CSE 333
Section 9

Boost & Hypertext Transfer Protocol

UNIVERSITY *of* WASHINGTON

Logistics

Friday, Mar 12:

HW4 @ 12pm

BOOOOOOOOST

BOOST (HW4)

Boost is a free C++ library that provides support for various tasks in C++

- **Note:** Boost does NOT follow the Google style guide!!!

Boost adds many string algorithms that you may have seen in Java

- Include with `#include <boost/algorithm/string.hpp>`

We are showcasing a few we think could be useful for HW4, but more can be found here:

- https://www.boost.org/doc/libs/1_60_0/doc/html/string_algo.html

Helpful BOOST Functions

```
void boost::trim(string& input);
```

- Removes all leading and trailing whitespace from the string
- `input` is an input *and* output parameter (non-const reference)

```
string s("     HI     ");
boost::algorithm::trim(s);
// results in s == "HI"
```

More Helpful BOOST Functions

```
void boost::replace_all(string& input, const string& search, const
string& format);
```

- Replaces all instances of `search` inside `input` with `format`
- `replace_all()` guarantees that '`format`' will be in the final result if-and-only-if '`search`' existed.`replace_all()` makes a singlepass over `input`

```
string s("queue?");
boost::algorithm::replace_all(s, "que", "q");
// results in s == "que?"
```

More Helpful BOOST Functions

```
void boost::split(vector<string>& output,
                  const string& input,
                  boost::PredicateT match_on,
                  boost::token_compress_mode_type compress);
```

- Split the string by the characters in `match_on`

```
boost::PredicateT boost::is_any_of(const string& tokens);
```

- Returns predicate that matches on any of the characters in tokens

split Example

```
vector<string> tokens;
string s("I-am--split");

boost::split(tokens, s, boost::is_any_of("-"),boost::token_compress_on);
// results in tokens == ["I", "am", "split"]

boost::split(tokens, s, boost::is_any_of("-"),boost::token_compress_off);
// results in tokens == ["I", "am", "", "split"]
```

Exercise 1

Write a function that takes in a string that contains words separated by whitespace and returns a vector that contains all of the words in that string, in the same order as they show up, but with no duplicates. Ignore all leading and trailing whitespace in the input string.

Example:

`RemoveDuplicates(" Hi I'm sorry jon sorry hi hihi hi hi ")` should return the vector `["Hi", "I'm", "sorry", "jon", "hi", "hihi"]`

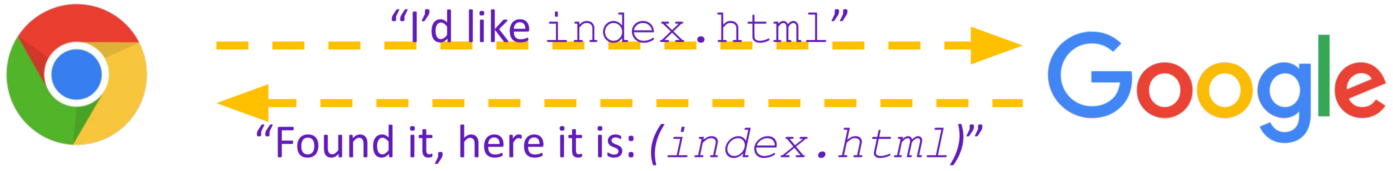
```
vector<string> RemoveDuplicates(const string& input);
```

Exercise 1 Solution

```
vector<string> RemoveDuplicates(const string& input){
      string copy(input);
      boost::algorithm::trim(copy);
      std::vector<string> components;
      boost::split(components, copy, boost::is_any_of(" \t\n"), boost::token_compress_on);
      std::vector<string> result;
      for (uint i = 0; i < components.size(); ++i) {
            bool unique = true;
            for (uint j = 0; j < i && unique; ++j) {
                  unique &= !(components[i] == components[j]);
            }
            if (unique) {
                  result.push_back(components[i]);
            }
      }
      return result
}
```

Hypertext Transfer Protocol

HTTP Basics



- A client establishes one or more TCP connections to a server
  - The client sends a request for a web object over a connection and the server replies with the object's contents

- We have to figure out how to let the client and server communicate their intentions to each other clearly
  - We have to define a *protocol*

Protocols

- A protocol is a set of rules governing the format and exchange of messages in a computing system
  - What messages can a client exchange with a server?
    - What is the syntax of a message?
    - What do the messages mean?
    - What are legal replies to a message?
  - What sequence of messages are legal?
    - How are errors conveyed?

