Project #01:   Spam filter program

Complete By:	**Bonus**:	**Saturday, Jan. 18th @ 11:59pm (+10%)**
	On-time:	**Monday, Jan. 20th @ 11:59pm**
	Late:	**Tuesday, Jan. 21st @ 11:59pm (-10%)**
Assignment:	**C++ program to filter email based on spam lists**	
Policy:	**Individual work only, late work *is* accepted**	
Submission:	**a single "main.cpp" file via Gradescope**	

Assignment

Most email systems automatically filter "spam" into a separate folder. The filtering process is typically based on a "spam list" that identifies spam when email is loaded into your inbox. In this assignment you're going to create a C++ program that is able to

1. load a spam list,
2. display the contents of a spam list,
3. check a single email address to see if it's spam,
4. filter an email list and output the resulting emails to a file

For example, to the right is a screenshot of the program in action, loading "spamlist1.txt", displaying the contents, checking a couple email addresses to see if they are spam, and then filtering an email list in "emails1.txt" to eliminate spam emails. The currently loaded spam list defines what is considered spam; the user can load different spam lists.

Pay close attention to the formatting of the output, as the Gradescope submission system will require you to adhere to this output format. Also note that when the program ends, some stats are output based on how efficiently the problem was solved. If M is the # of emails processed and N is the # of emails in the spam list, your solution should insert at most O(N) elements, and access at most O(MlgN) elements.

```
** Welcome to spam filtering app **

Enter command or # to exit> load spamlist1.txt
Loading 'spamlist1.txt'
# of spam entries: 12

Enter command or # to exit> display
aaa.com:user
bestbuy.com:coupons
bestbuy.com:offers
groupon.com:*
groupon.com:reseller
important.com:dont_ignore
massemail.com:*
organicfoods.com:noreply
uic.edu:accc
uic.edu:chancellor
uic.edu:rewards
xyz.com:user1

Enter command or # to exit> check offers@groupon.com
offers@groupon.com is spam

Enter command or # to exit> check user2@xyz.com
user2@xyz.com is not spam

Enter command or # to exit> filter emails1.txt output1.txt
# emails processed: 21
# non-spam emails:  11

Enter command or # to exit> #
************************************************
ourvector<class std::basic_string> stats:
 # of vectors created:    1
 # of elements inserted: 12
 # of elements accessed: 92
************************************************
```

The goals of this assignment are to (1) practice creating **functions**, (2) work with C++ **strings** and string functions, (3) work with **vectors**, and (4) read and write files. Many of you are new to C++, so working with strings, vectors, and files are an important introduction to C++.

When you load a spam list, you are required to input the data into 1 or more vectors. You cannot use any other data structure, and you cannot use a C-based array --- you must use a vector. In fact, you'll use an implementation of vector we provide: **ourvector<T>**, which is available on the course dropbox under Projects, project01-files. Your program must support 4 commands: load, display, check, and filter. Each of these commands must be implemented by a function that is called by main(). You can (and should) have additional functions, but these 4 are required. Finally, since efficiency is an important component of this class, you are required to search the spam list using **binary search**. Binary search requires only O(lgN) time, whereas linear search takes O(N) time. Example: if N = 1,000,000 elements, binary search takes roughly 20 comparisons to find an element, whereas linear search takes an average of 500,000 comparisons. Big difference.

Note that a spam list is guaranteed to be in sorted order by domain, and if two spam entries have the same domain, they are ordered by username (where "*" is naturally sorted first). This implies that binary search can be used without the need to sort the spam list.

Two larger lists are provided for you to test the efficiency of your program. As shown in the screenshot below, the file "spamlist_10k.txt" contains 10,000 spam entries. This list is then used to filter an email list of 1,000 emails ("emails_1k.txt"). Let M = 1,000, and N = 10,000. If your solution is efficient, searching the spam list once should required O(lgN) = $\log_2(10,000)$ = 14 comparisons / vector accesses. This search is repeated 1,000 times (once for each email), so an efficient filtering should require roughly 14,000 vector accesses. As shown below in the stats, the vector was accessed 12,949 times, denoting an efficient O(M * lgN) solution.

```
** Welcome to spam filtering app **

Enter command or # to exit> load spamlist_10k.txt
Loading 'spamlist_10k.txt'
# of spam entries: 10000

Enter command or # to exit> filter emails_1k.txt output.txt
# emails processed: 1000
# non-spam emails:  596

Enter command or # to exit> #
*********************************************************
ourvector<class std::basic_string> stats:
 # of vectors created:   1
 # of elements inserted: 10000
 # of elements accessed: 12949
*********************************************************
```

