**1. Purpose**

This assignment builds an understanding of text-based application-layer protocols, multi-threading, system APIs, and Windows sockets.

**2. Problem Description**

Using Visual C++, your goal is to create a simple web client that accepts URLs and then crawls them to display basic server/page statistics.

**2.1. Code (25 pts)**

Your program must accept a single command-line argument with a target URL. If the argument is missing or there are too many of them, print usage information and quit. You should be able to handle URLs that fall under this general format

```
scheme://host[:port]/[path][?query][#fragment]
```

where the only acceptable scheme in this homework is “http”. Examples (IRL servers require TAMU VPN):

```
hw1.exe http://tamu.edu                    [no path, no port, no query]
hw1.exe http://www.tamu.edu:80        [no path, port]
hw1.exe http://128.194.135.72:80/courses/index.asp#location [IP, port, path, fragment]
hw1.exe http://165.91.22.70/          [IP, path]
hw1.exe http://s2.irl.cs.tamu.edu/IRL7 [64MB HTML file]
hw1.exe http://s2.irl.cs.tamu.edu/IRL8 [128MB HTML file with zeros]
```

Note that if the path is omitted, you must use the root directory / in its place. If the URL passes basic checks (i.e., correct scheme, non-zero port number), you should attempt to obtain the page via HTTP 1.0. Note that HTTP 1.1 allows the server to chunk the transfer, which is harder to decode (see the extra-credit section at the end of part 3). It is therefore important to request the version of HTTP that you can process.

If you manage to connect and receive a valid reply (HTTP 2xx), parse the HTML result and display the required information about your download (see below); otherwise, the program should legibly report the type of error encountered and terminate gracefully, even if the remote host is hanging or not responding. Note that your code must be able to handle pages of arbitrary length by dynamically expanding the buffer provided to `recv()`.

The following two examples show the required behavior during successful downloads:
Note that one-tab indentation after the first line, an asterisk for the connection phase, a plus for the parsing phase, and timing of each networking step (e.g., using `clock()`) are required. The printout following a horizontal line contains only the HTTP header.

If you manage to receive the page, but the status code is not 2xx, skip the HTML parser, but print everything else:
2.2. General Guidelines

Efficient coding and well-structured programming is expected. You may lose points for copy-pasting the same function (with minor changes) over and over again, for writing poorly designed or convoluted code, not checking for errors in every API you call, and allowing buffer overflows, access violations, debug-assertion failures, heap corruption, synchronization bugs, memory leaks, or any conditions that lead to a crash. Furthermore, your program must be robust against unexpected responses from the Internet and deadlocks.

Basic operation of Winsock is covered in class, with supporting examples provided in the sample homework project on the course website. Additional caveats are discussed next.

2.3. Receive Loop

Reading from sockets is accomplished using this general algorithm that resizes the buffer as needed:

class Socket {
```cpp
SOCKET sock; // socket handle
char *buf; // current buffer
int allocatedSize; // bytes allocated for buf
int curPos; // current position in buffer
... // extra stuff as needed

Socket::Socket ()
{
    // create this buffer once, then possibly reuse for multiple connections in Part 3
    buf = ... // either new char [INITIAL_BUF_SIZE] or malloc (INITIAL_BUF_SIZE)
    allocatedSize = INITIAL_BUF_SIZE;
}

bool Socket::Read (void)
{
    // set timeout to 10 seconds
    while (true)
    {
        // wait to see if socket has any data (see MSDN)
        if ((ret = select (0, &fd, ..., timeout)) > 0)
        {
            // new data available; now read the next segment
            int bytes = recv (sock, buf + curPos, allocatedSize – curPos, ...);
            if (errors)
                // print WSAGetLastError()
                break;
            if (connection closed)
                // NULL-terminate buffer
                return true; // normal completion
            curPos += bytes; // adjust where the next recv goes
            if (allocatedSize – curPos < THRESHOLD)
                // resize buffer; you can use realloc(), HeapReAlloc(), or
                // memcpy the buffer into a bigger array
        }
        else if (timeout)
            // report timeout
            break;
        else
            // print WSAGetLastError()
            break;
    }

    return false;
}
```

The above fragment checks the socket to see if there is any data before attempting a receive. Without this, you may experience deadlocks inside `recv()` when the remote host neither provides any data nor closes the connection. Since `select()` modifies the parameters you pass to it, you must reinsert `sock` into `fd_set` each time you call `select()`. This is accomplished with macros `FD_ZERO` and `FD_SET`. For more details, see


