Web Crawler

Functional Specification

For this project you will design and implement a web crawler that generates a key word index for a web site (or portion thereof). The web crawler should take a starting URL (Uniform Resource Locator) and index all of the words on the HTML page referenced by that URL. It should then proceed to the other HTML pages that are linked to by the starting URL and index those pages and so on.

Your program must accept a URL that specifies the start of the search for web pages, the name of a directory where the pages generated by your program should be placed, and a file of "stop words" (i.e., words that should not be indexed). The form of the command line for your program should be:

```
crawler startURL outputDirectory stopWordFile
```

The *startURL* is an absolute URL starting with either http:// or file: that specifies the address of the web site to be crawled. For example,

```
http://www.cs.byu.edu/index.html
file:/cs240/project1/main.htm
```

The limit to the set of pages that is indexed by your web crawler is the set of pages that are stored within the prefix specified by the *startURL*. You should ignore any links on pages that move outside the prefix of the *startURL*. The prefix of the *startURL* is everything in the URL before the page name. For example, for the *startURL* http://cs-online.cs.byu.edu/cs240/default.htm, the prefix is http://cs-online.cs.byu.edu/cs240/. You will need to ensure that your program deals with cycles in the links so that it does not get into an infinite loop.

The *outputDirectory* is any valid UNIX directory name where the generated files can be placed. The home page for your generated index should be *outputDirectory*/index.html. All other file names that you generate can be anything you want as long as those files are found in *outputDirectory*.

The *stopWordFile* is the name of a text file that contains a list of words that should not be indexed. These words will appear in the file separated by spaces, newlines or both. These are words such as "a", "the", "to", "is", etc. that are very common but too trivial to appear in an index. We will provide you with this file during pass-off.

Function of a Web Crawler

Web crawlers do a breadth-first search of all of the web pages that are directly or indirectly linked to some starting page. In general the function is to:

```
Select a page that has not yet been indexed

Download the selected page
For all text areas in the page, parse out all of the words.

 For each word that is not a stop word, place the word in the index with
a reference to the page currently being indexed.

 [NOTE: Words are case insensitive.]

For all <A> tags with href attributes and all <FRAME> and <IFRAME> tags with src attributes,
 add the URLs in the href or src attributes to the set of pages that still need to be indexed
 (but only if they're HTML files with the same prefix as the start URL)

 Save summary information for the page

 Repeat until there are no pages left to index

 Generate the HTML pages for the index.
```

Friendly Web Crawling

Your web crawler must obey the restrictions specified by web site administrators in `robots.txt` files. Before crawling a web site, your crawler should try to download the site's `robots.txt` file. If the file exists, your crawler should not download any URLs that

are forbidden for `User-agent: *`. Of course, you may use the code you wrote for the Web Robot Filter project to perform this processing.

NOTE: `robots.txt` processing should only be done when the start URL begins with `http://`. If the start URL begins with `file:`, do not process `robots.txt`.

While testing your web crawler do not repeatedly hammer a particular site. For simple testing you should create a small set of web pages in a local directory and use them to work with, so that you do not slow down a real site with excessive traffic. Additionally, real web sites can be very large. Crawling such sites can consume a lot of network bandwidth and disk space. Please use wisdom when selecting web sites to test your crawler on.

HTML

This project requires some knowledge of HTML, but not much. If you don't already know HTML, one of the easiest ways to learn it is to look at the source for web pages that you commonly use. In Netscape you can use the Page Source item found in the View menu. There is a similar feature in Internet Explorer. One can also read about HTML here. The key tags that you should understand are <A>, <FRAME>, <IFRAME>, <TITLE> and the heading tags <H1>, <H2>, ...