- A protocol is (roughly) the network equivalent of an API

HTTP

- Hypertext Transfer Protocol
  - A request / response protocol
    - A client (web browser) sends a request to a web server
    - The server processes the request and sends a response
  - Typically, a **request** asks a server to retrieve a resource
    - A *resource* is an object or document, named by a Uniform Resource Identifier (**URI**)
  - A **response** indicates whether or not the server succeeded
    - If so, it provides the content of the requested response
  - Wikipedia: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

HTTP Requests

- General form:

  -
    ```
    [METHOD] [request-uri] HTTP/[version]\r\n
    [headerfield1]: [fieldvalue1]\r\n
    [headerfield2]: [fieldvalue2]\r\n
    [...]
    [headerfieldN]: [fieldvalueN]\r\n
    \r\n
    [request body, if any]
    ```

- ~~Demo: use nc to see a real request~~

HTTP Methods

- There are three commonly-used HTTP methods:
  - `GET`: "please send me the named resource"
  - `POST`: "I'd like to submit data to you" (*e.g.* file upload)
  - `HEAD`: "Send me the headers for the named resource"
    - Doesn't send resource; often to check if cached copy is still valid

- Other methods exist, but are much less common:
  - `PUT`, `DELETE`, `TRACE`, `OPTIONS`, `CONNECT`, `PATCH`, `. . .`
    - For instance: `TRACE` – "show any proxies or caches in between me and the server"

HTTP Uniform Resource Identifier (URI)

- Absolute URI
  - Composition: scheme:[//authority]path[?query]
  - Mainly used for communicating via proxy
- Most common form of Request-URI
  - Composition: path[?query]
  - Host is specified through headers
  - Query is optional
  - Path can be empty (just /)
- Example Request-URI:
  - /static/test_tree/books/artofwar.txt?terms=hello

HTTP Versions

- All current browsers and servers “speak” HTTP/1.1
  - Version 1.1 of the HTTP protocol
    - https://www.w3.org/Protocols/rfc2616/rfc2616.html
  - Standardized in 1997 and meant to fix shortcomings of HTTP/1.0
    - Better performance, richer caching features, better support for multihomed servers, and much more

- HTTP/2 standardized recently (published in 2015)
  - Allows for higher performance but doesn’t change the basic web request/response model
  - Will coexist with HTTP/1.1 for a long time

Client Headers

- The client can provide zero or more request “headers”
  - These provide information to the server or modify how the server should process the request

- You’ll encounter many in practice
  - `Host:` the DNS name of the server
  - `User-Agent:` an identifying string naming the browser
  - `Accept:` the content types the client prefers or can accept
  - `Cookie:` an HTTP cookie previously set by the server
  - https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html

A Real Request

```
GET / HTTP/1.1
Host: attu.cs.washington.edu:3333
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/66.0.3359.181 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
image/apng,*/*;q=0.8
DNT: 1
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: SESS0c8e598bbe17200b27e1d0a18f9a42bb=5c18d7ed6d369d56b69a1c0aa441d7 8f;
SESSd47cbe79be51e625cab059451de75072=d137dbe7bbe1e90149797dcd89c639b1;
...
```

GET / HTTP/1.1

- HTTP method
- URI
- Version

HTTP Responses

- General form:

  -
    ```
    HTTP/[version] [status code] [reason]\r\n
    [headerfield1]: [fieldvalue1]\r\n
    [headerfield2]: [fieldvalue2]\r\n
    [...]
    [headerfieldN]: [fieldvalueN]\r\n
    \r\n
    [response body, if any]
    ```

- Demo: use telnet to see a real response
  - telnet www.google.com 80
  - telnet attu4.cs.washington.edu 5555 (if running hw4 server)
    - ./solution_binaries/http333d 5555 ../projdocs unit_test_indices/*
  - GET / HTTP/1.1
    - hit double enter, ctrl +] -> quit
    - more requests next page

