the following pieces:

<scheme>://<location>/<path>?<query>#<fragment>

The <scheme> consists of anything preceeding the first : character.

The <location> consists of everything between the first two / characters, and the third / character.

The <path> is everything between the third / character and before the ? character.

The <query> is anything following the ? character and preceeding the # character.

The <fragment> is anything following the # character.

Disecting a URL

The URL parser should be able to take a url provided as a text string, and extract each of the component parts from it. For instance, given the url: https://example.com/file.txt#para1, the parser should identify:

- <scheme> = https
- <location> = example.com
- <path> = file.txt
- <fragment> = para1

Note that the <query> is missing in this url. The URL parser should also be able to print the full url by correctly combining all of the components.

In this project, the URL parser is implemented in class URL. This class uses a number of helper objects (extractors) to disect the url components.

Project Requirements

The starting code for this project can be found here. You may not remove any classes or methods from the current interface (header files). You may augment the interface with your own classes and methods as needed in order to help complete the project, but this is not required.

A main function is provided to indicate how the program should be used. You are free to modify this function as needed, however the expectation is that the initial main can be run on your final submission and still work correctly. I will be using the initial main function provided in order to evaluate the final code.

The initial code will not successfully build because many of the interface methods are missing definitions. You are to implement the missing definitions and fulfill the following requirements:

Implementation Steps

1. Define the missing methods and source files so that the code compiles
    - Will need to create new *cpp* files for the extractors (ex: FragmentExtractor.cpp)
    - Will need to implement missing methods in these classes as well as Url.cpp
2. Implement FragmentExtractor: The fragment extractor is given a string. It should find the # character if it exists. If it does, everything to the right of the # character becomes the fragment, and everything to the left becomes the base.
    - For each extractor that is implemented, the other public methods must also be implemented. In this case, HasComponent, GetBase, and GetComponent are to be implemented.
3. Implement SchemeExtractor: The scheme extractor is given a string. It checks whether the string begins with characters followed by the :// pattern. If so, everything to the left of the : is saved as the component, and everything to the right of the : including the // is saved as the base.
    - For the scheme extractor, the scheme type must also be determined. If the scheme matches http or https (ignoring case) then the type is Net. If the scheme matches file, then the type is Local. Otherwise, the scheme type is Uknown.
4. Implement LocationExtractor. If the string starts with two slashes //, then it has a location (hasLocation = true). The location consists of all the characters between the first two slashes // and the third slash / if there is one. It is possible to have an empty location (ex: file:///path) and it is possible for a string to have no ending slash (ex: http://www.example.com, location = www.example.com). The location component does not include any of the slashes, just the characters between them. The first two slashes are dropped. The base should hold the rest of the string that follows the location (including the third slash if there is one).
5. Implement QueryExtractor. Search the for a ? character. If found then the query component is everything following that character. Remove the ? character and everything remaining (everything before it) becomes the base. It is possible to have an empty query string (ex: http://example.com/path?). In this case, the query still exists, but the value of the component is an empty string.
    - Once you have the query, you can parse the query parameters. Inside of the query component, the query parameters are delimeted by a & character. A query (if it exists and is not an empty string) will have one or more parameters. For example the query string (www.example.com/path?this=true&that=false) has two parameters: this=true, followed by that=false. The & characters are not part of the parameters, so my query parameters vector should have two strings, one for each parameter noted above.
6. Impelement PathExtractor. Everything remaining is the path. Either nothing remains (path does not exist) or something remains and it is considered the path. If the path exists and begins with a / character,

then the path is absolute. If so, remove the / character and save what remains as the component.
    - The path extractor also has a vector of strings to hold the path subcomponents. Much like we did for query parameters, the path subcomponents are delimted by / characters. Separate out and store each one as an individual element in the _pathComponents vector. For example, if our url is example.com/path/to/file.txt?query, then the path subcomponents are path, to, and file.txt. Notice that the / characters are not part of the subcomponents.
7. Finish implementing the constructor for Url. By composing the extractors in the correct order (specified in the file) you will be able to correctly parse the url and break it down into its different pieces. In addition, if the url has a scheme and not a location, then it is invalid. In this case, throw a UrlFormatException.
8. Implement operator<< in the Url class as well. Use the extractors to recompose and print the full url. This allows us to print the url object directly to cout, which we do in the main function. This url should match the url passed into the program if done correctly.

Grading Breakdown (100 points)

- **(10)** Nothing is taken away from the initial interface
- **(10)** Program compiles and all methods are defined
- **(10)** FragmentExtractor is implemented and correctly parsing
- **(10)** SchemeExtractor is implemented and correctly parsing
- **(5)** Scheme type is correctly saved
- **(10)** LocationExtractor is implemented and correctly parsing
- **(10)** QueryExtractor is implemented and correctly parsing
- **(5)** Query parameters are stored
- **(10)** PathExtractor is implemented and correctly parsing
- **(5)** Path subcomponents are stored
- **(5)** An exception is thrown for an invalid url
- **(10)** The url object can be printed directly to cout.

Example

Running your program from the command line should produce output similar to this:

```
S:\...\Debug>UrlResolver.exe https://www.example.com/some/path/file.txt?
test=true#fragment
SCHEME: https
NETLOC: www.example.com
PATH: some/path/file.txt
QUERY: test=true
FRAG: fragment
URL: https://www.example.com/some/path/file.txt?test=true#fragment
```