

# CS32 Winter 2012

## Project #4

### Ad Hunter

Due: 9 PM Thursday, March 15

Make sure to read the entire document (especially Requirements and Other Thoughts)  
before starting your project.

For questions about this project, contact Carey Nachenberg: [climberkip@gmail.com](mailto:climberkip@gmail.com)

## Table of Contents

Introduction .....	3
Details of the Rules File .....	7
But I don't know how to use C++ to access the Internet! .....	8
The Tokenizer Class .....	9
Ok, so what is it I have to do? .....	10
The MyHashMap Class .....	10
The ExtractLinks Class .....	13
The Document Class .....	16
The Crawler Class .....	18
The Rule Class.....	22
The Matcher Class.....	25
The AdHunter Class .....	28
Your main() Function.....	29
How do Command Line Parameters/Arguments work?.....	31
Requirements and Other Thoughts.....	34
What to Turn In .....	36
Grading.....	37

## Introduction

For your fourth and final project, you've been hired by WeSellSleazyAds.com, the world's seventeenth largest online advertising firm. As with other advertising firms, WeSellSleazyAds.com makes its money by embedding advertisements on third-party web pages like blogs and Facebook pages. Due to the recent economic downturn, WSSA lost millions of dollars in advertising revenue and had to lay off their entire staff except for two people, Carey and David. In the past, WSSA used a stable of employees to manually decide which ads were appropriate for which web pages. However, with just two employees left, WSSA can no longer afford to have humans figure out which ads to display to customers on which web pages, so they've decided to automate this process.

So how does Internet advertising work?

Companies like Ford, Honda, and Tesla Motors would like to place their advertisements in front of users who are interested in buying cars. Apple, HP, and Lenovo would like to place their ads in front of users who are interested in buying new PCs. And so on. These advertisers are willing to pay WSSA (or Google, for that matter) a certain amount (e.g., \$.75/ad) to display ads to users on web pages for their products. Of course, Ford would rather place ads for its new cars on a car-related blog than on a cooking-related blog, so Ford (and other advertisers) specify requirements to WSSA about where they would like their ads displayed across the Internet.

Now let's say a blog owner who writes a blog about fancy cars wants to make some extra money on the side (from their main job as an auto mechanic). The blog owner regularly writes interesting blog entries about the latest cars on their blog page and this presumably attracts people interested in cars to their web site. These readers are a potential gold mine for (automotive) advertisers. The blog owner is called a "host" since they host content.

The blog owner (the host) can contact a company like WSSA and indicate to WSSA that they are willing to host one or more advertisements on their web site. WSSA can analyze the blog owner's web site and determine that it is automotive-related, and therefore that Ford, Honda, and Tesla Motors, among others, would be willing to post their advertisements on this page. WSSA tells the blog owner how to modify the web page so that it connects directly to WSSA's servers to retrieve new advertisements every time a user views the blog's web pages. These advertisements are then displayed in real time on the car blog as users surf to the blog's pages.

In this case, most of the ads will likely be from automotive companies, since WSSA can make more money by placing automotive-related ads on an automotive web site. Each time a reader of the blog clicks on such an ad, the advertiser (e.g., Ford) is billed by WSSA for their agreed-upon charge (e.g. \$.75). WSSA keeps 90% of this total, and gives the other 10% to the blog owner.

So everyone is happy. The advertiser gets their ads in front of potential buyers. The blog owner makes some extra money on the side from hosting ads on their blog page, and WSSA makes money by connecting the two parties together.

So what do you need to do for WSSA?

WSSA would like to hire you to build a C++ program that takes as input a series of advertisements (e.g., for popular products like Tide, Smuckers Jelly, Ford cars, etc.) and a series of web sites (e.g., carsblog.com, cooking.com, blogspot.com, etc.) whose owners have contacted WSSA and asked to host WSSA's advertisements. Your program must "crawl" the web pages on these sites and examine each page to determine the most appropriate ad(s) that should be displayed on those pages. WSSA will use this information to decide which ad or ads to place on each web page to maximize their revenue.

So how does WSSA know if an advertisement is appropriate for a given web page? Well, each advertiser (e.g., Ford) provides WSSA with one or more "Advertising Rules" that specify the details of their advertisement. Each rule contains the following information:

1. An identification code for the advertisement, e.g., "rule-00000576". This code uniquely identifies the advertisement for billing purposes.
2. The actual advertisement text, e.g., "Click here to save 30% on your new Ford vehicle!"
3. A dollar value for a click on that advertisement (e.g., if the advertisement is placed on a host's web page, and a user clicks on the advertisement while reading the page, how much must the advertiser pay WSSA for that click).
4. A set of keywords that must be found on a web page for the advertisement to be appropriate for that page. For example, Ford might indicate that a web page must contain the words "car" AND "purchase", or "automobile" AND "lease" for their advertisement to be displayed on that web page. Ford will only pay WSSA for ads that are displayed on appropriate web pages that meet their requirements. After all, why would Ford want its ads displayed on pages for garlic presses?

WSSA has collected tens of thousands of such Advertising Rules from thousands of different advertisers – these rules will be provided to you in a data file. WSSA has also collected a list of "hosts" who are willing to host these ads on their web pages to make extra money. This list of web URLs will also be provided to your program in a data file. Your program must determine which of the tens of thousands of rules are applicable to each of the web pages found on the advertising hosts' web sites.

Your program must "crawl" each host's web site (a given web site may consist of many individual web pages, not just one) and determine which advertisements, if any, are appropriate for each web page on the host's site. What does it mean to "crawl" a host's web site? Well, each host provides WSSA with "seed" web page – this is typically the root page of their web site (e.g. "www.carsblog.com").

Your program starts by downloading the specified seed web page off the Internet<sup>1</sup>. Your program then analyzes the web page to determine which of the tens of thousands of advertisements, if any, are appropriate for it. Any advertisements that are a match are collected for later output by your program for WSSA's executives. Once your program has finished processing the seed web page from a site and identified any appropriate ads for that page, your program must then search that web page for embedded links to other web pages on the host's web site.

For example, the [www.carsblog.com](http://www.carsblog.com) web page might contain links to [www.carsblog.com/ford](http://www.carsblog.com/ford), [www.carsblog.com/tesla](http://www.carsblog.com/tesla) and [www.carsblog.com/used\\_cars](http://www.carsblog.com/used_cars). Your program must download each of these other pages, in turn, and analyze each to determine which of the tens of thousands of ads are appropriate for them. Again, any advertisements that are a match for these subsequent pages are collected for later output to WSSA executives. These secondary pages (e.g., [www.carsblog.com/tesla](http://www.carsblog.com/tesla)) might also contain links to yet other web pages (e.g., [www.carsblog.com/chevy\\_volt](http://www.carsblog.com/chevy_volt)) on the host's web site, so your program must also download and analyze these pages for advertisements as well, and so on, and so on.

Of course a given web site could literally have millions of such linked pages, but WSSA only wants you to crawl through a limited number of the most highly-visible pages on each web site. For example, WSSA might want your program to crawl only the first 100 distinct web pages starting from [www.carsblog.com](http://www.carsblog.com) that it visits rather than all 50,000 pages on that website. This information (the # of pages to crawl per site) will be provided as an input to your program.

Once your program has processed all of the web pages from a particular web site (starting with its seed page and crawling out), it must then repeat the process for the next seed page from the next web site on its input list, etc. After all web pages from all web sites have been analyzed and appropriate advertisements have been identified for each of these pages, your program must output a sorted list of matching advertisements, ordered from highest-paying ad to lowest-paying ad that were relevant across all of the crawled pages.

WSSA will also specify (to your program) the minimum acceptable price they're willing to accept for ads. So if Ford is willing to offer only \$.10 if a user clicks their ad, and WSSA doesn't want to consider ads that pay less than \$.20/click, then your program must throw away these low-paying results rather than present them in its output.

So, to summarize:

The inputs to your project are as follows:

1. A data file containing tens of thousands of rules that WSSA provides, e.g. (the rule format is explained later):

---

<sup>1</sup> We'll provide you with code to download a web page off the Internet. Once you finish the project, you'll have all the tools you need to actually write programs that interact with the World Wide Web!

- rule-00000000 0.81 LOSING HAIR & > Click to learn how HayrGrow can help you grow hair!*  
*rule-00000001 0.75 BALD SPOT & THINNING HAIR & | > Click here for a full head of hair!*  
 ...  
*rule-00099999 1.50 ELECTRIC CAR AUTO | & > Click here to learn more about electric cars!*
2. A data file containing a list of seed web pages for host sites that your program must crawl, e.g.:
    - a. <http://www.baldpeople.com>
    - b. <http://www.hairblog.org>
    - c. <http://www.nohaironmyhead.com>
  3. A count, N, of the maximum number of web pages per website that must be crawled. For example, 100 would indicate that you would not crawl and analyze more than 100 distinct web pages on any of the above web sites.
  4. The minimum price that WSSA is willing to receive to host an ad for their advertisers. Any advertisements that have a lower payment value (even if they match a web page) must be omitted from your program's results.

You must do the following:

1. You will define a number of classes, detailed below, to implement the project.
2. You will write a main function that ties all of your classes together into a single program that crawls all of the websites and applies the advertising rules to each website.
3. Your program must output a list of all matching rules across all of the web pages that were crawled, ordered from the most profitable matching rule to the least profitable matching rule.
4. And of course, your program has to be extremely efficient since it applies tens of thousands of rules to hundreds or thousands of web pages. We'll specify efficiency requirements later in this document.
5. You must implement a hash table class and must use this class in two of your classes (details below).

So, what does this all look like? Well, below is the output from our solution. When you run your Ad Hunter program, you must specify four different parameters on the command line:

```
C:\CS32\PROJ4> proj4.exe rules.txt sites.txt maxpages minprice ← [Enter]
```

The first argument to your program, shown above as rules.txt, is the name of a file that contains WSSA's advertising rules. Details about the rules file can be found in the section below.