Finding Links

Your program needs to be able find links in HTML documents and process them appropriately. Links in HTML documents are stored as attribute values on various kinds of tags. While many HTML tags can contain links to other documents, for this project we will only process links contained in the following tag types:

- `<A>` tags have an attribute named `HREF` that contains the URL of a linked document
- `<FRAME>` and `<IFRAME>` tags have an attribute named `SRC` that contains the URL of the document that contains the frame's contents

Attribute values inside HTML tags can be delimited with double-quotes, single-quotes, or no quotes at all. Your program should handle all of these cases. For example, all of the following are valid HTML tags:

```
<A href=http://www.cnn.com/>

<A href=http://www.cnn.com/>

<A href=http://www.cnn.com/>
```

Your program should also properly handle whitespace characters when parsing HTML tags. For example, all of the following are valid HTML tags:

```
<A href   =http://www.cnn.com/   >

<A href=   http://www.cnn.com/>

<A
   href   =
   http://www.cnn.com/
>
```

In HTML, the names of tags and attributes are case insensitive. For example, the following HTML tags are equivalent:

```
<a href=http://www.cnn.com/>

<A HREF=http://www.cnn.com/>

<a HREF=http://www.cnn.com/>

<A hReF=http://www.cnn.com/>
```

However, attribute values (i.e., the stuff between the quotes) are case sensitive. Do not modify the case of attribute values, including the URLs stored in HREF and SRC attributes.

Many HTML files on the Internet have small syntax errors in them. While these files are technically incorrect, all web browsers are quite forgiving in ignoring such errors and they try to display such files the best that they can. If your program is to be robust enough to work on real web sites, it will also need to be as forgiving as possible when it encounters files containing invalid HTML. For the Web Crawler, this means that it would not be reasonable for the program to immediately terminate when it encounters an HTML file containing invalid syntax. You may choose to skip over a file that contains invalid HTML syntax, or, even better, just skip over the invalid parts and try to process the parts that are valid.

Processing URLs

The links in HTML files are represented as URLs. There are two kinds of URLs, absolute and relative. Your program must handle both absolute and relative URLs, as described below.

Absolute URLs

An absolute URL fully specifies the address of the referenced document. These are the addresses that you would type into a web browser in order to visit a web page.

The general format of an absolute HTTP URL is:

```
<scheme>://<net_loc>/<path>;<params>?<query>#<fragment>
```

For example, consider the following absolute URL:

```
http://www.espn.com:80/basketball/nba/index.html;lang=engl?team=dallas#Roster
```

The parts of this URL are:

```
<scheme>      http
<net_loc>     www.espn.com:80
<path>        /basketball/nba/index.html
<params>      lang=engl
<query>       team=dallas
<fragment>    Roster
```

Your program must be able to handle absolute URLs according to the syntax defined above. However, several parts of an absolute URL are optional, and will not appear in every URL. Specifically, the `<path>`, `<params>`, `<query>`, and `<fragment>` parts might not be present. If the `<path>` is missing, it is assumed to be /. If the `<params>`, `<query>`, and `<fragment>` parts are missing, they are empty.

The `<scheme>` and `<net_loc>` parts are case insensitive, but the `<path>`, `<params>`, `<query>`, and `<fragment>` parts are case sensitive. This means that

```
http://www.cnn.com/index.html
```

```
HTTP://WWW.CNN.COM/index.html
```

are equivalent, but

```
http://www.cnn.com/index.html
```

```
http://www.cnn.com /INDEX.HTML
```

are not equivalent.

The general format of a FILE URL is:

```
file:<path>#<fragment>
```

Restricting the Crawler's Scope

For this program, an absolute URL must start with either `http://` or `file:`. If a link does not start with `http://` or `file:`, then it is treated as a relative URL. In order to be a valid link, an absolute URL must have the same prefix as the *startURL* given on the command-line. The prefix of the *startURL* is everything in the URL before the page name. For example, if the *startURL* is:

```
http://cs-online.cs.byu.edu/cs240/default.htm
```

then the prefix of the *startURL* is

```
http://cs-online.cs.byu.edu/cs240/
```

If the first characters of the link's address do not exactly match the prefix of the *startURL* , then it is not a valid link.