A cleaner alternative to traditional Unix-style `select()` is `WSAEventSelect()` or the IOCP framework. The former lets you register an event that gets signaled when the socket has data in it. This allows your code to wait for multiple events and implement simple timeout-based socket disconnection. The latter is much more complicated and should be attempted only if the rest of the homework appears too simple:
2.4. Required HTTP Fields

The format of GET requests was shown in class. At minimum, you need to transmit the request line and the host string with the name of the server. For example:

GET /some/page/index.php?status=15 HTTP/1.0
Host: tamu.edu

However, this request may keep the connection open for some non-compliant servers, which makes it difficult to detect the end of transfer. You therefore may want to explicitly request that the server close the connection:

GET /some/page/index.php?status=15 HTTP/1.0
Host: tamu.edu
Connection: close

It is also common courtesy to specify your user-agent to keep webmasters aware of visiting browsers and robots. In fact, some websites (e.g., akamai.com) refuse to provide a response unless the user-agent is present in the request header:

GET /some/page/index.php?status=15 HTTP/1.0
User-agent: myTAMUcrawler/1.0
Host: tamu.edu
Connection: close

You should invent your own string in the format of crawlerName/x.y, where x.y can evolve from 1.1 to 1.3 as you progress through the parts of this homework.

2.5. Parser

The sample parser solution (from the course website) contains four library (.lib) files, which need to be copied into your project’s folder with .cpp files. Do not add lib files into the project in Visual Studio. You additionally need HTMLParserBase.h, which should be included into precompiled headers (pch.h or stdafx.h). There is also no need to add .lib files into linker input since HTMLParserBase.h already does this using #pragma directives. There are four different libraries that cover all possible combinations of Debug/Release/win32/x64, where the proper file is automatically determined by HTMLParserBase.h.

You may also run into an issue with Release mode when the newest Visual Studio refuses to include HTML parser libraries because they’re from an earlier version of the compiler. The solution is to disable whole program optimization (Project Properties → C/C++ → Optimization).

2.6. Helpful Functions, Tools, and Commands

You can use C-string functions strchr and strstr to quickly find substrings in a buffer. Comparison is usually performed using strcmp/stricmp or strncmp/strnicmp. It is recommended to use printf as it greatly reduces the amount of typing in this homework compared to cout. You can also use sprintf to assemble the various parts of a request.
Oftentimes, it is convenient to declare a fixed-size buffer that is large enough to accept even the longest link. To help with this, HTMLParserBase.h defines two constants MAX_HOST_LEN and MAX_REQUEST_LEN that upper-bound the examples we consider valid for this homework. If the user provides a string that violates either bound, you should reject it. An explicit check is required, especially in Part 3 where some of the crawled URLs are known to violate the maximum allowed host length.

Usage of gethostbyname for DNS lookups, printout of IPs via inet_ntoa, and connection to a server are provided in the sample solution.

For debugging responses, use an HTTP sniffer, e.g., http://websniffer.com, http://testuri.org/sniffer, or Firefox add-ons. If you need to see the contents of your outgoing packets, use http://www.wireshark.org/. For information about your network configuration, run ipconfig at the command prompt (to see the DNS servers, use ipconfig /all). To manually perform DNS lookups, try nslookup host or nslookup IP.
# 463/612 Homework 1 Grade Sheet (Part 1)

Name: ______________________________

<table>
<thead>
<tr>
<th>Function</th>
<th>Points</th>
<th>Breakdown</th>
<th>Item</th>
<th>Deduction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>1</td>
<td>1</td>
<td>No usage info if incorrect arguments</td>
<td></td>
</tr>
<tr>
<td><strong>Request</strong></td>
<td>3</td>
<td>1</td>
<td>Incorrect GET syntax</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>No hostname in request</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>No user-agent in request</td>
<td></td>
</tr>
<tr>
<td><strong>Receive loop</strong></td>
<td>4</td>
<td>1</td>
<td>No dynamic buffer resizing</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>Fails to receive/parse large files</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>No select()</td>
<td></td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>10</td>
<td>3</td>
<td>Incorrect host/port/request</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect DNS info</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>No timing of connect()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>No timing of recv()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect page size</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect HTTP status</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect number of links</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Incorrect HTTP header shown</td>
<td></td>
</tr>
<tr>
<td><strong>Errors</strong></td>
<td>6</td>
<td>1</td>
<td>Does not handle invalid port/scheme</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Does not notify of DNS failure</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Does not notify of connect failure</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Does not notify of recv timeout/failure</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Does not notify of non-HTTP reply</td>
<td></td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td>1</td>
<td>1</td>
<td>Missing files for compilation</td>
<td></td>
</tr>
</tbody>
</table>

Additional deductions are possible for memory leaks and crashing.

Total points: ________________