Writing an HTTP Request

- Example HTTP Request layout can be found in HttpRequest.h
- Example file request:
  - GET /static/test_tree/books/artofwar.txt HTTP/1.1
- Example query request:
  - GET /query?terms=books+of+war HTTP/1.1
- To send a request, hit [Enter] twice
- Compare the output of solution_binaries/http3d to ./http3d

Status Codes and Reason

- *Code*: numeric outcome of the request – easy for computers to interpret
  - A 3-digit integer with the 1st digit indicating a response category
    - `1xx`: Informational message
    - `2xx`: Success
    - `3xx`: Redirect to a different URL
    - `4xx`: Error in the client's request
    - `5xx`: Error experienced by the server

- *Reason*: human-readable explanation
  - *e.g.* "OK" or "Moved Temporarily"

Common Statuses

- `HTTP/1.1 200 OK`
    - The request succeeded and the requested object is sent
- `HTTP/1.1 404 Not Found`
    - The requested object was not found
- `HTTP/1.1 301 Moved Permanently`
    - The object exists, but its name has changed
        - The new URL is given as the "`Location:`" header value
- `HTTP/1.1 500 Server Error`
    - The server had some kind of unexpected error

Server Headers

- The server can provide zero or more response “headers”
  - These provide information to the client or modify how the client should process the response

- You’ll encounter many in practice
  - `Server:` a string identifying the server software
  - `Content-Type:` the type of the requested object
  - `Content-Length:` size of requested object
  - `Last-Modified:` a date indicating the last time the request object was modified
  - https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html

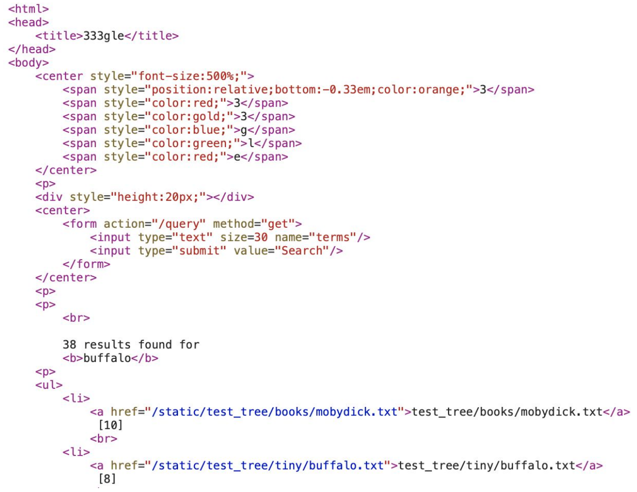
A Real Response

```
HTTP/1.1 200 OK
Date: Mon, 21 May 2018 07:58:46 GMT
Server: Apache/2.2.32 (Unix) mod_ssl/2.2.32 OpenSSL/1.0.1e-fips
mod_pubcookie/3.3.4a mod_uwa/3.2.1 Phusion_Passenger/3.0.11
Last-Modified: Mon, 21 May 2018 07:58:05 GMT
ETag: "2299e1ef-52-56cb2a9615625"
Accept-Ranges: bytes
Content-Length: 82
Vary: Accept-Encoding,User-Agent
Connection: close
Content-Type: text/html
Set-Cookie:
bbbbbbbbbbbbbbb=DBMLFDMJCGAOILMBPIIAAIFLGBAKOJNNMCJIKKBKCDMDEJHMPONHCILPIBLADEAKCI
ABMEEPAOPMMKAOLHOKJMIGMIDKIHNCANAPHMFMBLBABPFENPDANJAPIBOIOOOD; HttpOnly

<html><body>
<font color="chartreuse" size="18pt">Awesome!!</font>
</body></html>
```

hw4 Demo

- ./solution_binaries/http333d 5555 ../projdocs unit_test_indices/*
- http://attu4.cs.washington.edu:5555/
- Inspect element/page source
- Alternative to telnet
- Compare your server to the solution!



```
<html>
<head>
    <title>333gle</title>
</head>
<body>
    <center style="font-size:500%;">
        <span style="position:relative;bottom:-0.33em;color:orange;">3</span>
        <span style="color:red;">3</span>
        <span style="color:gold;">3</span>
        <span style="color:blue;">g</span>
        <span style="color:green;">l</span>
        <span style="color:red;">e</span>
    </center>
    <p>
    <div style="height:20px;"></div>
    <center>
        <form action="/query" method="get">
            <input type="text" size=30 name="terms"/>
            <input type="submit" value="Search"/>
        </form>
    </center>
    <p>
    <p>
        <br>

        38 results found for
        <b>buffalo</b>
    <p>
    <ul>
        <li>
            <a href="/static/test_tree/books/mobydick.txt">test_tree/books/mobydick.txt</a>
             [10]
            <br>
        <li>
            <a href="/static/test_tree/tiny/buffalo.txt">test_tree/tiny/buffalo.txt</a>
             [8]
```