The second argument to your program, shown above as sites.txt, is the name of a file that holds the list of seed web pages that you must crawl from. This is simply a text file with one or more URLs (e.g. <http://www.wikipedia.com>), one URL per line.

The third argument to your program, shown above as maxpages, is an integer that specifies the maximum number of pages per website that may be crawled starting from a given seed website.

The fourth and final argument, shown above as minprice, is a floating point number that indicates the minimum price per ad that is allowed. All ads that pay less than this amount should be omitted from any output of your program.

Here's what that output might look like when your program finishes running:

*There were 4 matches found that met your minimum of \$0.50 per ad:*

*Match 0: rule-00000e94 triggered on <http://baldblog.com/whyme> for a value of \$0.78*

*Advertisement: Losing hair? Click here to learn about how HayrGro can fix all your problems.*

*Match 1: rule-00075233 triggered on <http://baldblog.com/transplants> for a value of \$0.72*

*Advertisement: Click here to learn more about Smallberg hair restoration!*

*Match 2: rule-91952975 triggered on <http://carsblog.com/tesla> for a value of \$0.65*

*Advertisement: Don't buy a Tesla – buy a Chevy Volt instead. Click here for a deal!*

*Match 3: rule-00075241 triggered on <http://baldblog.com/whyme> for a value of \$0.52*

*Advertisement: Have hairy credit card problems? Click here for help!*

## **Details of the Rules File**

The rules file contains multiple advertisement rules, one rule per line, in the following format:

Rule_ID	Price	Keyword_expression	>	Advertisement text
---------	-------	--------------------	---	--------------------

Where:

- Rule\_ID is a string of the form: rule-#####, where # is a digit or a letter.
- Price is a floating point number like .75 (75 cents) or 1.5 (\$1.50)
- Keyword expression is a postfix expression of alphanumeric terms, & (AND) operators, and | (OR) operators. For example:
  - The expression “hair” would require that the web page must have the word “hair” for the rule to trigger.
  - The expression “thinning hair &” would require that the web page must have both the word “thinning” AND the word “hair” for the rule to trigger.
  - The expression “balding bald | drugs &” would require that the web page must have either the word “balding” or the word “bald”, and in addition to one of these two words, must also have the word “drugs” in order for the rule to trigger.
  - The expression “PC laptop tablet android iphone |||” would require that one or more of those terms be present in a web page in order for the rule to trigger.
  - The expression “Carey David Studs & &” would require that all three words be present in a web page for the rule to trigger.

- The > is just a separator character that is placed between the expression and the advertisement.
- The Advertisement text is a non-empty sequence of words, numbers, or punctuation such as: *Click here to win a new big-screen TV!*

Here are some example rules from the rules.txt file we'll provide to you as a test case:

```
rule-0000002e 0.36 ECCLESIA > Save 10% on ECCLESIA if you buy today!
rule-0000004a 0.96 UNFASHIONABLY COLORCASTS | TIPPET & > Click here to get your free UNFASHIONABLY
rule-000000cc 0.75 FOXSKIN VIRILIZE BREEDER & & > Buy VIRILIZE now and save!
```

You may assume for the purposes of this project that the rules file will contain no more than 200,000 unique rules.

## ***But I don't know how to use C++ to access the Internet!***

Oh, we knew you were going to say that! Such a whiner! But wouldn't you like to learn how to write a program that interacts with other computers over the Internet? We thought so. So we're going to provide you with a reasonably functional Internet HTTP interface that is capable of downloading pages off of the Internet for you. HTTP is the protocol used by web browsers to download web pages from servers on the Internet into your browser.

When you use our interface, you don't have to worry about the details of how to communicate over the Internet yourself. Of course, if you want to see how our interface works, you're welcome to do so... and before you know it, you'll be forming your own start-up Internet company to compete against Google<sup>2</sup>. Our HTTP interface's one public function is as easy to use as this:

```
#include "http.h"

int main()
{
    std::string page; // to hold the HTML in the web page

    // this next line downloads a web page for you! So easy!
    if (HTTP().get("http://wikipedia.org/wiki/Bald", page))
        cout << page; // prints <!DOCTYPE html PUBLIC ..."
    ...
}
```

Note that you don't need to declare an HTTP variable. The call above looks as if it calls a function named HTTP, then calls a get member function on what it returns.

---

<sup>2</sup> By agreeing to use our HTTP code for project #4, this license entitles Carey&David to a 20% cut of all profits.



## The Tokenizer Class

We provide a Tokenizer class for you to use in your program to simplify the process of tokenizing strings. Tokenizing is the process of breaking up a string into substrings, where each substring is separated by a specific set of divider characters, such as spaces, commas, periods, etc.

For example, if we tokenize the string “This is a test. Really!” with a separator of “ ” (space), we would end up with the following tokens:

“This”  
“is”  
“a”  
“test.”  
“Really!”

On the other hand, if you tokenized the same string with separators of “ .!” (space, period and exclamation), then you would end up with the following tokens:

“This”  
“is”  
“a”  
“test”  
“Really”

Here is the class declaration:

```
class Tokenizer
{
    public:
        Tokenizer(const std::string& text, std::string separators);
        bool getNextToken(std::string& token);
};
```

Here’s how you’d use the class:

```
string s = "This is a test. Really!";
Tokenizer t(s, " ,!.\""); // space, comma, exclamation point,
                        // period, and quote are separators

string token;
while (t.getNextToken(token))
    cout << "token: " << token << endl;
```

This program would print:

token: this  
token: is  
token: a

token: test  
token: Really

You may use this class anywhere in your program where tokenization is required. This class is defined in the provided.h file (which we provide to you ☺).

## Ok, so what is it I have to do?

You have to build a total of seven new classes and a main function for this project. Below is a description of each new class and its purpose:

### ***The MyHashMap Class***

You must build a templated class called MyHashMap that lets you associate C++ string objects to any other object or scalar type of your choosing.

***Any time you would otherwise need an STL map, unordered\_map, or hash\_map to implement a table in another part of of your project, you MUST use your MyHashMap class to implement the map/unordered\_map/hash\_map instead.***

***You must NOT use the STL map, unordered\_map, or hash\_map classes, anywhere in this project. You must NOT use any STL containers (e.g., vector) to implement your MyHashMap class. (You may use the STL vector, list, stack, queue, or set containers in any of your other classes if you like.)***

Note: If you'd like to use the STL map or unordered\_map to implement your MyHashMap early on to get your project up and running quickly, feel free to do so. Just make sure to remove this code before submitting your final project (if you have time to finish your MyHashMap class).

Here's how you might use this MyHashMap class:

```
#include "MyHashMap.h"

int main()
{
    MyHashMap<int> nameToAge;

    nameToAge.associate("Carey", 40); // first arg is always a string
    nameToAge.associate("David", 41);
    nameToAge.associate("King Tut", 3352); // King Tut is a mummy

    int* result = nameToAge.find("Carey");
    if (result != NULL)
        cout << "Carey is " << *result << " years old.\n";
    else
        cout << "Carey was not found in our map!\n";
}
```

```

        // enumerate all items in our map
        string name;
        int* age = nameToAge.getFirst(name);
        while (age != NULL)
        {
            cout << name << " is " << *age << " years old.\n";
            age = nameToAge.getNext(name);
        }
        cout << "Finished printing all items in our map.\n";
    }
}

```

In the above example, we are mapping a name (string) to the person's age (an integer). You could just as easily have mapped the name of a shape to a shape pointer. For example:

```

MyHashMap<Shape*> x;

Circle* c = new Circle(5);           // Circle is a subclass of Shape
Square* s = new Square(4);           // So is Square

x.associate("circle", c);
x.associate("square", s);
...

```

Your MyHashMap class must have the following public interface, and you must not add any additional public items:

```

template<typename ValueType>
class MyHashMap
{
public:
    MyHashMap( # );    // # must work with zero parameters
                       // but also accepts one int parameter

    ~MyHashMap();
    void associate(std::string key, const ValueType& value);
    const ValueType* find(std::string key) const;
    ValueType* find(std::string key);
    ValueType* getFirst(std::string& key);
    ValueType* getNext(std::string& key);
    int numItems() const;
};

```

You must implement your MyHashMap using a hand-written *open hash table*. Your MyHashMap constructor function must be able to accept an integer parameter that specifies the number of buckets in the hash table. If the user does not pass in a parameter for this, then it defaults to 1,000,000.

MyHashMap's associate method is responsible for adding a new item into the hash table, as well as replacing the existing value for a key that is already in the hash table.

The find method always accepts a string as its first parameter. If the method finds the passed-in key within the map, then it returns a pointer to the value associated with that key; otherwise, it returns a NULL pointer. This pointer can subsequently be used to examine the value associated with this key. If the map is allowed to be modified, then the pointer can be used to modify the value associated with the key directly within the

hash table. (The second overload of the find method enables this. Using a little C++ magic, we have implemented it in terms of the first overload of find, which you must implement.)

