CS32 Winter 2011
Project 4

News Junkie

Due: 9 PM, Thursday, March 10

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

Table of Contents

Introduction

For your fourth and final project, you've been hired by Carey&David Publishing, the world's 3452nd largest tabloid newspaper company. Due to the recent economic downturn, Carey&David (not to be confused with Harry&David, purveyors of expensive junk food) have lost thousands of subscribers and had to lay off their entire staff. Now they have just two employees left, Carey & David, who must write all of the stories for their fine newspaper. Of course, with just two employees, Carey & David can only research and write a few articles on their own each day, so they'd like to "borrow" news stories from other news services, like the New York Times, the Los Angeles Times, CNN, and the BBC for their paper.

Carey&David would like to hire you to build a C++ program to identify the most popular news stories hosted by other news services. They will use this information to determine what news is hot in the world, and "borrow" those same stories to include in their daily newspaper.

So what do they want you to do? Carey&David realized that most news services publish summaries of their daily news stories on special web pages called RSS feeds. An RSS feed is an internet web page that contains a summary of recent news stories in a simple text format called XML. Each RSS feed can be retrieved via a specific internet URL, just like a regular URL that you might type into your web browser. Each news service updates their RSS feed page on a daily basis, adding a summary of each new story they publish to the news feed.

Each RSS feed contains one or more news story summaries. Each summary contains a headline/title (we use the words "headline" and "title" interchangeably to mean the same thing) such as "UCLA beats USC for 10th year in a row", and a web URL that points to the full news story (e.g., "http://www.dailybruin.com/ucla-beats-sc-again.html") should the user want to read the full story.

Home users typically "subscribe" to one or more RSS feeds through their web browser or a special program called an RSS Reader. Once subscribed to a feed, the web browser or reader automatically checks the feed for updates several times per day to determine if any new summaries were added. If one or more new summaries were added, the RSS reader/browser pops up a window and shows the user the new headline(s), allowing them to click on any headline to follow the URL and read the complete story.

C&D would like to provide you with a list of RSS feed URLs for major news services and have you mine these feeds to identify important *news clusters* and *important keywords* that are making the news.

What is a news cluster? A news cluster is a group of related news stories (from one or more RSS feeds), where the headline of each news story in the cluster shares at least three words with the headline of at least one other news story that is also in the cluster.

The thinking is that if a headline shares many of the same words as another headline, the two are probably related. Moreover, if there are many stories that are related, then this is probably an important news event worth including in C&D's newspaper.

For example, consider the following five news summaries which each consist of a headline and a URL (which you can assume were taken from one or more RSS feeds):

> 1: CS32 students cheer for Computer Science: http://www.dailybruin.com/cs32.html
> 2: EE professor invents new circuit from CMOS: http://www.ee.com/new-cmos.html
> 3: Computer Science students love math: http://www.dailybruin.com/cslovemath.hml
> 4: Electrical Engineering students cheer for CS32: http://www.eenews.com/xx.html
> 5: Electrical Engineering professor discovers new CMOS circuit: http://www.cnn.com/cmos.html
> 6: David Smallberg voted sexiest geek third year in a row: http://www.geeknews.com/sexiest.html

Based on C&D's definition of a cluster, stories #1, #3, and #4 would be clustered together into a group, since all have at least three words in common with at least one of the other stories in the cluster:

> #1 and #3 share "Computer," "Science," and "students"
> #1 and #4 share "students," "cheer", and "CS32"

Note that a headline within a cluster does *not* need to share at least three common words with *each* other headlines in cluster. So long as every headline in the cluster shares at least three common words with at least one other headline in the cluster, that's sufficient for it to belong to the cluster. So, even though stories #3 and #4 only share one word in common ("students") with each other, they're still part of the same cluster, because both share at least three words in common with headline #1.

Similarly, stories #2 and #5 would be grouped together into their own cluster, since they share three words: "professor," "CMOS," and "circuit." However, both headlines #2 and #5 share fewer than three words in common with stories #1, #3, and #4, so they would be grouped in an entirely different cluster than the one made up from #1, #3, and #4.

Finally, story #6 would not be grouped in either of the above two clusters since it fails to share at least three words with any of the headlines in either of the two other clusters.

In addition to identifying newsworthy topics, C&D also want you to identify *important keywords* that are present in the headlines of news stories. An important keyword is a word which is found in a minimum percentage P of all processed headlines, where P is a value specified by the user. These important keywords will enable Carey or David to identify other newsworthy topics that might be missed by the news clustering approach. So, for example, let's assume that your program discovered 200 distinct stories in its RSS feeds, and the user specified a P value of 2%. In this case, let's further imagine that the word "Libya" occurred in exactly 5 of the headlines of those stories. In this case, "Libya" would be considered an important keyword since it was present in at least 2% of the 200 headlines.

In this project, you'll be building a set of classes and a main() function to do the following:

1. You need to take in a list of RSS feed URLs that C&D provides, e.g.:
   http://feeds.latimes.com/latimes/news?format=xml
   http://feeds.nytimes.com/nyt/rss/HomePage
   http://rss.news.yahoo.com/rss/topstories
   ...
2. You need to retrieve the web pages associated with each of these URLs, one at a time using a library we provide to get data off the Internet.
3. From each RSS web page, you need to extract the series of story summaries (each has a headline/title and a URL) embedded in each page. We'll explain the XML format of the RSS web pages below. For example, you might extract the following three stories from the feed:
   http://feeds.latimes.com/latimes/news?format=xml
   A. "Egypt overthrows government": http://www.latimes.com/egypt-govt-falls.html
   B. "Los Angeles gets a new mayor": "http://www.latimes.com/new-mayor.html"
   C. "LA has a new leader": "http://www.latimes.com/new-mayor.html"
   D. "Carey Nachenberg voted silliest prof": "http://www.latimes.com/sillyprof.html"
4. You need to discard all but one of a collection of duplicate stories. Two or more stories are duplicates of each other if they have identical URLs (even if they have different headlines). For example, stories B and C would be duplicates because although they have different headlines, they refer to the same web page. You would retain one of them (either one is OK) and discard the other.
5. You then need to group all of the stories into clusters of related headlines. We'll provide more detail on how to cluster stories in the sections below.
6. You then need to determine the most important keywords across all of the headlines.
7. Finally, you will output the biggest clusters and the most important keywords to the user so Carey & David can use this to write the C&D newspaper.