Relative URLs

Relative URLs are all the links that do not begin with either http:// or file:. Unlike absolute URLs, relative URLs do not fully specify the address of the referenced document. Rather, they specify the address of the document relative to the address of the document containing the link (the *base document*).
Here are some examples of relative URLs:

```
���  /images/nasdaq.jpg

  ·  ./images/nasdaq.jpg

     ../../images/nasdaq.jpg

     #HEADLINES

     images/nasdaq.jpg
```

Before your program will be able to download a document whose address is specified by a relative URL, it will first need to convert (or resolve) the relative URL to an absolute URL that represents the document's full address.· Resolving a relative URL is done by combining the absolute URL of the base document (the *base URL*) with the relative URL, as described below.

If the relative URL begins with a ·/·, the absolute URL is constructed by pre-pending file: or http://<net_location> from the base URL (whichever applies).· For example,

```
     Base URL: ·····  http://www.cnn.com/news/financial/index.html

···· Relative URL: ···  /images/nasdaq.jpg

···· Resolved URL:···· http://www.cnn.com/images/nasdaq.jpg


     Base URL:·····  file:/news/financial/index.html

···· Relative URL: ···  /images/nasdaq.jpg

···· Resolved URL:···· file:/images/nasdaq.jpg
```

If the relative URL begins with ·. /·, the URL is relative to the directory containing the base document.· For example,

```
     Base URL:·····  http://www.cnn.com/news/financial/index.html

     Relative URL: ···  ./images/nasdaq.jpg

     Resolved URL: ···  http://www.cnn.com/news/financial/images/nasdaq.jpg


     Base URL:·····  file:/news/financial/index.html

     Relative URL: ···  ./images/nasdaq.jpg

     Resolved URL: ···  file:/news/financial/images/nasdaq.jpg
```

If the relative URL begins with ·. . /·, the URL is relative to the parent directory of the directory containing the base document.· For example,

```
     Base URL:·····  http://www.cnn.com/news/financial/index.html
```

Relative URL: ��� ../images/nasdaq.jpg

Resolved URL: ��� http://www.cnn.com/news/images/nasdaq.jpg

Base URL:��������� file:/news/financial/index.html

Relative URL: ��� ../images/nasdaq.jpg

Resolved URL: ��� file:/news/images/nasdaq.jpg

Note that a relative URL can begin with any number of**.**/, such as ../../../images/nasdaq.jpg. Each **..**/ indicates that you should go up one more levels in the directory hierarchy.

If the relative URL begins with #, the URL is relative to the base document itself. In this case, the fragment represents a specific location within the base document. For example,

Base URL:��������� http://www.cnn.com/news/index.html

Relative URL: ��� #HEADLINES

Resolved URL: ��� http://www.cnn.com/news/index.html#HEADLINES

Base URL:��������� file:/news/index.html

Relative URL: ��� #HEADLINES

Resolved URL: ��� file:/news/index.html#HEADLINES

If none of the preceding cases apply (i.e., it doesn't begin with /, **.**/, **..**/, or #) , the URL is relative to the directory containing the base document, just as if it had started with **.**/. For example,

Base URL:��������� http://www.cnn.com/news/financial/index.html

Relative URL: ��� images/nasdaq.jpg

Resolved URL: ��� http://www.cnn.com/news/financial/images/nasdaq.jpg

Base URL:��������� file:/news/financial/index.html

Relative URL: ��� images/nasdaq.jpg

Resolved URL: ��� file:/news/financial/images/nasdaq.jpg

Fragment Processing

URLs may have fragment identifiers at the end to identify particular locations within a document. For example, the two URLs below are identical, except for the fragment identifiers "#SPORTS" and "#WEATHER" at the end:

http://www.cnn.com/news/headlines.html#SPORTS

http://www.cnn.com/news/headlines.html#WEATHER

Both of these URLs refer to the page http://www.cnn.com/news/headlines.html, but when loaded in a browser they display different sections of that page. In general, fragment identifiers can be used control which part of a page is displayed when it's loaded in a browser. This is useful for interactive web browsing, but has little meaning for a web crawler. After resolving relative URLs (if necessary), your web crawler should strip off and throw away fragment identifiers if they are present. This will prevent your crawler from erroneously indexing the same web page multiple times, simply because the fragment identifiers made the URLs different. In the example above, http://www.cnn.com/news/headlines.html should be indexed only once, even if there are multiple URLs with

different fragment identifiers that reference it.