For example, you could use your MyHashMap to map strings to Circle structures:

```
struct Circle
{
    double x, y;
    double radius;
};

void someFunc()
{
    MyHashMap<Circle> mhm;    // maps strings to Circle objects

    Circle c;
    c.x = 10;
    c.y = 20;
    c.radius = 100;

    mhm.associate("mycircle", c); // associates the string "mycircle" with a Circle
                                // at 10,20 with a radius of 100

    ...

    Circle* ptr = mhm.find("mycircle");

    if (ptr != NULL)
    {
        // we found the Circle object associated with the key "mycircle"
        cout << "Our circle has a radius of: " << ptr->radius << endl;
        // let's change our radius inside the map now!
        ptr->radius = 50;
        // now "mycircle" maps to a circle with a radius of 50 instead of 100
    }
    else
        cout << "There was no value associated with \"myCircle\"\n";
}
```

The `getFirst` method accepts one parameter, whose contents will be set by the method if there is at least one item in the hash table. If there are no items in the hash table, then this method must return `NULL` and must not change its parameter. On the other hand, if there is at least one item in the hash table, then your `getFirst` method must fill in the key parameter with the key for one of the items, and must return a pointer to the value associated with that key. This method thus finds the first (however you want to define first) item in the hash table and returns its key/value pair to the user. After this method has been called, if it returns a non-`NULL` pointer (indicating that there was at least one item in the hash table), then the user of the class may call the `getNext` method one or more times to get the remaining items, if any, in the hash table.

The `getNext` method also accepts one parameter, whose contents will be set by the method if there is at least one remaining mapped item in the hash table. During each call to this method, it must iterate to the next active item in the hash table (skipping past all empty buckets) and return the key/value pair from that item to the user via its parameter and return value. If there are no remaining items in the hash table, then this method must return a `NULL` pointer and must not change its parameter.

There are no requirements on the order in which items must be returned by the `getFirst/getNext` methods. In other words, you may iterate through the active items in your hash table in any order you like.

The `numItems` method must return the number of active items in your hash table that have been associated using the `associate` method. It must not return the total number of buckets in the hash table, but rather the total number of active associations in the table.

Regardless of how you implement your `MyHashMap` class, it must meet the following requirements:

1. A call to either the `getFirst()` or the `getNext()` method must set the key parameter and return the value pointer without using loop statements (`while`, `do-while`, `for`, `goto`, etc), without using recursion, and without calling any other function(s) that use loops or recursion. (Hint: As you know, items inserted into a hash table are distributed randomly across the buckets, so you might think that to find each item in the hash table, you'd have to loop through every empty bucket until you find a valid entry. But, in fact, by creatively adding a second linked list into your hash table bucket nodes, you can improve the basic hash table implementation, enabling you to rapidly iterate through only those items that were inserted).
2. If  $N$  items were inserted into your `MyHashMap`, your destructor must not perform more than  $N$  operations to destroy a `MyHashMap` object that contains exactly  $N$  entries. For example, if your hash table has 10,000 buckets but only 5 inserted items, your destructor must loop only 5 times to destroy the items. (This is basically solved by the same solution to problem #1 above)
3. If the user of your `MyHashMap` class inserts an item (using the `associate` method), then the behavior of the next call to `getNext()` is undefined if there is no intervening call to `getFirst()`, so if the user calls `getNext()` after inserting an item (without calling `getFirst()` first), your program may perform any behavior it likes, including crashing.
4. If the user calls the `getNext()` method without first calling the `getFirst()` method, then the behavior of the `getNext()` method is also undefined.
5. ***The final, submitted version of your `MyHashMap` class must not use any STL containers if you want full credit on this part of the project.*** However, you may find it useful to implement your `MyHashMap` class using the STL `map` or `unordered_map` type initially so you can get up and running quickly.

## ***The ExtractLinks Class***

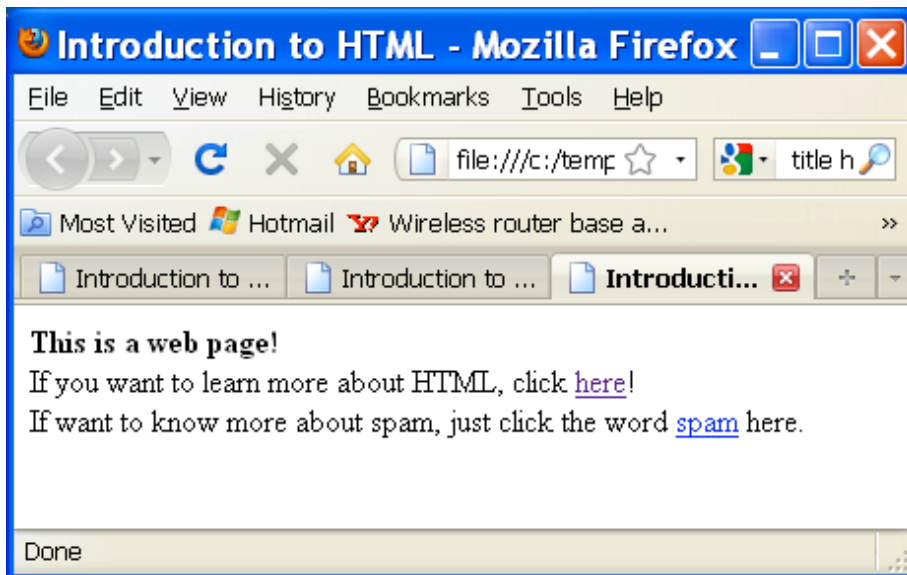
The second class you have to build is one that can take a web page as input (as a really long C++ string) and can locate and extract every link from the page. Web pages are encoded in the Hyper Text Markup Language (HTML), which while not as complex as C++, is a programming language in its own right.

Each web page contains both *tags* and textual content. Tags are special HTML programming commands that tell the browser how to render a web page: where to put images, what links to other pages to include in the page, etc. Many tags come in pairs, like `<html>` and `</html>`, which delineates the start and end of a web page. Another example of a tag is the bold command, `<b>``this is in bold``</b>` which is used to change the text in between the two tags to a **bold** font. Similarly, the `<br/>` tag indicates that a newline should be placed in the webpage (this is a special one – it doesn't have a corresponding `</br>`).

A typical web page might look like this:

```
<html>
<head>
<title>Introduction to HTML</title>
</head>
<body>
<b>This is a web page!</b> <br/>
If you want to learn more about HTML,
click <a href="http://www.wikipedia.com/wiki/html">here</a>! <br/>
If want to know more about spam, just click the word
<a href="http://www.wikipedia.com/wiki/spam">spam</a> here.
</body>
</html>
```

And this is what it would look like in your browser:



As you can see above, there is an anchor tag `<a>` with an HREF attribute:

```
<a href="http://www.wikipedia.com/wiki/html">here</a>
```

Such tags are used to include a link in the web page to another web page. In the above example, the tag indicates that if the user clicks the purple highlighted “here” in the web

browser, this will leave the “Introduction to HTML” page and switch to the Wikipedia page that has an introduction to HTML.

You are to build an `ExtractLinks` class that can parse through the HTML of a web page, locate each link, and return the URL associated with it to the user. Now, as it turns out, there are many ways to encode a link to another web page in HTML – and it would be near impossible for you to identify all such links in the two weeks you have to do this project. Therefore, for the purposes of this assignment, you may assume that you can identify the start of a link by searching for the following text (in either upper, lower or mixed case):

```
<a href=
```

You can identify the end of the link by searching for the first occurrence of the following character appearing after the starting sequence:

```
"
```

If there are links within a web page that do not meet these exact requirements, you are not required to extract these links and return them to the user.

What is the *required* interface of the `ExtractLinks` class? Let’s see:

```
class ExtractLinks
{
    public:
        ExtractLinks(const std::string& pageContents);
        ~ExtractLinks();
        bool getNextLink(std::string& link);
};
```

The `ExtractLinks` class has a constructor that accepts one string argument. The argument should contain the complete text of the web page you downloaded off of the internet, e.g.:

```
std::string myWebPage = "<html><head><title>Introduction to HTML...";
ExtractLinks ext(myWebPage);
```

You may then call the `getNextLink` function one or more times to retrieve the links from the webpage. The first time you call `getNextLink`, it should place the webpage’s first linked URL (if any have the appropriate format specified above) in its *link* parameter, and then return true. Each subsequent call to the `getNextLink` function should get the next link from the page, place it in the *link* parameter and return true. If there are no more links to get, your `getNextLink` functions should return false. Here’s how you might use the `ExtractLinks` class:

```
std::string link;
while (ext.getNextLink(link))
    cout << "Another link: " << link << endl;
```

For our earlier webpage, this little program snippet should print:

```
Another link: http://www.wikipedia.com/wiki/html
Another link: http://www.wikipedia.com/wiki/spam
```

To ensure that you do not change the interface to the `ExtractLinks` class in any way, we will implement that class for you. But don't get your hopes up that we're doing any significant work for you here: Our implementation is to simply give `ExtractLinks` just one private data member, a pointer to an `ExtractLinksImpl` object (which you can define however you want in `ExtractLinks.cpp`) The member functions of `ExtractLinks` simply delegate their work to functions in `ExtractLinksImpl`.<sup>3</sup> You still have to do the hard work of implementing those functions.

Once you've finished your implementation of this class, test it with several sample web pages. (You can view the source code for a web page by selecting View / View Source in many browsers, or by hitting Ctrl-U in FireFox.) If you can't get your `ExtractLinksImpl` class to work, don't worry – we'll also provide our own working version when we test your program if yours doesn't work perfectly. Your class must not print any output at all to cout, but you may print out any debug output to cerr if desired.

```
cerr << "blah, blah, blah";
```

**Other than `ExtractLinks.cpp`, no source file that you turn in may contain the name `ExtractLinksImpl`.** Thus, your other classes must not directly instantiate or even mention `ExtractLinksImpl` in their code. They may use the `ExtractLinks` class that we provide (which indirectly uses your `ExtractLinksImpl` class).

Ok, so now we know how to download a web page. And you have a class that can extract all of the links from the web page (in the order that those links were found in the page). Now what?