So, what does this all look like? Well, below is the output from our solution. When you run the News Junkie program, you must specify a file that contains the list of RSS feed URLs as well as a number (explained a few paragraphs below) representing a percentage that determines how many newsworthy clusters and keywords to print. Here's an example:

```
C:\CS32\PROJ4> proj4.exe rsslist.txt 1


Finding the top news stories and keywords...

There were 5 newsworthy topics:

Cluster # 1 has 5 stories about: Bomber kills 26 Shiite pilgrims in Iraq
 http://feeds.latimes.com/~r/latimes/news/~3/p9wCZyEAFto/la-fgw-iraq-bombing-20110213,0,1125423.story
 http://feeds.nytimes.com/click.phdo?i=6a65779b0ae0991159344c0091e79e6d
 http://feeds.reuters.com/~r/reuters/worldNews/~3/ORvAkHdtWOA/us-iraq-violence-idUSTRE71B1CT20110212
 http://www.bbc.co.uk/go/rss/int/news/-/news/world-middle-east-12440175
 http://www.nypost.com/p/news/international/iraq_suicide_bomber_kills_shiite_ubdEbpxipjje6YLA1QjEoJ?

Cluster # 2 has 4 stories about: Egypt seeks fresh start after Mubarak
 http://feeds.latimes.com/~r/latimes/news/~3/elVU9kZNxxY/la-fg-egypt-mubarak-20110212,0,5712952.story
 http://us.rd.yahoo.com/dailynews/rss/topstories/*http://news.yahoo.com/s/afp/20110212/wl_mideast_afp/egyptpolunrest
 http://www.ft.com/cms/s/0/280e21c0-35da-11e0-b67c-00144feabdc0.html?ftcamp=rss
 http://www.un.org/apps/news/story.asp?NewsID=37513&amp;Cr=Egypt&amp;Cr1=
Cluster # 3 has 3 stories about: Musharraf to defy Pakistan arrest warrant: spokesman
 http://feeds.reuters.com/~r/reuters/worldNews/~3/kQp_LxVHfWo/us-pakistan-bhutto-musharraf-idUSTRE71B0W720110212
 http://www.bbc.co.uk/go/rss/int/news/-/news/world-south-asia-12438702
 http://www.nypost.com/p/news/international/pakistan_court_issues_arrest_warrant_QMhUup7zCF7WrPyAN3yECJ?CMP

Cluster # 4 has 3 stories about: Nightclub horror as San Diego cab plows into crowd
 http://feeds.latimes.com/~r/latimes/news/~3/CdgU2Yd07Pc/la-me-san-diego-gaslight-cabbie-m,0,6870066.story
 http://us.rd.yahoo.com/dailynews/rss/topstories/*http://news.yahoo.com/s/ap/20110212/ap_on_re_us/ussandiego_taxi_crash
 http://www.nypost.com/p/news/national/nightclub_horror_as_san_diego_cab_08mHSl9i4Y7P3O5bQJjfhN?CMP

Cluster # 5 has 3 stories about: Suicide bomber kills 38, wounds dozens in Iraq
 http://feeds.latimes.com/~r/latimes/news/~3/p9wCZyEAFto/la-fgw-iraq-bombing-20110213,0,1125423.story
 http://feeds.reuters.com/~r/reuters/worldNews/~3/ORvAkHdtWOA/us-iraq-violence-idUSTRE71B1CT20110212
 http://www.bbc.co.uk/go/rss/int/news/-/news/world-middle-east-12440175


There were 7 newsworthy keywords:
 Egypt: 27 uses
 VIDEO: 11 uses
 Mubarak: 10 uses
 Reuters: 10 uses
 with: 10 uses
 after: 9 uses
 Obama: 3 uses
```

Notice that for each news cluster, the program prints out a representative headline (which must be taken from the first story in the cluster) and then the set of URLs for stories in the cluster. Also note that clusters are printed out from most significant to least significant (the more stories in a cluster, the higher up it is in the program's output). Any clusters that tie (i.e., that have the same number of stories) are ordered alphabetically based on their representative headline.

The number on the command line (1 in the example above) is interpreted as a percentage. Only clusters that contain at least that percentage of the total number of all distinct stories should be printed. For example, suppose that there at 10 RSS feeds that, in toto, contain references to 290 distinct news stories, and that the percentage specified on the command line was 1 (as in the example above). Since 1% of 290 is 2.9, the only clusters that are considered newsworthy enough to print are those that contain at least 2.9 stories (actually, of course, that would mean at least 3 stories, since a cluster cannot contain a fraction of a story). If a cluster contained only 2 related stories, this would not rise to the required level of newsworthiness and therefore wouldn't be printed out.

Next, notice the newsworthy keyword section of the output. Again, since the user in the example specified a threshold of 1 on the program command line, only keywords that were found in at least 1% of all headlines (2.9 of the 290 distinct stories) will be printed out. In the above example, there were seven such keywords that met this requirement. They were outputted in a sorted fashion, from the most prevalent keyword to least prevalent keyword, with keywords that tie in prevalence sorted alphabetically (e.g., "Mubarak" comes before "Reuters" and "Reuters" comes before "with"). A keyword that appeared only 2 times would not meet the 2.9 threshold, so would not be printed.

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 class 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 class, 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 HTTP class 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[1]. Our HTTP class'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 webpage for you! So easy!
   if (HTTP().get("http://feeds.latimes.com/latimes/news?format=xml",
                                                                page))
      cout << page;  // prints <?xml version="1.0" encoding="UTF-8"?>…
   ...
}
```

Note that you don't need to declare an HTTP variable. The call above looks as if it default constructs a temporary HTTP object and calls the get member function on it.

Ok, so what is it I have to do?

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

The RSSProcessor Class

The fist class you have to build is called *RSSProcessor* and is responsible for retrieving all of the current story summaries from an RSS feed URL such as *http://feeds.latimes.com/latimes/news?format=xml*. The class then allows the user to enumerate the stories on the page, one at a time, and obtain their headline and the URL of each story.

An RSS feed page contains content encoded in the XML format, as opposed to traditional web pages which are encoded in the HTML format.. An XML page encodes data in a well-defined regular format, making it easy to parse. Here's an example of an RSS feed in XML with line numbers added for clarity. We retrieved this page from the web page

[1] By agreeing to use our HTTP code for project #4, this license entitles Carey&David to a 20% cut of all profits.

"http://feeds.latimes.com/latimes/news?format=xml" and then thinned it out a bit to make it more readable:

```
01: <?xml version="1.0" encoding="UTF-8"?>
02: <?xml-stylesheet type="text/xsl" media="screen"
03:   href="/~d/styles/rss2full.xsl"?><?xml-stylesheet type="text/css" media="screen"
04:   href="http://feeds.latimes.com/~d/styles/itemcontent.css"?>
05:
06: <rss xmlns:media="http://search.yahoo.com/mrss/"
07:   xmlns:feedburner="http://rssnamespace.org/feedburner/ext/1.0" version="2.0">
08:  <channel>
09:   <title>Los Angeles Times - Top News</title>
10:
11:   <item>
12:     <title>Egypt army as steward of transition in question</title>
13:     <link>http://feeds.latimes.com/~r/latimes/news/~3/TeiWr8bojbs/la-fg-egypt-army-
14:           20110213,0,6341757.story</link>
15:   </item>
16:
17:   <item>
18:     <title>A reborn Egypt gets back to business</title>
19:     <link>http://feeds.latimes.com/~r/latimes/news/~3/h-aLZlLeNKU/la-fg-egypt-day-
20:       after-20110213,0,6026871.story</link>
21:   </item>
22:
23:   <item>
24:     <title>'Tea party' litmus tests pull Republicans to right</title>
25:     <link>http://feeds.latimes.com/~r/latimes/news/~3/8hj2jc2GDXA/la-na-cpac-
26:           20110213,0,116773.story</link>
27:   </item>
28:  </channel>
29: </rss>
```

As you can see, the RSS feed has header information (on lines 1 through 10), followed by a series of news stories enclosed by <item>story</item> tags. The first story is on lines 11 through 15, the second is on 17 through 21, etc. Each item provides the headline/title of a single news story as well as a URL where the full text of that news story can be found on the Internet. Note, the example above is actually a simplified RSS feed - real feeds can have a number of other fields (not shown above) such as "<description>this describes the story in a paragraph or so of detail</description>", or "<pubDate>Sat, 26 Feb 2011 10:30:00 -0800</pubDate>" which describes the date of publication of each item, etc. However, the only items you need to worry about are <title>headline</title> and <link>URL</link> found inside each <item> … </item> set of tags.

The job of the RSSProcessor class is to retrieve such an RSS page from the Internet and then parse its contents for every item (story) in the page, extracting a title (headline) and a URL for each item. The class will then allow the user to retrieve each individual story's headline and URL, one at a time, using other member functions.

What is the *required* interface of the RSSProcessor class? Let's see:

```
class RSSProcessor
{
public:
   RSSProcessor(std::string rssURL);
   bool getData();
   bool getFirstItem(std::string& link, std::string& title);
   bool getNextItem(std::string& link, std::string& title);
};
```

As you can see, the RSSProcessor class has a constructor that accepts a single string argument: the URL for an RSS feed.