Exercise 4

Write a function called `ExtractRequestLine` that takes in a well-formatted HTTP request as a `string` and returns a `map` with the keys as `method`, `uri`, `version` and the values from the corresponding request. For example,

Example Input:

```
"GET /index.html HTTP/1.1\r\nHost: www.mywebsite.com\r\nConnection:
keep-alive\r\nUpgrade-Insecure-Requests: 1\r\n\r\n"
```

Map Returned:

```
{
 "method" : "GET"
 "uri" : "/index.html"
 "version" : "HTTP/1.1"
}
```

Exercise 4 Solution

```
map<string,string> ExtractRequestLine(const string& request) {

 vector<string> lines;
 boost::split(lines, request, boost::is_any_of("\r\n"), boost::token_compress_on);

 vector<string> components;
 string firstLine = lines[0];
 boost::split(components, firstLine, boost::is_any_of(" "), boost::token_compress_on);

 map<string, string> res;
 res["method"] = components[0];
 res["uri"] = components[1];
 res["version"] = components[2];
 return res;
}
```

Good luck!!! :-)

Cool HTTP/1.1 Features

- “Chunked Transfer-Encoding”
  - A server might not know how big a response object is
    - *e.g.* dynamically-generated content in response to a query or other user input
  - How do you send Content-Length?
    - Could wait until you’ve finished generating the response, but that’s not great in terms of *latency* – we want to start sending the response right away
  - Chunked message body: response is a series of chunks

Cool HTTP/1.1 Features

- Persistent connections
  - Establishing a TCP connection is costly
    - Multiple network round trips to set up the TCP connection
    - TCP has a feature called “slow start”; slowly grows the rate at which a TCP connection transmits to avoid overwhelming networks
  - A web page consists of multiple objects and a client probably visits several pages on the same server
    - <u>Bad idea</u>: separate TCP connection for each object
    - <u>Better idea</u>: single TCP connection, multiple requests

20 years later...

- World has changed since HTTP/1.1 was adopted
  - Web pages were a few hundred KB with a few dozen objects on each page, now several MB each with hundreds of objects (JS, graphics, …) & multiple domains per page
  - Much larger ecosystem of devices (phones especially)
  - Many hacks used to make HTTP/1.1 performance tolerable
    - Multiple TCP sockets from browser to server
    - Caching tricks; JS/CSS ordering and loading tricks; cookie hacks
    - Compression/image optimizations; splitting/sharding requests
    - etc., etc. …

HTTP/2

- Based on Google SPDY; standardized in 2015
    - Binary protocol - easier parsing by machines (harder for humans); sizes in headers, not discovered as requests are processed; …
        - But same core request/response model (GET, POST, OK, …)
    - Multiple data steams multiplexed on single TCP connections
    - Header compression, server push, object priorities, more…
- All existing implementations incorporate TLS encryption (https)
- Supported by all major browsers and servers since ~2015
- Widely used now by all major web sites
    - Coexists with HTTP/1.1
    - HTTP/2 used automatically when browser and server both support it

Peer Instruction Question

- Are the following statements True or False?

	Q1	Q2
A.	False	False
B.	False	True
C.	True	False
D.	True	True
E.	We're lost...	

Q1: A protocol only defines the "vocabulary" that clients and servers can communicate with.

Q2: Clients and servers use the same header fields.

Peer Instruction Question

- Which HTTP status code family do you think the following Reasons belong to?

	Q1	Q2
A.	4xx	2xx
B.	4xx	3xx
C.	5xx	2xx
D.	5xx	3xx
E.	We're lost…	

Q1: Gateway Time-out

Q2: No Content