## ***The Document Class***

As the name implies, the `Document` class is used to represent a web document (e.g., a web page). Not only does `Document` hold the URL and contents of a web page, but it also provides a means to efficiently determine if the web page contains a specific word. Furthermore, it provides an efficient means to obtain all unique words in the document. Here is the required interface for the `Document` class:

```
class Document
{
public:
    Document(std::string url, const std::string& text);
    ~Document();
    std::string getURL() const;
```

---

<sup>3</sup> This is an example of what is called the [pimpl idiom](#) (from "pointer-to-implementation").



```

    bool contains(std::string word) const;
    bool getFirstWord(std::string& word);
    bool getNextWord(std::string& word);
};

```

As you can see, when you construct a Document object, you must specify the URL of the document (e.g., “http://www.carey.com”) as well as the full HTML text of the web page that was downloaded from the Internet, e.g., “<html><body>This is a web page</body></html>”.

The getIdentifier method must return the URL of the Document’s web page.

The contains method must return true if a particular word is present in your Document’s text. A word is a sequence of letters and/or digits. This function must be case-insensitive – so if your document contains the text “<html>I love CS32</html>” and the user called the contains method with a parameter of “cS32” or “Cs32” or “CS32”, the method must return true. This method must run in  $O(1)$  time, meaning that if there are  $N$  words in your Document, the contains method must complete its operation in a constant number of steps which is not dependent on  $N$ .

The getFirstWord and getNextWord methods can be used to obtain a sequence of unique words from the Document. These methods must never return the same word more than once, even if a given word is present multiple times in the Document. For example, if your document contains “I really like to say I like you”, then the words returned by getFirstWord and successive calls of getNextWord might be “I”, “really”, “like”, “to”, “say”, “you” (in any order), but not something like “I”, “really”, “like”, “to”, “say”, “I”, “like”, and “you”. Each of these methods must run in  $O(1)$  time, meaning that if there are  $U$  unique words in your Document, these methods must complete their operation of getting a word in a constant number of steps which is not dependent on  $U$ .

As with the other classes you must write, the real work will be implementing the auxiliary class DocumentImpl in Document.cpp. Once you’ve finished your class, test it. If you can’t get your DocumentImpl class to work, don’t worry – we’ll provide our own working version when we test your program if yours doesn’t work. Your class must not print any output at all to cout, but you may print out any debug output you like to cerr.

**Your DocumentImpl class MUST use your MyHashMap class to implement the contains, getFirstWord and getNextWord methods efficiently.**

**Other than Document.cpp, no source file that you turn in may contain the name DocumentImpl.** Thus, your other classes must not directly instantiate or even mention DocumentImpl in their code. They may use the Document class that we provide (which indirectly uses your DocumentImpl class).

## The Crawler Class

Your Crawler class is responsible for taking a URL for a starting (seed) web page and then downloading that web page, extracting its links, and downloading those web pages, extracting their links, downloading those pages, etc.

Here's the required interface for your Crawler class:

```
class Crawler
{
    public:
        Crawler(std::string seedSite);
        ~CrawlerImpl();
        Document* crawl();
};
```

As you can see, when you construct a Crawler object, you provide one parameter, `seedSite`, which is the starting URL (i.e., a seed URL) from which the crawler should start crawling.

```
CrawlerImpl cr("http://www.wikipedia.org/wiki/DNA");
```

For instance, the above line would construct a new Crawler object called `cr` which will start crawling on Wikipedia's DNA page.

Once you've constructed a Crawler object, you can repeatedly call the `crawl` method. The `crawl` method must do the following:

1. It determines the next URL that it should crawl to.
2. It connects over the Internet to the specified URL and retrieves this webpage from the host site. (Use our HTTP interface for this.)
3. It extracts all of the links on the webpage and adds *those that have the same domain name as that of the seed web site* (\*), in the order they were found in the web page, to its group of URLs to be processed.
4. It then dynamically creates a new Document object, based on the contents of the just-retrieved web page, and returns a pointer to this new Document object to the caller of the `crawl` method.

\* Imagine that we begin crawling on this seed web page of "www.foobar.com/root":

```
<html>
Some text...
<a href="http://www.foobar.com/bletch">
Some more text...
<a href="http://www.goober.com/oof">
Some more text...
<a href="http://www.foobar.com/barf">
Some more text...
<a href="http://www.wikipedia.com/foobar">
```

```
Some more text...
<a href="/bletch">
</html>
```

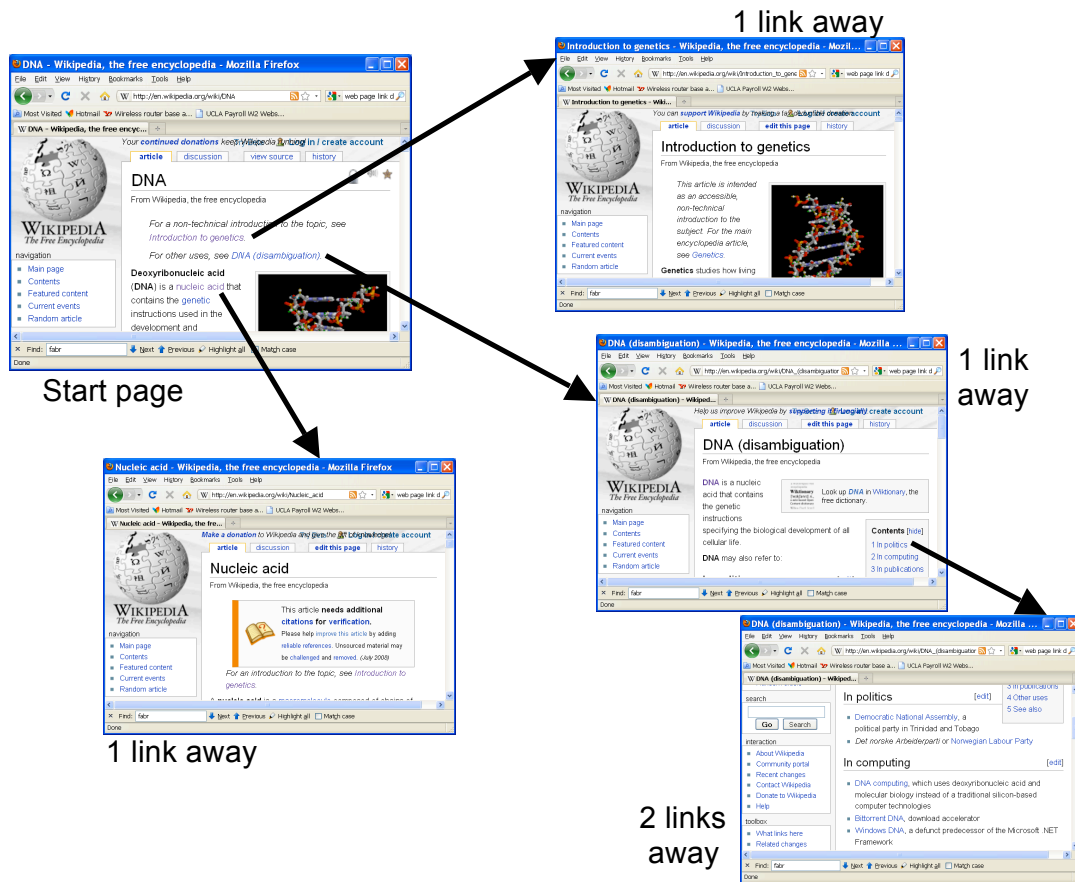
Your crawl method would find the following links on this page:

1. <http://www.foobar.com/bletch>
2. <http://www.goober.com/oof>
3. <http://www.foobar.com/barf>
4. <http://www.wikipedia.com/foobar>
5. </bletch>

Links #1 and #3 above have the same domain name ([www.foobar.com](http://www.foobar.com)) as that of the seed page URL ([www.foobar.com/root](http://www.foobar.com/root)), so they would be crawled normally. Links #2 and #4, however, have different domain names ([www.goober.com](http://www.goober.com) and [www.wikipedia.com](http://www.wikipedia.com)) than that of the seed site ([www.foobar.com](http://www.foobar.com)), so they would not be crawled/retrieved by the crawler. Similarly, your crawler would ignore link #5 because it is not explicitly listed as being on the [www.foobar.com](http://www.foobar.com) web site. Your crawler must ignore all such links that would either lead the crawler to a different website, or that do not have the FULL seed domain name (e.g., [www.foobar.com](http://www.foobar.com)) in the link.

Your crawl function must return pages in the order of a *level-order (i.e., breadth first) tree traversal*, starting at the page specified in the constructor. Here's an example of how the crawl function might be used:

```
Crawler cr("http://www.wikipedia.org");
for (;;)
{
    Document* curDoc = cr.crawl();
    if (curDoc == NULL)
        break;
    cout << "Got a valid document object for URL: "
         << curDoc->getURL() << endl;
}
```