```
std::string rssURL = "http://feeds.latimes.com/latimes/news?format=xml";

RSSProcessor rssp(rssURL);
```

After constructing an RSSProcessor object, the user would then call its getData() method, which instructs the object to connect to the specified website (latimes.com), retrieve the specified web page, and then extract all of the news story summaries from the page. Here’s an example of how a user might call this class’s getData() method:

```
if (rssp.getData())
   cout << "Woohoo, I got it!";
else
   cout << "Something bad happened and I couldn't get the feed";
```

Your getData() method is responsible for parsing through the XML-encoded data, searching for the <item>s and extracting the title/headline and URL for all items on the page. Given how well-defined XML is, this should be pretty easy to do using the standard C++ string class and its many find() functions.

After successfully calling getData(), the user may then call the getFirstItem function to retrieve the title (headline) and link (URL) of the first item in the data. If the data has no items, this function returns false, otherwise it returns true. The title and link are placed in the function's parameters. After getFirstItem has been called, the user may call the getNextItem function one or more times to retrieve the title and the link of each successive story from the data; they are placed in the function's parameters. The getNextItem function returns true each time it is called until there are no more items to get, in which case it returns false. Here’s how you might use the RSSProcessor class:

```
string headline;
string url;
bool gotItem = rssp.getFirstItem(url, headline);
while (gotItem)
{
   cout << "Headline: " << headline << endl
        << "URL of story: " << url << endl << endl;
   gotItem = rssp.getNextItem(url, headline);
}
```

For our earlier RSS feed, this little program snippet above should print:

```
Headline: Egypt army as steward of transition in question
URL of story: http://feeds.latimes.com/~r/latimes/news/~3/TeiWr8bojbs/la-fg-egypt-army-
20110213,0,6341757.story

Headline: A reborn Egypt gets back to business
```

```
URL of story: http://feeds.latimes.com/~r/latimes/news/~3/h-aLZlLeNKU/la-fg-egypt-day-
after-20110213,0,6026871.story

Headline: 'Tea party' litmus tests pull Republicans to right
URL of story: http://feeds.latimes.com/~r/latimes/news/~3/8hj2jc2GDXA/la-na-
cpac20110213,0,116773.story
```

To ensure that you do not change the interface to the RSSProcessor 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 RSSProcessor just one private data member, a pointer to a RSSProcessorImpl object (which you can define however you want in RSSProcessor.cpp). The member functions of RSSProcessor simply delegate their work to functions in RSSProcessorImpl.[2] You still have to do the hard work of implementing those functions.

Once you’ve finished your class, test it with several sample RSS feed page URLs. (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 RSSProcessorImpl class to work, don’t worry – we’ll also provide our own working version when we test other parts of 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";
```

OK, so now we know how to download an RSS feed. And you know now how to parse the XML feed and extract its stories. So you know everything you need to implement the RSSProcessor class. Now what?

The NewsCluster Class

The goal of this project is to identify related news stories and cluster (group) them together. Your NewsCluster class is responsible for growing a cluster of stories. Since you may have many different clusters of stories, your other classes will be using multiple NewsCluster objects.

For this project, we shall consider two stories related if their headlines have in common at least three words of at least four characters in length, in any order. So for example, consider the following three headlines:

1: I am a cat meow hiss scratch
2: I am a cat and I like fish
3: You often scratch and hiss and sometimes meow

Since for the purpose of grouping/clustering we ignore all words with fewer than four letters, we can think of our headlines like this:

[2] This is an example of what is called the pimpl idiom (from "**p**ointer-to-**impl**ementation").

1: meow, hiss, scratch
2: like, fish
3: often, scratch, hiss, sometimes, meow

In this example, stories 1 and 3 would be related, since they share the three words: meow, hiss, and scratch, although not necessarily in the same order. Even though stories 1 and 2 both have “I,” “am,” “a,” and “cat” in common, these words are ignored due to their short length, and therefore stories 1 and 2 would not be considered related.

So now we know how to determine if two or more stories are related for the purposes of this project.

Ok, here’s what the second class that you have to write looks like:

```
class NewsCluster
{
public:
   NewsCluster();
   bool submitKernelStory(std::string headline, std::string url);
   bool submitStory(std::string headline, std::string url);
   std::string getIdentifier() const;
   bool getFirstNewsItem(std::string& headline, std::string& url);
   bool getNextNewsItem(std::string& headline, std::string& url);
   int size() const;
};
```

When you use this NewsCluster class, you will first specify a “kernel” story that will form the basis of the current cluster:

```
NewsCluster nci;
nci.submitKernelStory("green cheese found on the moon",
                        "http://latimes.com/green-cheese-found.html");
```

Every cluster must start out with some first story in it and then grow by adding other stories (which are related in some way to the first story by their headlines). The “kernel” story is the first story to be added to a cluster and it defines the character of the cluster.

After submitting a kernel story, the user of this class can then call the submitStory() method zero or more times to submit other stories to potentially add them to the cluster:

```
nci.submitStory("green eggs and ham found to annoy astronauts",
                "http://latimes.com/green-eggs-and-ham.html");

nci.submitStory("green astronauts eat cheese on the moon",
                "http://latimes.com/astronauts.html");

nci.submitStory("green food coloring annoys moon astronauts",
                "http://latimes.com/green-coloring.html");
```

Each time the user calls the submitStory() method, it must chop the passed-in headline into separate words (using our provided *WordExtractor* class if you like), throw out all words shorter than 4 characters long, and then check to see if the newly submitted story's words match at least three of the (4-character-or-longer) words that make up headlines that are already in the cluster (including the kernel story headline that started out the cluster). If so, then the submitStory() method must add the new story to the cluster and return true. If not, then the submitStory() method simply ignores the submitted story, leaves the cluster unchanged and returns false. If the cluster already contains the submitted story headline, then the cluster should remain unchanged and the function should return false.

Let's review the example above. The cluster would start out with a kernel headline of *"green cheese found on the moon."* The second submitted headline *"green eggs and ham found to annoy astronauts"* matches only two words ("green" and "found") of the cluster's only headline, so therefore our new story does not belong in the cluster and would not be added to the kernel by submitStory(), and the function would return false.

The third submitted headline "green astronauts eat cheese on the moon" matches three words with the kernel (and currently the only) headline in the cluster, so this new story would be added to the cluster and the function would return true. Now our cluster would contain:

```
green cheese found on the moon
green astronauts eat cheese on the moon
```

The fourth submitted headline "green food coloring annoys moon astronauts" has only two words in common with the kernel story headline, but it shares three words ("green", "moon", and "astronaut") in common with the second story's headline in the cluster. Therefore, the submitStory() method would add this fourth story headline and URL to the cluster and then return true. The new cluster would contain:

```
green cheese found on the moon
green astronauts eat cheese on the moon
green food coloring annoys moon astronauts
```

(as well as the associated URL for each headline, not shown above)

And so on. So you can see that the cluster may grow each time we submit a new story to it, so long as that new story matches the headline of at least one existing story within the cluster. Note that repeated submission of the same headline to a cluster should be ignored once the cluster already contains a story (i.e., your cluster should never have duplicate stories).