HTML vs. Non-HTML Files

Your program only needs to index HTML files. Therefore, it will need distinguish between HTML and non-HTML files. For the purposes of this project, a file is considered to be HTML if any of the following conditions hold:

(1) The `<path>` part of the URL is a directory name (i.e., it ends with "/"). For example, `http://www.espn.com/football/`

(2) The file name in the URL's `<path>` does not end with a file extension (i.e., the file name contains no periods). For example, `http://www.espn.com/football/scores`

(3) The file name in the URL's `<path>` ends with one of the following extensions: `.html, .htm, .shtml, .cgi, .jsp, .asp, .aspx, .php, .pl, .cfm`. (You may add other file extensions to this list if you want.) For example, `http://www.espn.com/football/scores/index.html`

NOTE: Determining whether or not a URL represents an HTML file is based solely on the `<path>` portion of the URL. If the URL contains the `<params>`, `<query>`, or `<fragment>`, they should be ignored when distinguishing between HTML and non-HTML files.

In the case of a relative URL, the URL should be resolved to an absolute URL before deciding whether the URL represents an HTML document.

Your crawler may encounter HTML files that contain syntax errors, or non-HTML files that pass for an HTML file according to the rules above. In order to be robust, your HTML parser should be forgiving of files that do not precisely conform to the HTML specification. As long as the HTML tags you're searching for are correctly formed, your parser should be able to index a file. If your parser decides that a file does not conform to the HTML specification well enough to be indexed, you may skip that file, but your program should not terminate simply because an invalid HTML file is encountered. Instead of quitting, it should print out a good error message including the URL of the invalid file, and then move on to the next page.

Web Crawler Output

The home page for your generated output "index.html" should contain some kind of a welcoming header and a list of all letters in the alphabet. The welcoming header should also indicate the *startURL* for which this is an index. Each letter should be a link to a letter page for that letter.

A letter page should contain a header which indicates letter that this page is an index for and a link back to the home page. Each such letter page should contain all of the index words that begin with that page's letter. Remember that an index word is any word that appears in the text portion (not inside of the tags) of any HTML page and excluding any words found in the stop word file. Index words are also case insensitive. Each index word should only appear once on a letter page and should be a hyperlink to a word page for that word. The index words should appear in alphabetical order.

A word page should contain a header that indicates the word for this page as well as links to the home page and to the letter page for this word. On a word page each located page that contains that word should be listed. A located page is listed using its summary. A page's summary is the contents of the <TITLE> tag (if there is one). If there is no <TITLE> tag in the page, then it should be the text contents of the first header tag (<H1>,<H2>, <H3>,...) in the file. If there are no <TITLE> or header tags then the summary should be the first 100 characters of whatever text is not found inside of a tag. The page summary information should be hyperlinked to the actual page itself using the URL that you stored during the web crawling phase.

Efficiency and Data Structure Selection

Although your program will usually be run on fairly small web sites, it must be designed so that it will perform well even when it is run on large web sites. You should assume that the internal data structures of your program could grow to be very large, and therefore you should choose data structures that support fast insert and search operations even when they become very large.

Memory Management Errors

Your program must not contain memory management errors. You should use the Linux tool named `valgrind` to check your program for memory management errors. `valgrind` can detect many kinds of memory-related bugs, including memory leaks, reading from invalid memory locations, and writing to invalid memory locations. The TAs will use `valgrind` to check your program for memory management errors when you pass off.