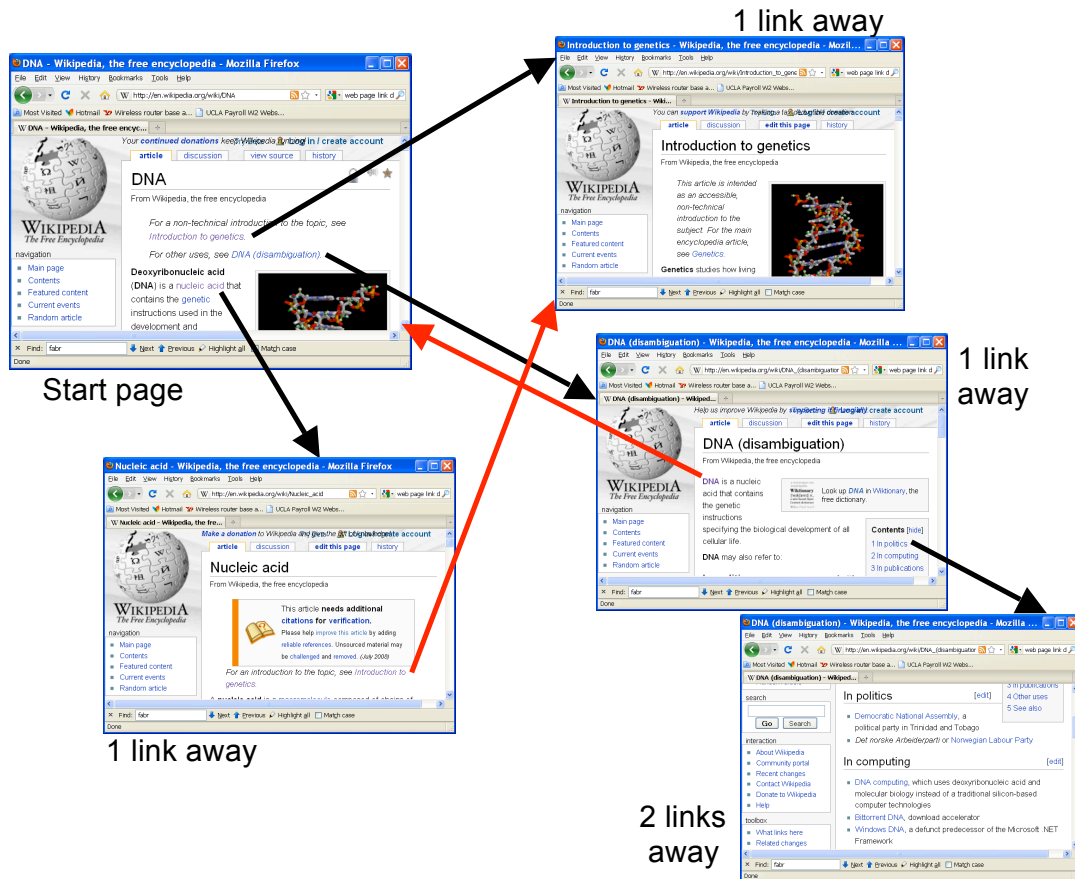


So, for example, if you ran the snippet above on the set of pages above (assuming that the pages contain only the links shown with arrows above), the crawl method should return Document objects for URLs in the following order:

<http://www.wikipedia.org/wiki/DNA>  
[http://www.wikipedia.org/wiki/Introduction\\_to\\_genetics](http://www.wikipedia.org/wiki/Introduction_to_genetics)  
[http://www.wikipedia.org/wiki/DNA\\_disambiguation](http://www.wikipedia.org/wiki/DNA_disambiguation)  
[http://www.wikipedia.org/wiki/Nucleic\\_acid](http://www.wikipedia.org/wiki/Nucleic_acid)  
[http://www.wikipedia.org/wiki/In\\_politics](http://www.wikipedia.org/wiki/In_politics)

Specifically, your crawl method should first return a Document object for the starting webpage (the starting URL is specified in the constructor); then, subsequent calls to it should return all web pages that are linked-to that are exactly 1 link away from the start webpage, in the order those links were found in the HTML of the starting page. Once the links from the starting page have been processed, then subsequent calls to crawl should return all web pages that are exactly 2 links away from the start page, etc. Notice that the In\_politics page, which is two clicks away from our start DNA page, was returned by crawl only after the Introduction\_to\_genetics, DNA\_disambiguation and Nucleic\_acid pages were first visited, since these pages were just one link/click away from our starting DNA page.

Unlike a binary tree, in which each node has exactly two children and nodes never link back to their ancestors, web pages can have all sorts of complicated linking patterns. For example, it is possible that two pages both link to the same third page, or a page may link back to another page which linked to it:



For example, in the set of pages above, both the starting DNA page and the Nucleic Acid page both link to the Introduction to Genetics page. Moreover, the DNA Disambiguation page links back to the starting DNA page.

Your CrawlerImpl must ensure that no web page is ever crawled twice; that is, the crawl method may never return a Document for a web page that it has already processed before. If one of the links on a page is one that has already been visited, it must simply be ignored and the page it links to not revisited.

Thus, given the link structure in the diagram above, we would expect five successive calls to your getNextPage function to return the same set of links in the same order as if the links in red weren't present:

<http://www.wikipedia.org/wiki/DNA>

[http://www.wikipedia.org/wiki/Introduction\\_to\\_genetics](http://www.wikipedia.org/wiki/Introduction_to_genetics)  
[http://www.wikipedia.org/wiki/DNA\\_disambiguation](http://www.wikipedia.org/wiki/DNA_disambiguation)  
[http://www.wikipedia.org/wiki/Nucleic\\_acid](http://www.wikipedia.org/wiki/Nucleic_acid)  
[http://www.wikipedia.org/wiki/In\\_politics](http://www.wikipedia.org/wiki/In_politics)

Once your crawl function runs out of pages, it should return a value of NULL, indicating that no more pages are available.

Unfortunately, the Internet is far from perfect – links are often broken or point to old/removed web pages, so you should expect that your crawl function may occasionally run into a non-existent URL. If, for whatever reason, a page cannot be reached or retrieved by our HTTP class, then the crawl function must simply ignore that page, skipping over it and returning a Document object for the next available page. Your crawl function should not return NULL unless there are no more links left to crawl.

As with the other classes you must write, the real work will be implementing the auxiliary class CrawlerImpl in Crawler.cpp. Once you've finished your class, test it! If you can't get your CrawlerImpl class to work, you should worry – this is an important part of the project. Your class must not print any output at all to cout, but you may print out any debug output to cerr if you desire.

Your Crawler class will likely need to make use of the functionality provided by the other classes. However, it must never instantiate or use your other Impl classes directly; instead, it should use the classes that don't have Impl in their names (e.g., it may use ExtractLinks, but not ExtractLinksImpl). This will allow us to test your Crawler class even if you can't get one or more of your other classes, like ExtractLinksImpl, to work, since we can easily provide our own ExtractLinksImpl to replace your buggy/incomplete version.

**Other than Crawler.cpp, no source file that you turn in may contain the name CrawlerImpl.** Thus, your other classes must not directly instantiate or even mention CrawlerImpl in their code. They may use the Crawler class that we provide (which indirectly uses your CrawlerImpl class).

## ***The Rule Class***

The Rule class is responsible for representing a single rule out of the provided rules data file. Your program will ultimately need to have many Rule instances, one for each rule in the rules data file.

Here is the interface for the Rule class:

```
class Rule
{
    public:
        Rule(std::string ruleText);
```

```

~Rule();
Rule(const Rule& other);
Rule& operator=(const Rule& other);
std::string getName() const;
double getDollarValue() const;
int getNumElements() const;
std::string getElement(int elementNum) const;
std::string getAd() const;
bool match(const Document& doc) const;
};

```

To construct a Rule object, you must pass in a rule line from the rules data file. The format of a rule line is described in the *Details of the Rules File* section above; a rule might look something like this:

```
rule-000000cc 0.75 FOXSKIN VIRILIZE BREEDER & | > Buy VIRILIZE now and save!
```

Once you have constructed a Rule object, you can obtain the name of the rule by calling the getName method. For the example rule above, this would return "rule-000000cc".

You may call the getDollarValue method to determine the dollar value of the rule. For the example rule above, this method would return .75.

You may call the getNumElements method to determine the number of elements (i.e., terms and operators) in the matching part of the rule. For the example rule above, the matching part of the rule would be "FOXSKIN VIRILIZE BREEDER & |", so this method should return 5, since there are 5 space-separated elements ("FOXSKIN", "VIRILIZE", "BREEDER", "&" and "|") in the matching part of the rule.

The getElement method returns the specified element (starting at 0) in the matching part of the rule. For example, if you called getElement with a value of 1, it would return "VIRILIZE". If you call getElement with a value that is too small or too large, then it should return an empty string ("").

The getAd method returns the text of the advertisement (the human readable text the user would see). For the example above, this method would return "Buy VIRILIZE now and save!"

Finally, the match method is responsible for determining if a given Document object, which is passed in as a parameter, matches the Rule's requirements. For example, if the Document's contents were "I like the words FOXSKIN and virilize but I don't like the word BREEDER", then the rule we showed above would trigger, and the match method would return true. Notice that the match method performs case-insensitive matching. That is, even though the rule specifies a capitalized version of "VIRILIZE", it will match a document with this word in any form, such as "virilize" or "ViRiLiZe". Also note that your matching expressions are encoded in a postfix form.

Your match method must be extremely fast. Assuming the Document object passed in has N words (of which U are unique), and assuming your rule has T elements, your match

method must be  $O(T)$ . That is, your match method must not have runtime that is proportional to the number of words ( $N$  or  $U$ ) in the Document, but only proportional to the number of elements in the Rule.

When constructing a Rule, if the ruleText is not properly formed (e.g., there are too few items on the line, or there's not a valid postfix expression), the constructor must ensure that any later call to `getNumElements` returns 0 and any later call to `match` returns false. (We don't care what the result of calling any of the other member functions would be.) This is an awkward way of signalling an error in a constructor, but we haven't covered exceptions in this course.

Here are some sample documents and sample rules:

Documents:

D1: "I like spam"  
D2: "I LIKE green eggs"  
D3: "I think CS32 professors are smart"  
D4: "I think CS32 professors is smart"

Rules:

R1: "Spam EGGS | like &"  
R2: "eggs"  
R3: "I like &"  
R4: "professors are smart & &"

Results:

R1 and R3 trigger on D1  
R1, R2, and R3 trigger on D2  
R4 triggers on D3  
None of the rules trigger on D4

As with the other classes you must write, the real work will be implementing the auxiliary class `RuleImpl` in `Rule.cpp`. Once you've finished your class, test it! Your class must not print any output at all to `cout`, but you may print out any debug output to `cerr` if you desire.

**Other than `Rule.cpp`, no source file that you turn in may contain the name `RuleImpl`.** Thus, your other classes must not directly instantiate or even mention `RuleImpl` in their code. They may use the `Rule` class that we provide (which indirectly uses your `RuleImpl` class).