Now, if you're paying attention, you may notice something interesting. When we first tried to submit the second story ("green eggs and ham found to annoy astronauts") for inclusion in our cluster, it didn't belong, since it only shared two words in common with our original kernel headline. However, some time after we originally tried adding this story, things changed in our cluster. Our new cluster now has three different stories. What would happen if we tried to add our second story now? Well, it still only shares two words in common with our kernel headline. However, it shares three words in common with our third cluster entry: "green", "annoys" and "astronauts" can be found in both headlines. And so if we were to call the submitStory() method again (a fourth time) with the second story, this time it would end up adding that story to the cluster.

The moral of the story is that the order that you submit items to a cluster matters. The implications of this will make more sense later.

What must the other functions in NewsCluster do?

```
std::string getIdentifier() const;
```

The getIdentifier() method should return an alphabetized, concatenated string of all the headlines that are in the cluster, placing '+' signs in between each item. For the cluster we created above:

```
      green cheese found on the moon
      green astronauts eat cheese on the moon
      green food coloring annoys moon astronauts
```

This method would return:

```
green astronauts eat cheese on the moon+green cheese found on the moon+green food coloring annoys moon astronauts
```

```
bool getFirstNewsItem(std::string &headline, std::string &url);
bool getNextNewsItem(std::string &headline, std::string &url);
```

The getFirstNewsItem() method should select the kernel story of the cluster, place its headline into the headline parameter, and place its URL into the url parameter. If there is at least one headline in the cluster, then this method should return true. If the cluster is empty, then obviously there is no headline/URL to return, so this method should return false and leave its parameters unchanged.

The getNextNewsItem() method should retrieve another story from the cluster, fill in the headline and url parameters for this story, and return true. Repeated calls to this method should return successive headline/URL pairs that are part of this cluster. If no more headlines are available, then this method should return false and leave its parameters unchanged.

```
int size() const;
```

This method returns the number of distinct news stories in the cluster.

As with the other classes you must write, the real work will be implementing the auxiliary class NewsClusterImpl in NewsCluster.cpp. Once you've finished your class, test it with numerous inputs to make sure it works under many different circumstances. If you can't get your NewsClusterImpl class to work, don't worry – we'll also provide our own working version when we test other parts of 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.

Oh, and by the way, we know how much you hate parsing strings to extract words. Since you were already so brave to write your own RSSProcessor class that had to parse RSS feed pages, we'll provide you with a simple class to extract consecutive words from a string (such as a headline). Our provided class is called WordExtractor, and here's its interface:

```
class WordExtractor
{
public:
    WordExtractor(const std::string& text);
    bool getNextWord(std::string& word);
};
```

You pass the constructor a string, such as a headline. You can then use the getNextWord function to extract the series of alphabetical words that make up the page:

```
WordExtractor we("Carey and David deemed AWESOME!");
std::string word;
while (we.getNextWord(word))
    cout << "Next word: " << word << endl;
```

This would print:

```
Next word: Carey
Next word: and
Next word: David
Next word: deemed
Next word: AWESOME
```

Now doesn't that make life easier?

The StringMapper Class

You must build a template class named StringMapper that lets you map C++ strings to any other object or scalar type of your choosing.

Any time you would otherwise need an STL map or unsorted_map to implement your code, you MUST use your StringMapper class to implement the code instead.

You must NOT use the STL map class or unsorted_map class, anywhere in this project.

You may use the STL set, unsorted_set, vector, list, stack, or queue in any of your classes (other than StringMapper) if you like.

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

Here's how you might use this StringMapper class:

```
#include "Mapper.h"

int main()
{
    StringMapper<int> nameToAge;

    nameToAge.insert("Carey", 39); // first parameter's always a string
    nameToAge.insert("David", 41);
    nameToAge.insert("King Tut", 3352);   // King Tut is a mummy

    bool found;
    int result;

    if (nameToAge.find("Carey", result))
        cout << "Carey is: " << result << " years old." << endl;
    else
        cout << "Carey was not found in our map!" << endl;

      // enumerate all items in our map
    string name;
    int age;
    bool gotPair = getFirstPair(name, age);
    while (gotPair)
    {
        cout << name << " is " << age << " years old." << endl;
        gotPair = getNextPair(name, age);
    }
    cout << "Finished printing all items in our map." << endl;
}
```

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:

```
StringMapper<Shape*> m;

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

m.insert("circle", c);
m.insert("square", s);
…
```

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

```
template<typename T>
```

```
class StringMapper
{
public:
    StringMapper( ♠ );    // ♠ must work with zero parameters
                          //   but may also accept one parameter
    ~StringMapper();
    void insert(std::string from, const T& to);
    bool find(std::string from, T& to) const;
    bool getFirstPair(std::string& from, T& to);
    bool getNextPair(std::string& from, T& to);
    int size() const;
};
```

You are to implement your StringMapper using either a hand-written *open hash table* or with a hand-written *binary search tree*. You may choose which data structure to use.

If you decide to implement a hash table, your 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 the default hash table size should be 10000.

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

1. A call to either the getFirstPair() or to getNextPair() method must return a {string, T} pair 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 for those that want to implement this with a hash table: 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 StringMapper, your destructor must not perform more than N iterations to destroy a StringMapper object that contains exactly N entries. For example, if your hash table has 10,000 buckets but only 5 inserted items, your destructor must only loop 5 times to do its job. (This is basically solved by the same solution to problem #1 above)
3. If the user of your StringMapper class inserts an item, then the behavior of the next call to getNextPair() is undefined without an intervening call to getFirstPair, which means your program may perform any behavior it likes, including crashing.
4. If the user calls the getNextPair() method without first calling the getFirstPair() method, then the behavior of the getNextPair() method is also undefined.
5. ***The final, submitted version of your StringMapper class must not use any STL classes.*** However, you may find it useful to implement your StringMapper class using the STL map or unsorted_map class initially so you can get up and running quickly.

The NewsAggregator Class

Now things are getting interesting. Your NewsAggregator class is responsible for bringing the whole news aggregation system together. This class takes as input a set of URLs to RSS feed pages. It must retrieve all of the *distinct* stories from each of the RSS feeds (using the RSSProcessor class), and then cluster all of these stories (using the NewsCluster class).

Your class must also identify all prevalent keywords used in all of the headlines of all of the stories.

Once it has clustered all of the stories and identified the most prevalent keywords, then it should return this data to the user.

Your NewsAggregator class must have the following public interface:

```
class NewsAggregator
{
public:
    NewsAggregator();
    void addSourceRSSFeed(std::string feed);
    int getTopStoriesAndKeywords
      (
        double thresholdPercentage,
        std::vector<Cluster>& topStories,
        std::vector<Keyword>& topKeywords
      );
};
```

The user can call the addSourceRSSFeed() method one or more times to add new RSS feed URLs for retrieval.

When the user calls the getTopStoriesAndKeywords() method, it should do the following:

1. Connect to all of the RSS feed websites and retrieve all *distinct* stories (each story is a {headline,URL} pair) from all of the specified RSS feeds using the RSSProcessor class. If any RSS feeds are unavailable (their URLs are invalid, for example), then your method may skip these feeds. Stories are identified by their URL *not* their headline, and two stories are considered distinct if they have different URLs. Two stories are considered to be duplicates if they have the same URL, even if they have different headlines. You must throw away any duplicate stories so you just have one copy of each story across all of the RSS feeds; which one you retain is up to you.
2. Cluster all of the retrieved stories using the approach described below.
3. Eliminate duplicate clusters using the approach described below.
4. Identify the top distinct news clusters. We'll explain how do this below.