To use `valgrind`, you should first compile and link your program using the `-g` flag, which tells the compiler and linker to include debugging information in your executable. After doing that, `valgrind` may be executed as follows:

```
valgrind --tool=memcheck --leak-check=yes --show-reachable=yes executable param1 param2
```

`valgrind` will print out messages describing all memory management errors that it detects in your program. This is a valuable debugging tool. After your program completes, `valgrind` will print out information about any heap memory that was not deallocated before the program terminated. You are required to remove all memory leaks before passing off. You are not responsible for memory leaks in the standard C++ library or other libraries on which your program depends. For example, the C++ `string` class allocates memory on the heap which it never deallocates. `valgrind` will report this as a memory leak, but you are not responsible for it. You are only responsible for memory allocated directly by your program. `valgrind` will also print out `Invalid read` and `Invalid write` messages whenever your program reads from or writes to an invalid memory location. You are required to remove all invalid reads and writes before passing off.

Please note that running your program with `valgrind` will cause your program run much slower than usual. Therefore, you might not want to always run your program with `valgrind`. Rather, run `valgrind` periodically during your program's development to find and remove memory errors. It is not recommended that you wait until your program is completely done before running `valgrind`.

Exception Handling

Your program must properly handle C++ exceptions. For this project, this means that your program must not abnormally terminate due to an unhandled exception. Please note that the Web Access classes throw exceptions when they encounter file or network I/O errors, which can be a common occurrence when running a web-based application like a web crawler.

Additional Notes

- A word is any contiguous sequence of alphabetic letters, numbers, underscore _ or dash - that begins with a letter, and does not appear inside a tag. Words are terminated by whitespace, punctuation or any other character that is not a letter, number, underscore or dash. Examples: `123Bob` is not a word because it does not begin with a letter. `Bob&Jim` contains two words: `Bob` and `Jim`. `Bob's house` gives three words: `Bob`, `s`, and `house`, since Bob and s are separated by punctuation. BobbyJim gives the three words: `Bob`, `by`, and `Jim`, since the HTML tags count as punctuation to separate the words.

- The correct definition of an HTML tag is a sequence of characters that starts with < followed by an alphabetic character (A-Z or a-z), without any whitespace between the < and the alphabetic character. All properly formatted HTML tags will end with >. If there is a < in the file that is not directly followed by an alphabetic character, then it should be treated as punctuation. HTML tags can appear anywhere inside an HTML document. All HTML tags should be ignored, except for title tags, header tags, and links.

- URLs are case sensitive, so changing the case of a URL will result in a URL that does not work.

- You don't have to parse any text or HTML tags that appear outside of the <HTML> ... </HTML> tags in a file. You should only index words that appear inside either the title tags or the body tags of the page. Any words outside the <BODY> </BODY> tags should not be indexed (unless they are in the page's <TITLE></TITLE> tags).

- You should process all links that are in <A>, <FRAME>, or <IFRAME> tags. Remember that there could be other information before the href or src part of the tag. For example, the following is a valid link:

  <a target="main" href="bob.htm">

- Ignore everything inside <SCRIPT> </SCRIPT> tags and HTML comments <!-- -->

- You should also ignore any special characters - there are two types of HTML special characters:
  1. The first type of special character always starts with & (an ampersand) and ends with ; (a semi-colon) and never has any whitespace in between the & and the ;. For example:
     >

2. The second type of special character always starts with & (an ampersand) followed by a # (a pound sign) followed by three digits (with no whitespace in between) followed by a ; (a semi-colon). For example:
   {
   <

- Your crawler must be scalable to any size of web site.

- Your crawler must be able to handle VERY LARGE stop word files (e.g., 100,000 entries) and still run efficiently.&#65533; This means that you should not use a linear search when searching for stop words.

- The stop words file is guaranteed to be sorted in ascending order.

Design Document

The first step in developing a larger program like a web crawler is to spend some time understanding the problem that is to be solved.&#65533; Once you understand the problem, you can start to design the classes that you will need to implement by simulating the operation of the program in your mind, and creating classes that perform each of the functions required by the program.&#65533; For each class that you create, you should document what responsibilities the class has and how it interacts with the other classes in your design to perform its responsibilities.&#65533; This exercise will help you determine what classes you need to write, and how those classes will work together to produce a working program.&#65533; Doing this type of design work before coding saves time because it helps avoid spending effort on dead-end ideas that don't work.