## The Matcher Class

Your Matcher class is responsible for orchestrating the matching of all rules against a particular document. Since you may have hundreds of thousands or millions of rules that you'll want to use (provided by all of your advertisers), you need some way to efficiently check a given Document to see which of those millions of rules match.

Here's its public interface:

```
class Matcher
{
    public:
        Matcher(std::istream& ruleStream);
        ~Matcher();
        void process(Document& doc, double minPrice,
                     std::vector<Match>& matches) const;
};
```

The constructor loads the potentially hundreds of thousands of rules from the already-opened input source supplied as its argument (the format of the rules is shown in the *Details of the Rules File* section of this document). The *Input from Files* writeup on the CS32 home page explains the parameter of type `istream&`.

For rule lines from the input source that represent valid rules, the constructor should add Rule objects to an efficient internal data structure. We are placing no performance restrictions on your constructor; it may take as long as it likes (within reason) to load the rules and construct data structures required for efficient matching in your process method.

The process method accepts a Document object (e.g., for a web page) and a minimum advertisement price value (e.g. .75 for 75 cents) as inputs. It must determine which of your potentially hundreds of thousands of rules match this Document AND also have a dollar value which is greater than or equal to the minPrice parameter. For each such matching Rule, you must append a Match object with the proper data to the end of the matches vector parameter.

Here is what a Match looks like:

```
struct Match
{
    std::string url;
    std::string ruleName;
    double dollarValue;
    std::string ad;
};
```

As you can see, a given Match contains a URL (e.g., "www.foo.com/bar"), a rule name (e.g. "rule-00000576"), the advertisement (e.g. "Buy tomatoes today! Click here!"), and the dollar value that the advertiser will pay to WSSA if a user were to click on this ad).

Your process method must not change/remove any existing items within the matches argument; instead, it must append any new matches from the current Document onto the end of the matches vector.

Regardless of how you implement your Matcher class, it MUST meet the following requirement:

Your process method must apply only *applicable* rules to a document; it must not apply *all* rules to a given document unless all rules are *applicable*. Our definition is that a Rule is *applicable* to a particular Document if and only if the Rule's postfix matching expression contains at least one term that is also contained in the Document.

For instance, consider the following document:

*“The rain in Spain falls mainly on the plain.”*

And the following four rules:

```
rule-0000002e 0.36 smallberg Nachenberg & > Save 10% on professors if you buy today!  
rule-0000004a 0.96 falls Gordon booger & |> This is a random ad! Click on it!  
rule-000000cc 0.75 spain Portugal italy & & > Thinking of going to Spain, Portugal and Italy? Click here!  
rule-000000d1 0.42 goat milk tasty & & > Like goat milk? Click here
```

Your matcher function must apply only Rules 4a and cc to this Document. It must not apply the other rules since there is no way they could possibly match the Document (these other rules have no terms whatsoever that appear in the Document).

Practically, this means that you must set up some efficient data structures within your class that allow you to quickly determine which rules could possibly match a Document, and your process method must apply no rules other than those possibly-matching rules to the Document.

So, if you find your code doing something like this, you'll get 0 points for this part of the project:

```
void process(Document& doc, double minPrice, std::vector<Match>& matches) const  
{  
    // assume allRules contains all rules loaded from rules file  
    for (int i = 0; i < allRules.size(); i++)  
    {  
        if (allRules[i].match(doc) && allRules[i].getDollarValue() >= minPrice)  
        {  
            ... // add your match to the matches vector  
        }  
    }  
}
```

Notice how the code above goes through every single Rule and tries to match each Rule against the current Document. This violates the requirement above. Instead, one way your code could meet the requirement is to do something like this:

```
void process(Document& doc, double minPrice, std::vector<Match>& matches) const
{
    std::vector<Rule*> applicableRules =
        findTheSubsetOfApplicableRulesThatApply(doc);
    // applicableRules now holds pointers only to rules that have at least
    // one keyword that is also found in doc.

    for (int i = 0; i < applicableRules.size(); i++)
    {
        if (applicableRules[i]->match(doc) &&
            applicableRules[i]->getDollarValue() >= minPrice)
        {
            ... // add your match to the matches vector
        }
    }
}
```

Why do we impose this requirement? Well, we expect to have hundreds of thousands of rules, but most words appear in only a handful of rules, and most rules have only a handful of words. Since even a long document has only a few thousand unique words, the vast majority of rules have no chance of matching the document because they don't share even a single word with the document. It takes time to see if a document matches a rule: We have to evaluate the match expression, look up words in the document, etc. To speed up our program, we'd like to apply as few rules as possible to each document.

As with the other classes you must write, the real work will be implementing the auxiliary class `MatcherImpl` in `Matcher.cpp`. Once you've finished your class, test it! Your class must not print any output at all to `cout`, but you may print out any debug output to `cerr` if you desire.

**Other than `Matcher.cpp`, no source file that you turn in may contain the name `MatcherImpl`.** Thus, your other classes must not directly instantiate or even mention `MatcherImpl` in their code. They may use the `Matcher` class that we provide (which indirectly uses your `MatcherImpl` class). **Your `MatcherImpl` class must NOT use the STL `map`, `unordered_map`, or `hash_map` classes. If you need a `Map`, you must use your `MyHashMap` class for this purpose.**

## The AdHunter Class

Now things are getting interesting! Your AdHunter class is responsible for tying your entire ad hunting system together in a single, cohesive class.

```
class AdHunter
{
    public:
        AdHunter(std::ifstream& ruleStream);
        ~AdHunter();
        void addSeedSite(std::string site);
        int getBestAdTargets(double minPrice, int pagesPerSite,
                           std::vector<Match>& matches);
};
```

The constructor loads the rules from the already-opened input source supplied as its argument. It will probably delegate most of its work to Matcher's constructor.

The addSeedSite method is called to add one or more seed websites (e.g., "http://yahoo.com" or "http://symantec.com") to your AdHunter object. The user may call this method multiple times to add multiple seed websites to crawl to check for advertisement matches. This method should keep track of all added sites in some efficient data structure that doesn't have a fixed size (since there could be thousands of seed web sites), like a list or a vector.

The getBestAdTargets method is responsible for crawling through all of the websites provided by previous calls to addSeedSite (using the Crawler class), applying appropriate rules to each appropriate web page from each site (using the Matcher class), and then passing back a collection of Match objects for all rules that matched the crawled web pages **and** met the minimum price requirement. All matches must be placed into the matches vector, and your method must clear the contents of the matches vector when it starts (so if it had items in it before getBestAdTargets started, these would all be cleared out before any new matches were added). The matches must be ordered from most profitable to least profitable (with most profitable matches at the beginning of the matches vector). If two Matches have the same dollar value, then they should be further ordered alphabetically by their URL. If two Matches have both the same dollar value AND the same URL, then they should further be ordered alphabetically by their rule ID.

Your getBestAdTargets method must return the total number of matches (only those that paid enough money) found across all web pages that were crawled.

As with the other classes you must write, the real work will be implementing the auxiliary class AdHunterImpl in AdHunter.cpp. Your class must not print any output at all to cout, but you may print out any debug output to cerr if you desire. **Other than AdHunter.cpp, no source file that you turn in may contain the name AdHunterImpl.**

## Your main() Function

Now that you've built and tested each of your classes, its time to tie them all together into a cohesive program.

You are to write a main function, and any required supporting functions, to do the following in your main.cpp file:

1. Your main function must accept four command line arguments:

```
C:\CS32\PROJ4> proj4.exe rulesFile seedSiteFile maxPages minPrice
```

- The first parameter is the name of a file that contains your rules. The format of the rules data file was described in the *Details of the Rules File* section.
- The second parameter is the name of a file that contains all of the seed site URLs, one per line, e.g.:

```
http://www.yahoo.com  
http://www.myblog.com  
http://www.lasvegas.com  
http://www.goats.com
```

- The third parameter specifies the maximum number of web pages that should be crawled for each seed domain. For example, your program might be asked to crawl up to 100 pages from each of the above domains in search of web pages that match your rules.
- The fourth parameter specifies the minimum dollar value required for a rule to match. For example, we might not want to bother placing ads unless the advertiser (e.g., Reebok) is willing to pay at least \$1.50 per ad.

Here's how it might actually be called:

```
C:\CS32> proj4.exe rules.txt sites.txt 100 1.50 ← enter
```

**If you don't know what command line arguments are, read the *How do Command Line Parameters/Arguments Work?* section below.**

2. If all of the required parameters aren't present, then your main function should write the following reminder and exit:

```
Usage: proj4.exe rulesFile seedSiteFile maxPages minPrice
```

3. Otherwise, your main function (or a supporting function called by main) must attempt to open the rules file. If it fails, it must write the following and exit:

```
Error: cannot open advertisement rules file!
```

4. Otherwise, your main function (or a supporting function called by main) must attempt to open the seed sites file. If it fails, it must write the following and exit:

Error: cannot open seed sites file!

5. Otherwise, your main function (or a supporting function called by main) must load the rules from the rules file. You should create an AdHunter to do this. It must then add each seed URL from the seed sites file to the AdHunter object.
6. Your main function (or a supporting function called by main) should then write:

Crawling...

followed by an empty line. It should then use its AdHunter object to crawl through the websites and locate all appropriate matching ads. When the AdHunter's `getBestAdTargets()` function returns, your program should write

There were XXX matches found that met the minimum price of \$YYY per ad:

followed by an empty line, where XXX is replaced with the number of matches that were found, and YYY with the minimum price required (e.g., 1.50 from the example above). You must print out the YYY value such that it has two digits after the decimal point. For example, printing "1.5" would be incorrect; you must print "1.50" in this case.