5. Clear out the topStories vector parameter and fill it with information on the top news clusters.
6. Identify all of the top keywords across all of the headlines using the approach described below.
7. Clear out the topKeywords vector parameter and fill it with information on the top keywords.
8. Return the results to the caller of your function.

How to Cluster

So, how should you go about clustering news stories? Here's an algorithm you can use – it's not the most efficient, but it yields reasonably good results:

1. For each of your K distinct stories S_i, create a new NewsCluster object C_i based on story S_i (each story is the kernel of a single cluster).
2. Repeatedly do the following:
    a. For each of your K distinct news stories, submit that story to all clusters for potential inclusion using the cluster's submitStory() method. Each time you submit a news story to a cluster, check to see if the cluster was updated to include the new story. You must keep track of whether or not at least one cluster was expanded to include a new story during the current pass.
    b. After submitting all K stories to each of the clusters, if at least one cluster was expanded to include a new story, then repeat step #2 (a and b) again, and so on. You must continue to submit stories to each of the clusters until there are no changes to any of the clusters.

After you have finished step 2, you will have completely clustered all of your news stories. However, you may have duplicate clusters. For example, imagine that you were clustering the following stories:

Story #1: UCLA wins the championship
Story #2: USC found to have subpar students
Story #3: UCLA wins one championship after another
Story #4: Caltech wins another championship

Initially, cluster 1 would contain story #1, cluster 2 would contain story #2, cluster 3 would contain story #3, and cluster #4 would contain story #4.

After running steps 2a and 2b just once, cluster 1 would contain stories #1 and #3, cluster 2 would contain story #2, cluster 3 would contain stories #1 and #3, and cluster 4 would contain story #4 and #3.

After running steps 2a and 2b a second time, cluster 1 would contain stories #1, #3 and #4. Cluster 2 would still contain just story #2. Cluster 3 would contain stories #1, #3 and #4. Finally cluster 4 would end up with stories #1, #3 and #4.

After running steps 2a and 2b a third time, none of your clusters change (they don't gain any new stories), so you're done building your clusters. Now you'll notice that clusters #1, #3 and #4 all contain the same stories – they're duplicates of each other. This makes sense since if two or more different clusters share at least one similar item at any point during the algorithm, then our algorithm will ensure that they all eventually end up with the same items.

Now, when we return our top stories to the user, we don't want to return cluster 1, cluster 3 and cluster 4, since they all represent the same cluster of stories, so we need to eliminate the duplicates.

How to Identify Duplicate Clusters

To identify duplicate clusters and eliminate them, you can call each cluster's getIdentifier() method. If two or more clusters have the same identity string (as returned by getIdentifier()), then you know they're duplicate clusters and you can throw all but one of them away. You may choose which of the clusters to throw away, so long as you keep at least one of each duplicate cluster, and all non-duplicated clusters.

How to Identify the Top Words

Now, to identify the top words across all of the stories, you have to parse each headline from each distinct news story and extract all of the words that are at least 4 letters long from each headline (you may use our WordExtractor class if you like). You must then maintain a count for each word of how many news headlines contained the word. For example, if you had the following 4 headlines:

Story #1: UCLA wins the championship
Story #2: Harvard found to have subpar students. Poor Harvard.
Story #3: UCLA wins one championship after another
Story #4: Caltech wins another championship

You should end up with the following counts (for words of at least length 4):

UCLA: 2
wins: 3
championship: 3
Harvard: 1 // Even though Harvard is used twice in story #2, it only occurs in 1 headline!
found: 1
have: 1
subpar: 1

students: 1
Poor: 1
after: 1
another: 2
Caltech: 1

What to Pass Back

Let N be the threshold percentage times the number of distinct stories in all feeds. Your method must fill in the topStories vector parameter with Cluster objects representing all clusters that have at least N stories in them, ordered from the largest cluster to the smallest cluster. (We provide you with the Cluster class implementation, so you don't have to write that yourself.) For example, if there were 400 total stories found, and thresholdPercentage is 2.5, then a cluster would meet the threshold if it had at least 2.5% of 400 stories or 10 stories in it. Any cluster with fewer than 10 stories would not be returned to the user. If multiple clusters that meet this minimum requirement have the *same* number of stories, then you must further order these tied clusters alphabetically within the vector based on the title of each cluster.

Each Cluster object in your topStories vector must contain:

1. A title, i.e., the headline of the kernel story in the cluster (the first story that the cluster started with).
2. The complete collection of URLs associated with all stories that were in the cluster.

Note that each Cluster object holds only one headline but potentially many URLs. This is OK since all of the stories are related anyway. The headline is just meant to give the user a general idea about the nature of the stories in the cluster. The URLs allow the user to find all stories related to this headline.

So, given the following clusters with headlines H_i and URLs U_i, and assuming N was equal to 3:

Cluster #1: { {H4, U4}, {H5, U5} }
Cluster #2: { {H6, U6}, {H7, U7}, {H8, U8} }
Cluster #3: { {H1, U1}, {H2, U2}, {H3, U3} }
Cluster #4: { {H9, U9}, {H10, U10}, {H11, U11}, {H12, U12} }

And further assuming that headline H_i is earlier alphabetically than H_j, for $i < j$ (e.g., H1 comes earlier alphabetically than H2, and H2 comes earlier alphabetically than H3). Your output topStories vector would be:

```
topStories[0]: H9: U9, U10, U11, U12     // since this cluster has the most stories
topStories[1]: H1: U1, U2, U3            // since H1 comes earlier than H6 alphabetically
```

topStories[2]: H6: U6, U7, U8 // and otherwise these two clusters are tied

Your method must also fill in the topKeywords vector parameter with Keyword objects (we provide the Keyword class implementation for you) representing all keywords that were seen in the headlines of at least N stories, ordered from the most popular keyword to the least popular. So for example, if we had 400 stories and thresholdPercenatge was 2.5, then a keyword would only be returned if it was found in at least 2.5% of 400 (or 10) distinct stories. If multiple keywords meeting this requirement have the same count, then you must further order these tied keywords alphabetically within the vector.

So, for the keywords we extracted in our last example:

UCLA: 2
wins: 3
championship: 3
Harvard: 1
found: 1
have: 1
subpar: 1
students: 1
Poor: 1
after: 1
another: 2
Caltech: 1

and a thresholdPercentage of 50 (50%), your topKeywords vector would contain:

topKeywords[0]: {championship, 3}
topKeywords[1]: {wins, 3}
topKeywords[2]: {UCLA, 2}
topKeywords[2]: {another, 2} // lower case 'a' comes after upper case 'U' in ASCII

As with the other classes you must write, the real work will be implementing the auxiliary class NewsAggregatorImpl in NewsAggregator.cpp. Once you've finished your class, test it! If you can't get your NewsAggregatorImpl 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 NewsAggregator class will likely need to make use of the functionality provided by the other classes (e.g., RSSProcessor). However, it must never instantiate or use your other Impl classes directly. Instead, it must use our cover versions of these classes (e.g., RSSProcessor). This will allow us to test your NewsAggregator class even if you can't get one or more of your other classes, like RSSProcessorImpl, to work, since we can easily provide our own correct RSSProcessorImpl to replace your buggy/incomplete version.