Once you've thought through your design the best that you can without writing any code, you should make a first attempt at implementing your design.&#65533; As your implementation progresses, you will probably find that your design was incomplete or faulty in some respects.&#65533; This is to be expected because some insights only come from actually implementing a design.&#65533; As you proceed, make necessary changes to your design, and incorporate them into your code.

To encourage you to follow this type of design process, you are required to submit a design document for your program. Your design document must include three sections:

1)&#65533; A **DETAILED** description of the data structures that you will use to store the program's data.&#65533; Describe in detail what data needs to be stored and how it will be stored (e.g., binary trees, hash tables, queues, arrays, linked-lists, etc.).&#65533; Also explain why you chose the data structures that you did.

2)&#65533; For each class in your design, provide the following:

- The name of the class and a description of its purpose
- A list of the major variables and methods in the class, and a description of each one&#65533;s purpose or function

3)&#65533; A **DETAILED** description of the major algorithms in your program. For the web crawler, this includes: 1) the algorithm that drives the downloading and indexing of documents, 2) your HTML processing algorithm, and 3) your output generation algorithm. The best way to document these algorithms is to actually write the top-level code for them in terms of the specific classes and methods in your design. This will help you see how your classes will work together at runtime to accomplish the web crawling task.

   Submit a hard-copy printout of your design document to the TAs before midnight on the due date.&#65533; Please make sure that your name is clearly visible on the front of your design document.&#65533; Design documents may <u>not</u> be submitted by email.

The C++ string Class

You are encouraged to use the C++ string class in this project.

Standard Template Library (STL)

One of the learning objectives of this project is to give you significant experience with pointers and low-level memory management in C++. Since the STL largely relieves the programmer of these responsibilities, you <u>are not</u> allowed to use the STL on this project. Specifically, the following header files may not be used:

- <algorithm>
- <deque>
- <list>
- <map>
- <queue>
- <set>
- <stack>
- <vector>

Web Access

Your Web Crawler will obviously need to download web documents. For your convenience, the CS 240 Utilities include several classes that can be used to download web documents. The documentation for these classes is found here. You are required to use these classes to download documents. These classes throw exceptions when errors occur, and your program must handle these exceptions (i.e., it can't just crash due to an unhandled exception).

Unit Tests

You are required to write thorough, automated unit test cases for your URL class (i.e., the class that parses URLs, resolves relative URLs, etc.). Your URL class should have a public method with the following signature

```
static bool Test(ostream & os);
```

that contains the unit tests. The `Test` method should create one or more URL objects, call methods on them, and automatically check that the results returned by the methods are correct. If all test cases succeed, `Test` should return true. If one or more test cases fail, `Test` should return false. For each test case that fails, an informative error message that describes the test that failed should be written to the passed-in output stream. This will tell you which tests failed so that you can fix the problems.

You should also write a test driver program that runs all of your automated test cases. This program will be very simple, consisting only of a main function that calls your `Test` method. Whenever you make a change to your URL class, you can recompile and run your test program to make sure that the new code works and that it didn't break any code that was already there.

Of course, you are free to write unit tests for as many classes as you like, but this is not required. We just want you to get hands on experience writing some unit tests so you will better internalize the concept of unit testing.

The file `UnitTest.h` in the CS240 Utilities contains code that is useful for creating automated test cases.

Static Library

You are required to create a static library containing the CS 240 Utilities classes, and to link this library into your program. In class we will teach you how to create static libraries using the Linux `ar` command, and how to link them into a program. When you pass off your program, the TA will ask you to un-tar your source code and build your program. Creating and linking the static library will be part of the build process. Of course, you will also be asked to demonstrate that your program works.