7. Your main function (or a supporting function called by main) must then write the list of matching ads (if any) in the order that the AdHunter's `getBestAdTargets` method returned them:

```
Match 0: rule-0000156b triggered on http://www.foo.com/bletch for a value of $1.50
        Advertisement: Buy cheap bananas today!
Match 1: rule-00004123 triggered on http://www.foo.com/bletch for a value of $1.20
        Advertisement: Try adult diapers now
Match 2: rule-00023992 triggered on http://www.foo.com/goof for a value of $1.20
        Advertisement: Click here to grow hair!
Match 3: rule-0417454d triggered on http://www.foo.com/goof for a value of $1.20
        Advertisement: Click here to grow a new arm!
Match 4: rule-00031143 triggered on http://www.foo.com/bletch for a value of $1.10
        Advertisement: Click here if you like CS32!
```

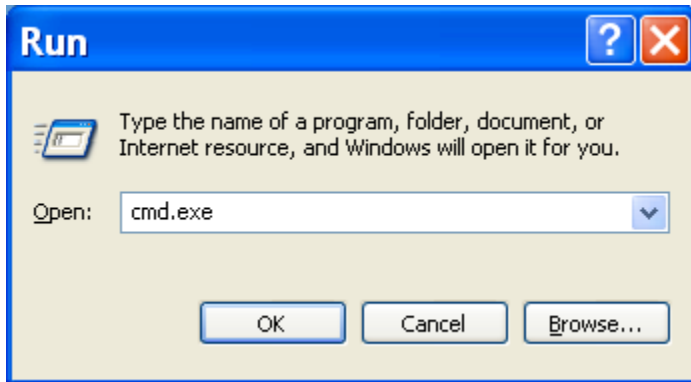
The general format here is:

```
Match 0: ruleId triggered on URL for a value of $value
<tab>Advertisement: textOfAd
Match 1: ruleId triggered on URL for a value of $value
<tab>Advertisement: textOfAd
...
Match n-1: ruleID triggered on URL for a value of $value
<tab>Advertisement: textOfAd
```

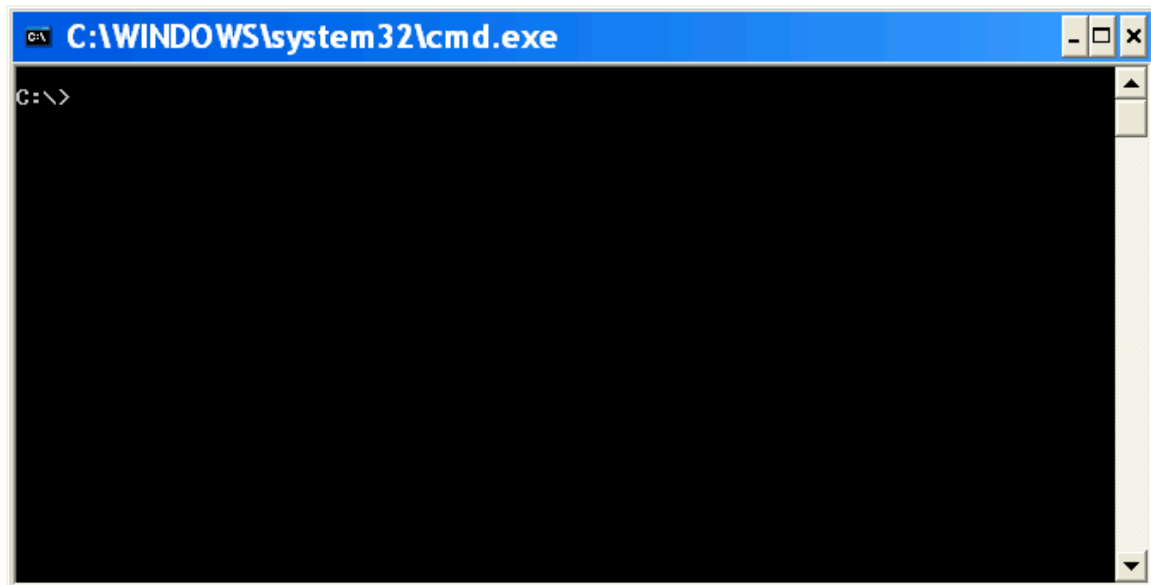
Your main function then terminates.

## How do Command Line Parameters/Arguments work?

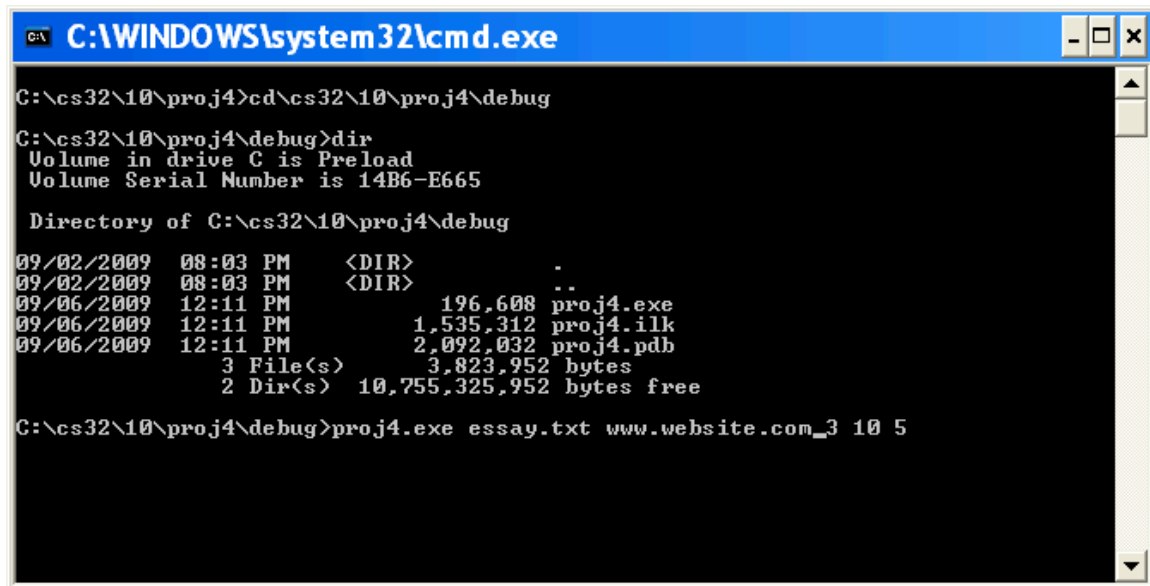
When you write a program, there's often a need for the user of the program to specify some parameters to the program to customize certain aspects of its operation. One way to specify parameters to a program is on the command line, via a UNIX or DOS command shell. You can, for example, launch a Windows command shell by using the Start→Run command, and then typing in `cmd.exe` in the dialog.



This will bring up a box that looks like this.



From the C prompt, you can then change directories, using the “cd” command, to the directory where your EXE is stored and run your program:



```
C:\WINDOWS\system32\cmd.exe
C:\cs32\10\proj4>cd\cs32\10\proj4\debug
C:\cs32\10\proj4\debug>dir
Volume in drive C is Preload
Volume Serial Number is 14B6-E665

Directory of C:\cs32\10\proj4\debug

09/02/2009  08:03 PM    <DIR>          .
09/02/2009  08:03 PM    <DIR>          ..
09/06/2009  12:11 PM             196,608 proj4.exe
09/06/2009  12:11 PM        1,535,312 proj4.ilc
09/06/2009  12:11 PM        2,092,032 proj4.pdb
               3 File(s)        3,823,952 bytes
               2 Dir(s)    10,755,325,952 bytes free

C:\cs32\10\proj4\debug>proj4.exe essay.txt www.website.com_3 10 5
```

When you run a program and specify additional parameters, like “essay.txt” or “www.web site.com”, etc., these parameters are provided to your program via a pair of arguments – `argc` and `argv` - to your main function (which are optional – you only need to include them if you intend for your program to process arguments on the command line):

```
// cmdargs.cpp

int main(int argc, char* argv[])
{
    cout << "There are << argc << " arguments to this program\n";
    cout << "The program's filename is " << argv[0] << endl;
    for (int i = 1; i < argc; i++)
        cout << "Arg " << i << " has a value of " << argv[i] << endl;
    ...
}
```

The `argc` variable is an integer that contains a count of the number of arguments to your program. The `argv` variable is a pointer to an array of pointers to strings (see diagram below). Each element of the `argv` array, e.g. `argv[0]`, `argv[1]`, etc., points to a different argument string.

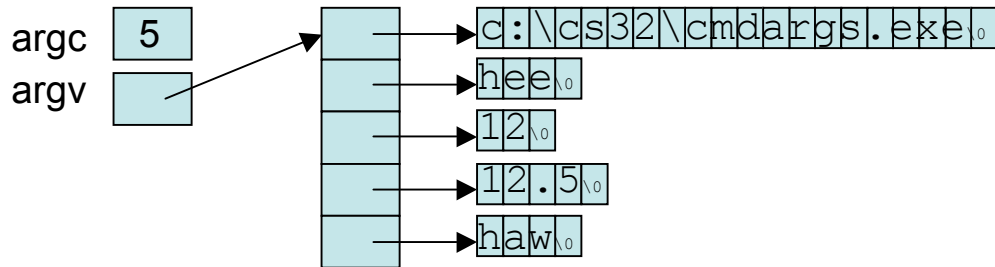
The zero'th argument to every program is the name of the program itself (although Windows may modify it a bit); it is automatically added by the operating system into the argument list before all explicitly provided arguments. (In our example above, it's `c:\cs32\10\proj4\debug\proj4.exe`). The remaining arguments are the explicit arguments that the user typed after the filename, such as “essay.txt”, “www.website.com”, “3”, “10”, “5”. All arguments are passed in the form of C strings. Each argument must be separated by a space on the command line; the command interpreter uses these spaces to identify each parameter.