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 should accept a list of parameters from the command line, e.g.:

```
C:\CS32> proj4.exe rsslist.txt 3.5
```

- The first parameter is the name of a file that contains all of your RSS feed URLs. The file will have one URL per line and could look something like this:

```
http://feeds.latimes.com/latimes/news?format=xml
http://feeds.nytimes.com/nyt/rss/HomePage
http://rss.news.yahoo.com/rss/topstories
http://news.google.com/news?pz=1&cf=all&ned=us&hl=en&output=rss
http://www.npr.org/rss/rss.php?id=1001
http://feeds.reuters.com/reuters/worldNews
http://rss.cnn.com/rss/cnn_topstories.rss
http://www.cnbc.com/id/19789731/device/rss/rss.html
http://feeds.bbci.co.uk/news/rss.xml
http://content.usatoday.com/marketing/rss/rsstrans.aspx?feedId=news1
http://feeds.cbsnews.com/CBSNewsMain
http://www.un.org/apps/news/rss/rss_top.asp
http://www.ft.com/rss/home/us
http://www.nypost.com/rss/all_section.xml
```

- The second parameter is the threshold percentage value (e.g., 3.5 for 3.5%) that is used to determine the minimum size of a cluster to be included in the output, and the minimum number of hits for a keyword to be included in the output.

Here's how it might actually be called:

If you don't know what command line arguments are, read the next section for more information.

2. If two parameters aren't present, then your main function should print the following usage message to the screen and exit immediately:

```
usage: proj4.exe RSSlist thresholdPercentage
```

3. Otherwise, your main function (or a supporting function called by main) should load each URL in the input file named as the first argument into a string using C++ file operations and submit each of these lines to a NewsAggregator object using its addSourceRSSFeed() method. (The Input From Files link on the class web page shows you how to load the contents of a text file into a C++ string a line at a time). If there is an error opening or reading the contents of the input file, your main function should print the following to the screen and exit immediately:

```
error: unable to open RSS list file: theNameOfTheRSSFile
```

Where *theNameOfTheRSSFile* should be the name of the input file specified on the command line, e.g., "rsslist.txt"

4. Assuming you were able to load the list of RSS URLs, your main function (or a supporting function called by main) should print out

   ```
   Finding the top news stories and keywords...
   ```

   followed by an empty line. Your main function (or a supporting function called by main) should then use the NewsAggregator object to retrieve and cluster all of the stories across all of the RSS feeds. When the getTopStoriesAndKeywords function returns, your program should print out either:

   ```
   No newsworthy topics were found.
   ```

   if no news clusters had at least the threshold percentage of the total stories in them, or

   ```
   There were P newsworthy topics:
   ```

   followed by an empty line, where *P* is replaced by the number of clusters meeting the threshold.

5. Your main function (or a supporting function called by main) should then print out the list of all clusters (if any matched), in the order returned by your getTopStoriesAndKeywords function:

   ```
   Cluster # 1 has 3 stories about: Iran warships Suez trip 'back on'
    http://rss.cnn.com/~r/rss/cnn_topstories/~3/3SigeSxmAyE/index.html
    http://us.rd.yahoo.com/dailynews/rss/topstories/*http://news.yahoo.com/s/ap/20110217/ap_on_re_mi_ea/ml_egypt_iranian_warships
    http://www.bbc.co.uk/go/rss/int/news/-/news/world-middle-east-12493614

   Cluster # 2 has 3 stories about: Obama talks jobs with Jobs, other tech leaders
    http://feeds.cbsnews.com/~r/CBSNewsMain/~3/PVDM6v_eGYE/main20033160.shtml
    http://us.rd.yahoo.com/dailynews/rss/topstories/*http://news.yahoo.com/s/ap/20110218/ap_on_hi_te/us_obama
    http://us.rd.yahoo.com/dailynews/rss/topstories/*http://news.yahoo.com/s/nm/20110218/bs_nm/us_obama_meeting_jobs

   Cluster # 3 has 3 stories about: Wisconsin in near-chaos over anti-union bill - Los Angeles Times
    http://feeds.latimes.com/~r/latimes/news/~3/Mwehxjit8pg/la-na-wisconsin-unions-20110218,0,7955126.story
    http://news.google.com/news/url?sa=t&amp;fd=R&amp;usg=AFQjCNHkahKf4SX-
    http://us.rd.yahoo.com/dailynews/rss/topstories/*http://news.yahoo.com/s/ap/20110218/ap_on_re_us/us_wisconsin_budget_unions

   Cluster # 4 has 2 stories about: "5 Browns" Dad Guilty of Sex Abuse of Daughters
    http://feeds.cbsnews.com/~r/CBSNewsMain/~3/5FNIskFVF9I/8301-504083_162-20033000-504083.html
    http://www.npr.org/blogs/thetwo-way/2011/02/17/133852233/father-of-musical-act-the-5-browns-pleads-guilty-to-abuse

   Cluster # 5 has 2 stories about: Middle East Protests Force US on Defensive
    http://feeds.nytimes.com/click.phdo?i=633021c2fc7c4d347f61ae2e17dce90e
    http://www.cnbc.com//id/41638910

   Cluster # 6 has 2 stories about: Regional jitters as Bahrain faces flashpoint funerals
    http://feeds.reuters.com/~r/reuters/worldNews/~3/UhGGtLqT00M/us-protests-idUSTRE71F41K20110218
    http://us.rd.yahoo.com/dailynews/rss/topstories/*http://news.yahoo.com/s/nm/20110218/ts_nm/us_protests
   ```

   The general format here is:

   Cluster #1 has <fill-in-the-number> stories about: <kernel headline>
   <a single space>URLa
   < a single space>URLb
   <empty line>
   Cluster #2 <fill-in-the-number> has <fill-in-the-number> stories about: <kernel headline>

< a single space>URLg
< a single space>URLh
< a single space>URLi
<empty line>
…

6. Your program should then print out two empty lines, and then either:

   ```
   No important keywords were found.
   ```

   it no newsworthy keywords were present in at least the threshold percentage of the total stories, or…

   ```
   There were K newsworthy keywords:
   ```

   where *K* is replaced by the number of clusters meeting the threshold.

7. Finally, if there were one or more keywords meeting the threshold, your program should print the keywords out:

   ```
   There were 173 newsworthy keywords:
    Libya: 16 uses
    over: 13 uses
    Egypt: 9 uses
    VIDEO: 9 uses
    Reuters: 8 uses
    with: 8 uses
   ```

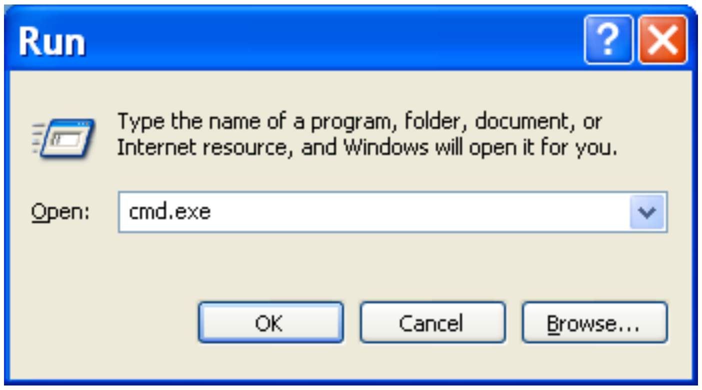
   with each line beginning with a single space, then the keyword, then a colon, then the number of different headlines the keyword was found in.

Your main function may then terminate.