Make File Automation

You are required to automate the compilation and testing of your project using a make file. Your make file should support the following functions:

- build the static library containing the CS 240 Utilities classes
- build the executable `make` program
- compile and run your automated unit test cases
- remove all of the files created by the build process

Your make file must recognize the following targets:

Target	Example	Meaning
lib	`$ make lib`	Compile the CS 240 Utilities classes, and package them into a static library named `libcs240utils.a`

bin	`$ make bin`	Compile and link your program. This target should first rebuild the static library if it's out of date.
test	`$ make test`	Compile and run your automated test program. The test program should contain a main function that simply calls the Test methods on all of your classes. If all tests succeed, the program should print out a message indicating success. If one or more tests fail, the program should print out a message indicating failure, and also print out a message describing each test that failed. This target should first rebuild the executable if it's out of date.
clean	`$ make clean`	Remove all files and directories created by the other targets. This returns the project directory to its original state.

An initial version of your make file will be passed off early on in the project, along with your implementation of Collections II. The purpose of this deadline is to help you get started early on the first large project in the course. When you write your initial make file, you will not have written much code yet, so the question arises: What should the initial make file do? The answer to this question is the following:

1. The lib and clean targets should function as described above.
2. The bin target should compile and link your executable, which won't be much more than an empty main function to begin with.
3. The test target should build and run your unit test driver executable, which eventually will be used to run the Test methods on your classes. At the beginning of the project you won't have any Test methods yet, so your test driver will initially do nothing more than print out a message like "All Tests Succeeded" (which is true, since you don't have any tests yet).

Code Evaluation

After you have submitted and passed off your program, your source code will be graded on how well you followed the good programming practices discussed in class and in the textbook. The criteria for grading your source code can be found at the following link:

Code Evaluation Criteria

Important Advice

This is a big project. It will take the average student approximately 50 hours to complete. You have 4 weeks to complete it, and you are strongly encouraged to get started immediately and work at a steady pace for the entire 4 weeks until you are done. If you procrastinate, it is highly unlikely that you will finish on time. Although you can pass it off late, every additional day you spend working on this project will be one less day that you have to work on the next project, which is approximately the same size as this one. Your success in this class will largely depend on how diligently you follow this advice.

Submitting Your Program

Create a gzip-compressed tar file containing all of your project's source files and directories. The name of the compressed tar file must have the following format:

```
firstname_lastname.tgz
```

where `firstname` should be replaced with your first name and `lastname` should be replaced with your last name. For example, if your name is Bob White, you would use the following command to create your Web Crawler tar file:

```
$ tar czf Bob_White.tgz webcrawler
```

(this assumes that your project files are stored in a subdirectory named `webcrawler`).

NOTE: In order to minimize the size of your tar file, please make sure that it only contains C++ source files, build scripts, and any data files that are needed by your test cases. It should not include .o files, .so files, executable files, or HTML files created by previous executions of your web crawler. If your tar file is larger than 500kb (half a megabyte), you will not be allowed to submit it. If you follow these instructions, your tar file will probably be much smaller than 500kb. If it is bigger than 500kb, you need to delete some files and recreate the tar file.

After you have created your tar file, click on the following link to go to the project submission web page:

Web Crawler Submission Page

After you provide your CS account login name and password and the name of your tar file, this page will upload your tar file to our server.

After submitting your tar file, in order to receive credit you must also pass off your project with a TA. When you pass off, you will be asked to download your tar file from our server using the the project retrieval page, which is located at the following link:

Web Crawler Retrieval Page

After downloading your tar file, you will be asked to:

1. un-tar your tar file
2. demonstrate that your make file supports all of the required functions
3. demonstrate that your program works (this includes having no Valgrind violations)

If the TA finds problems with your program, you will need to fix your program, submit a new tar file containing your modified program, and then pass it off again with the TA.

Your tar file must be submitted before the deadline in order to receive full credit, but you may pass off your project with a TA after the deadline. If you do this, you run the risk that the TA might find problems with your program, and you will need to resubmit your tar file after the deadline. In this case, the fact that you submitted your first tar file before the deadline will not help you. The time of your last tar file submission will be used to compute your grade.

Ken Rodham