If we were to run the above source code in the following manner:



```
C:\cs32> cmdargs.exe hee 12 12.5 haw
```

argc and argv would look like this:



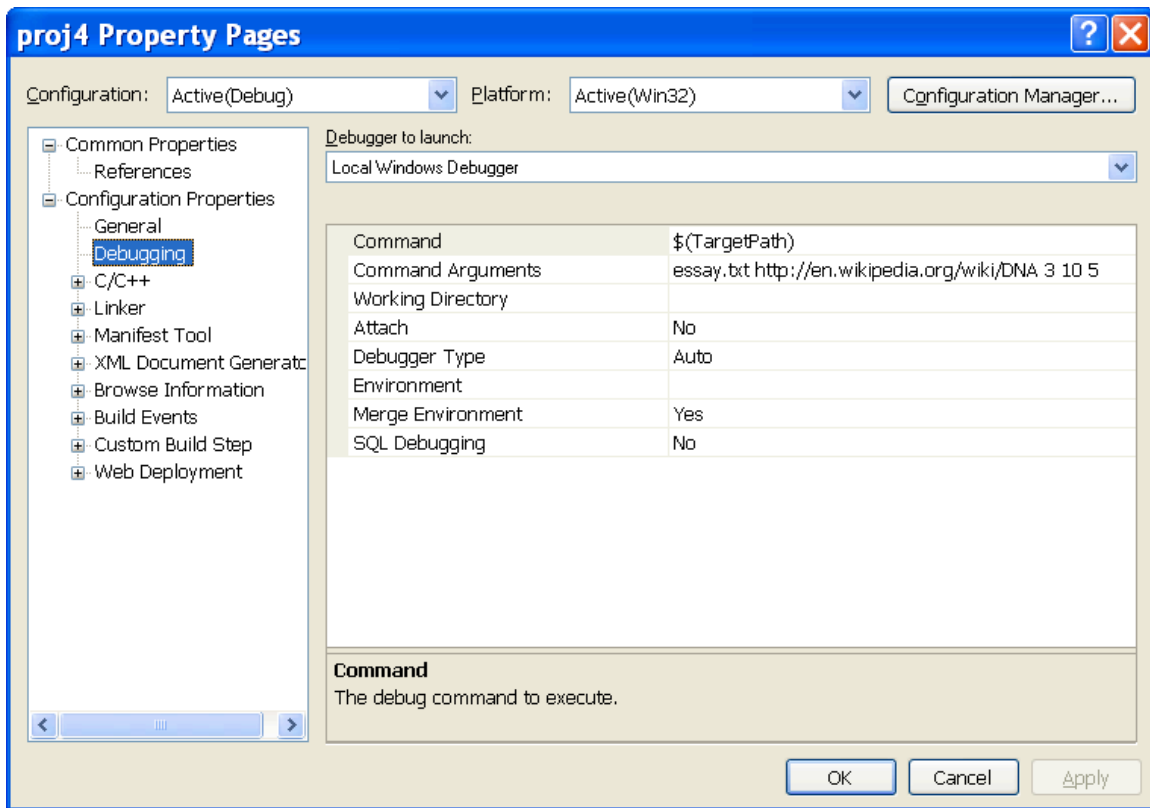
and our simple program would print:

```
There are 5 total args to this program
The program's filename is c:\cs32\cmdargs.exe
Arg 1 has a value of fee
Arg 2 has a value of 12
Arg 3 has a value of 12.5
Arg 4 has a value of fum
```

Note that both string parameters and numeric parameters are passed as C strings to your program, so you will need to convert each argument intended to be a number into a number before you can use it as a number:

```
#include <cstdlib>          // needed for the atoi and atof functions
...
int k = atoi(argv[2]);      // convert C string to int
double d = atof(argv[3]);  // convert C string to double
cout << k*k << " " << d*d << endl; // writes 144 156.25
```

If you would like to specify command-line parameters to your program when testing in the Visual Studio development environment, you can do so by going to the Project menu and selecting your “Proj4 Properties”. Then click on the + next to the “Configuration Properties” item, and click on “Debugging.” You should see a window that looks something like this:



From here, you can type in the command line arguments in the Command Arguments field. In the dialog above, we've already specified the five arguments to the program.

Using Xcode, you select Project / Edit Scheme / Run / Arguments to set the command line arguments.

Under Linux, or under Mac OS X in a Terminal window, to run a command with arguments, you just type them on the command line, as in

```
cmdargs hee 12 12.5 haw
```

## Requirements and Other Thoughts

**Make sure to read this entire section before beginning your project!**

1. In Visual C++, make sure to change your project from UNICODE to Multi Byte Character set, by going to Project → Properties → Configuration Properties → General → Character Set
2. In Visual C++, make sure to add *wininet.lib* to the set of input libraries, by going to Project → Properties → Linker → Input → Additional Dependencies ; otherwise, you'll get a linker error!

3. The entire project can be completed in under 600 lines of C++ code beyond what we've already written for you, so if your program is getting much larger than this, talk to a TA – you're probably doing something wrong.
4. Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. Will you need a map, a stack, a set? How will you use these data structures? Plan before you program!
5. Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
6. You must not modify any of the code in the files we provide you that you will not turn in. We will incorporate the files that you turn in into a project with special test versions of the other files.
7. Your Impl classes (e.g. ExtractLinksImpl, AdHunterImpl) and your main() function must **never directly** use your other Impl classes. They MUST use our provided cover classes instead:

INCORRECT:

```
class AdHunterImpl
{
    ...
    void someFunction(...)
    {
        CrawlerImpl crawler(...);    // BAD!
        ...
    }
};
```

CORRECT:

```
class AdHunterImpl
{
    ...
    void someFunction(...)
    {
        Crawler crawler(...);    // GOOD!
        ...
    }
};
```

8. Make sure to implement and test each class independently of the others that depend on it. Once you get the simplest class coded, get it to compile and test it with a number of different unit tests. Only once you have your first class working should you advance to the next class.
9. Make heavy use of the STL for this project (with the exception of the STL map/hash\_map/unordered\_map classes – you must NOT use the map, hash\_map or unordered\_map classes at all in your solution – use only your MyHashMap class if you need a map). Without the STL, this project would require thousands of lines of code. Many subproblems in this project, such as ordering matches that would otherwise seem like they would require complex algorithms, can be solved trivially with the clever use of STL classes and functions (including the <algorithm> functions like *sort*)!

10. If you decide to use `unordered_set`, the following will `#include` the appropriate header under either Visual C++ or g++:

```
#ifndef _MSC_VER
#include <unordered_set>
using std::unordered_set;
#else
#include <tr1/unordered_set>
using std::tr1::unordered_set;
#endif
```

11. The `std::string` type offers some functions you may find useful:

```
//          1111111112
// 012345678901234567890
string s("No news is good news.");
size_t k = s.find("news"); // k == 3, the position of the first "news"
k = s.find("news", 0); // k == 3, the position of the first "news"
                        // starting at or after position 0
k = s.find("news", 5); // k == 16, the position of the first "news"
                        // starting at or after position 5
k = s.find("news", 17); // k == string::npos, indicating that there was
                        // no "news" starting at or after position 17
string t = s.substr(3, 4); // t is "news", the substring of s of
                        // length 4 starting at position 3
```

If you don't think you'll be able to finish this project, then take some shortcuts. Use the STL map class instead of creating your own `MyHashMap` class if necessary to save time. You can still get a good amount of partial credit if you implement most of the project. Why? Because if you fail to complete a class (e.g., `MatcherImpl`), we will provide a correct version of that class and test it with the rest of your program. If you implemented the rest of the program properly, it should work perfectly with our version of the class and we can give you credit for those parts of the project you completed.

But whatever you do, make sure that **ALL CODE THAT YOU TURN IN COMPILES AND LINKS** without errors!

## What to Turn In

You must turn in **eight to ten** source files. These eight are required:

<code>AdHunter.cpp</code>	Contains your news ad hunter implementation
<code>Crawler.cpp</code>	Contains your web crawler implementation
<code>Document.cpp</code>	Contains your document class implementation
<code>ExtractLinks.cpp</code>	Contains your link extractor implementation
<code>Matcher.cpp</code>	Contains your matcher class implementation
<code>MyHashMap.h</code>	Contains your hash map declaration
<code>Rule.cpp</code>	Contains your rule class implementation
<code>main.cpp</code>	contains your main code for the Ad Hunter project

These two are optional:

support.h	You may define support classes/constants in these
support.cpp	support files and use them in your other classes

Use support.h if there are constants, class declarations, and the like that you want to use in *more than one* of the files. (If you wanted to use them in only one file, then just put them in that file.) Use support.cpp only if you declare things in support.h that you want to implement in support.cpp.

You are to define your classes and all member function implementations directly within the specified .h and .cpp files. You may add any #includes or consts you like to these files. You may also add support functions for these classes if you like (e.g., operator<). You must also submit a report that details the following:

1. Describe the algorithm and data structures you used for your:
  - **AdHunterImpl class**: Describe the data structures and algorithms you used to efficiently construct, order and return your ad target match list to the caller.
  - **MatcherImpl class**: Describe the data structures and algorithms you used to efficiently find the matches in a document.
  - **MyHashMap class**: Describe the data structures and algorithms you used to implement your hash table mapper class.
2. Provide a thorough list of test cases that you used to test each of your classes and note what each test tested.
3. Give a list of any known bugs or issues with your program (e.g., it crashes under these circumstances, I wasn't able to finish class X and get it working properly, etc.).

## Grading

- 80% of your grade will be assigned based on the correctness of your solution.
- 5% of your grade will be assigned based on the documentation of your data structures and algorithms in your report.
- 5% of your grade will be based on the style of your code and the appropriateness of your algorithms and data structures.
- 10% of your grade will be assigned based on the thoroughness of your test cases.

Good luck!