Your main function will likely need to make use of the functionality provided by your classes (e.g., NewsAggregator). However, it must never instantiate or use your Impl classes directly. Instead, it must use our cover versions of these classes (e.g., NewsAggregator). This will allow us to test your main function even if you can't get one or more of your classes like NewsAggregatorImpl, to work, since we can easily provide our own NewsAggregatorImpl to replace your buggy/incomplete version.

How do Command Line Parameters/Arguments work?

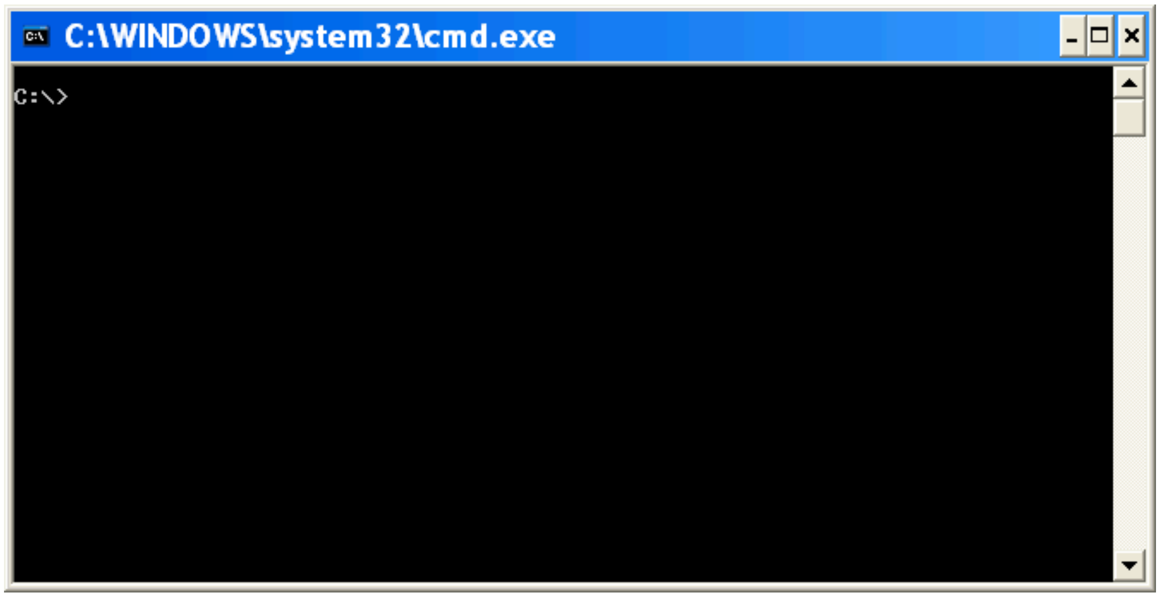
When you write a program, there's often a need for the user of the program to specify some parameters to the program to customize certain aspects of its operation. One way to specify parameters to a program is on the command line, via a UNIX or DOS command shell. You can, for example, launch a Windows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



Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

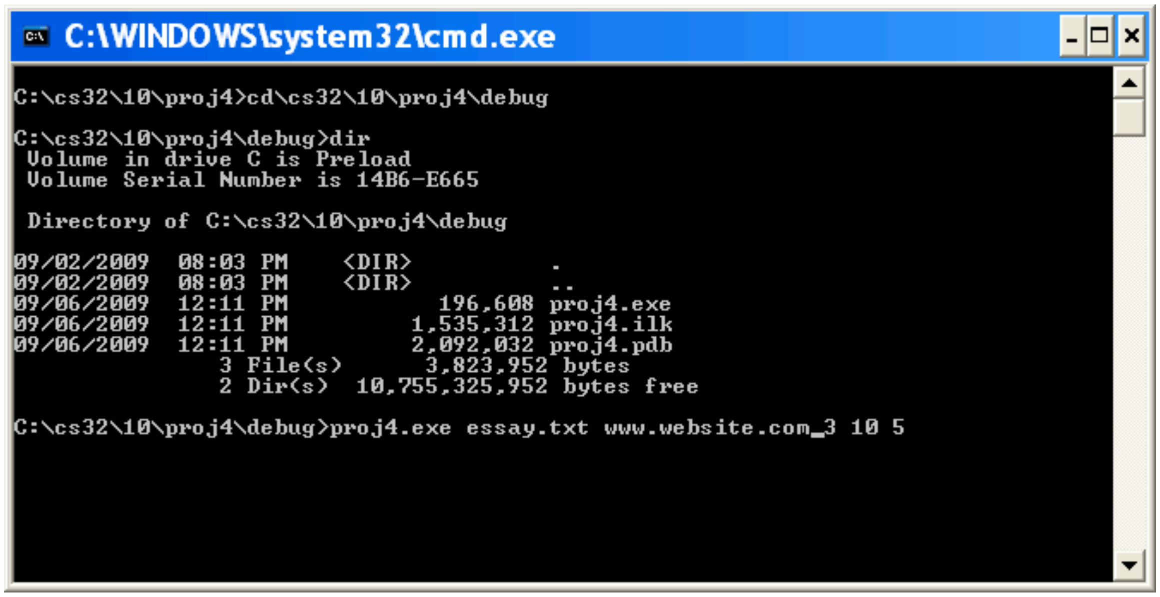
```
cmd.exe
```

This will bring up a box that looks like this.



```
C:\>
```

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:\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.ilk
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 “rsslist.txt”, 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 << " total args to this program\n";
    cout << "The program’s filename is " << argv[0];
    for (int i = 1; i < argc; i++)
      cout << "Arg  " << i << " has a value of: " << argv[i] << endl;
    ...
}
```

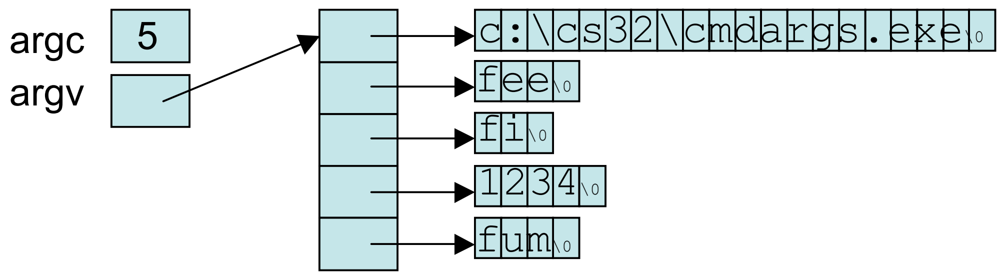
The argc variable is an integer that contains a count of the number of arguments to your program. The argv variable is a pointer to an array of pointers to strings (see diagram below). Each element of the argv array, e.g. argv[0], argv[1], etc., points to a different argument string.

The zero’th argument to every program is *always* the filename of the program itself, e.g. c:\cs32\10\proj4\debug\proj4.exe, in our above example – it is automatically added by C++ into the argument list before all explicitly provided arguments. The first through Nth arguments are the explicit arguments that the user typed after the filename, such as “rsslist.txt” or “3.5”. All arguments are passed in the form of C strings. Each argument must be separated by a space on the command line, and C++ uses these spaces to identify each separate parameter.

If we were to run the above source code with the following arguments:

```
C:\cs32> cmdargs.exe fee fi 1234 fum
```

argc and argv would look like this:



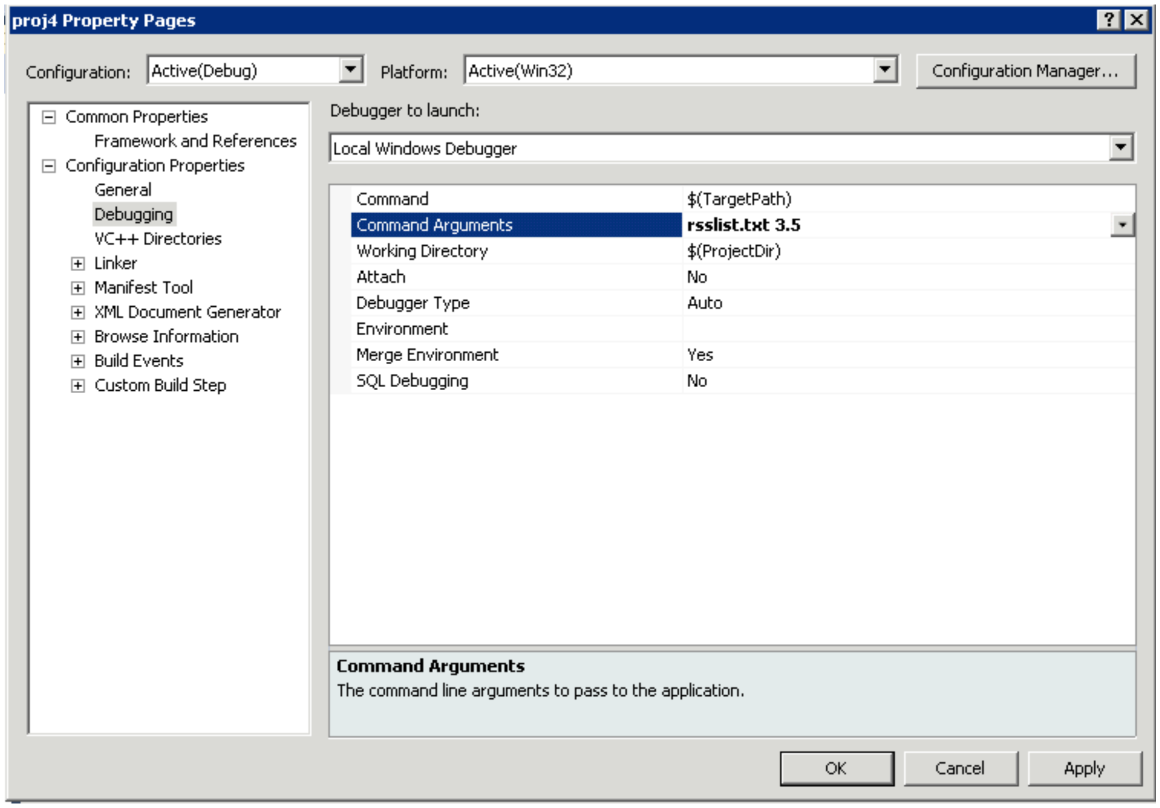
and our simple program would print:

```
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 fi
Arg 3 has a value of 1234
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 a numeric argument into a double before using it:

```
#include <cstdlib>      // needed for the strtod function
...
doouble val = strtod(argv[3], NULL);
cout << "val: " << val << endl; // prints val: 1234
```

If you would like to specify command line arguments 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:



Configuration: Active(Debug)

Platform: Active(Win32)

Debugger to launch: Local Windows Debugger

Command	$(TargetPath)
Command Arguments	rsslist.txt 3.5
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No

Command Arguments
The command line arguments to pass to the application.

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

Requirements and Other Thoughts

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

1. The entire project can be completed in under 600 lines of C++ code, so if your program is getting much larger than this, talk to a TA – you're probably doing something wrong.
2. Your Impl classes (e.g. RSSProcessorImpl, NewsClusterImpl) and your main() function must never ***directly*** use your other Impl classes. They MUST use our provided cover classes instead:

   INCORRECT:

   ```
       class NewsAggregatorImpl
       {
           …
           void someFunction(...)
           {
   ```

```
            RSSProcessorImpl rpi(…); // BAD!
            …
        }
    };
```

CORRECT:

```
    class NewsAggregatorImpl
    {
        …
        void someFunction(...)
        {
            RSSProcessor rp(…);      // GOOD!
            …
        }
    };
```

3. Make sure to implement and test each class independently of the others. Once you get the simplest class (RSSProcessor) coded, get it to compile and test it with a number of different unit tests. Only after you have your first class working should you advance to the next class.
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 stack, a set, a StringMapper? How will you use these data structures? Plan before you program!
5. Make heavy use of the STL for this project (with the exception of the STL map/unordered_map classes – you must NOT use the map or unordered_map classes at all in your solution; use your StringMapper 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 news clusters, 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)!
6. If you decide to use unordered_set, the following will #include the appropriate header under either Visual C++ or g++:
```
    #ifdef _MSC_VER
    #include <unordered_set>
    #else
    #include <tr1/unordered_set>
    #endif
    using namespace std::tr1;
```
7. You may not modify any of our provided classes such as WordExtractor, Keyword, Cluster, RSSProcessor, NewsCluster, NewsAggregator, or the HTTPController class *in any way*. You must use them as is in your program.
8. For 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 (as you did for Project #3)
9. For Visual C++, make sure to add *wininet.lib* to the set of input libraries, by going to Project → Properties → Linker → Input → Additional Dependencies or you'll get a linker error!

10. For MAC OS X and LINUX, you can build your program from the command line with the command (all on one line):
```
g++ -o proj4 main.cpp RSSProcessor.cpp NewsCluster.cpp
        NewsAggregator.cpp
```
and then run it with
```
./proj4 rsslist 3.5
```
11. The std::string type offers some functions you may find useful:
```
      //           11111111112
      //  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 map class instead of creating your own StringMapper class if necessary to save time. Or implement just your news clustering algorithm but don't worry about identifying the top keywords. You can still get a good amount of partial credit if you implement most of the project. But whatever you do, make sure that whatever you turn in BUILDS without errors!

What to Turn In

You should turn in **six** files: five source files and one report file:

NewsAgg.cpp	Contains your news aggregator implementation
NewsCluster.cpp	Contains your news cluster implementation
RSSProcessor.cpp	Contains your RSS processor implementation
Mapper.h	Contains your StringMapper template class declaration and implementation
main.cpp	Contains your main routine for the news aggregator
report.doc or report.docx or report.txt	Your report

You may make whatever changes you wish to the Impl classes in the first four .cpp files, except that the only .cpp file that may mention *Whatever*Impl is Whatever.cpp (e.g., only RSSProcessor.cpp may mention RSSProcessorImpl). Otherwise, you may freely add new data members, new member functions, new classes, new non-member support functions (like an operator< function), etc. You may #include the http.h and provided.h headers that we wrote for you (and that we will use when we build your programs for testing).

You must also submit a report that details the following:

1. Describe the algorithm and data structures you used for your:

    - **NewsAggregatorImpl class**: Describe the data structures and algorithms you used to efficiently construct, order and return your clusters and keywords to the user.
    - **NewsCluster class**: Describe the data structures and algorithms you used to maintain and grow individual clusters.
    - **StringMapper class**: Describe the data structures and algorithms you used to implement your hash table or binary search tree mapper class.

2. Provide a thorough list of test cases that you used to test each of your classes and explain why you performed each test.
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 based on the style of your code and the appropriateness of your data structures and algorithms
- 5% of your grade will be assigned based on the documentation of your data structures and algorithms in your report
- 10% of your grade will be assigned based on the thoroughness of your test cases

Good luck!