A spam list contains 1 or more lines, where each line contains a single string denoting a spam address. There are 2 possible formats for a spam address: domain:* or domain:username. The "*" means any username associated with that domain is spam; otherwise an email is spam only if the username and domain both match. Here's the contents of "spamlist1.txt" that was shown earlier (and again in screenshot to right):

```
aaa.com:user
bestbuy.com:coupons
bestbuy.com:offers
groupon.com:*
groupon.com:reseller
important.com:dont_ignore
massemail.com:*
organicfoods.com:noreply
uic.edu:accc
uic.edu:chancellor
uic.edu:rewards
xyz.com:user1
```
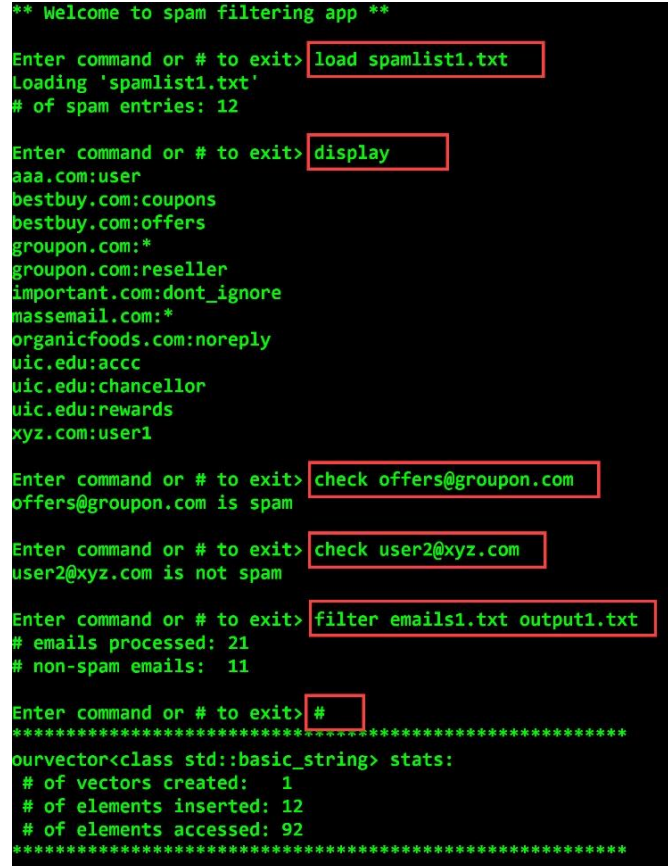
The file is guaranteed to be in alphabetical order by domain; if 2 strings have the same domain, they will be ordered by username ("*" will proceed any other username).

```
** Welcome to spam filtering app **

Enter command or # to exit> load spamlist1.txt
Loading 'spamlist1.txt'
# of spam entries: 12

Enter command or # to exit> display
aaa.com:user
bestbuy.com:coupons
bestbuy.com:offers
groupon.com:*
groupon.com:reseller
important.com:dont_ignore
massemail.com:*
organicfoods.com:noreply
uic.edu:accc
uic.edu:chancellor
uic.edu:rewards
xyz.com:user1

Enter command or # to exit> check offers@groupon.com
offers@groupon.com is spam

Enter command or # to exit> check user2@xyz.com
user2@xyz.com is not spam

Enter command or # to exit> filter emails1.txt output1.txt
# emails processed: 21
# non-spam emails:  11

Enter command or # to exit> #
************************************************************
ourvector<class std::basic_string> stats:
 # of vectors created:    1
 # of elements inserted: 12
 # of elements accessed: 92
************************************************************
```

The other possible input file is an email list, which is processed by the filter command. An email list consists of 1 or more lines, which each line contains 3 values:

```
MsgID  EmailAddress  Subject
```

For example, here is the contents of the "email1.txt" file used in the screenshot shown above:

```
9 jhummel2@uic.edu Question about grading...
10 coupons@bestbuy.com This week only: free TV with purchase
15 pooja@piazza.com Please update profile
16 abcd@uic.edu This is not spam, seriously, not spam
17 adef@uic.edu This is also not spam, seriously, spam it is not
21 reed@uic.edu Can you help?
22 offers@bestbuy.com Please buy more stuff!
23 techsupport@bestbuy.com Please update your profile
33 user@xyz.com Important msg frm user
41 user1@xyz.com A really non-important msg
42 user1@xyzz.com Seriously, this one is important
43 reseller@groupon.com Yup, you got me, spam
99 fred@massemail.com you'll like this offer
121 jon@groupon.com Please update your pwd
123 sue@groupons.com You can't miss this deal
```

```
155 chancellor@uic.edu Just kidding
198 accc@uic.edu VPN is now operational
199 accc@uic.edu VPN is down
254 important@uic.edu Tuition waivers for all!
260 rewards@uic.edu Subway sandwiches free every friday
261 testing@uic.edu Testing
```

The message ID is always an integer, the email address is a single string (i.e. no spaces), and the subject is a string containing 1 or more words. The message IDs may or may not be in ascending order. As noted later on in the section entitled "Learning C++", use the **>>** operator to input the message ID and email address, and use the **getline()** function to input the multi-word subject.

Based on a spam list and an email list, the filter command filters the email and produces a corresponding output file containing the email list with spam emails removed. Given the spam list and email lists shown above, the program would filter and produce the following output file ("output1.txt" in the screenshot):

```
9 jhummel2@uic.edu  Question about grading...
15 pooja@piazza.com  Please update profile
16 abcd@uic.edu  This is not spam, seriously, not spam
17 adef@uic.edu  This is also not spam, seriously, spam it is not
21 reed@uic.edu  Can you help?
23 techsupport@bestbuy.com  Please update your profile
33 user@xyz.com  Important msg frm user
42 user1@xyzz.com  Seriously, this one is important
123 sue@groupons.com  You can't miss this deal
254 important@uic.edu  Tuition waivers for all!
261 testing@uic.edu  Testing
```

Sample input files --- both spam lists and email lists --- are provided in the course dropbox under Projects, project01-files.

Program functionality

Email filtering is based on the currently loaded spam list. Your program should support filtering by different spam lists. An example is shown here:

Hint: your program should have a spam vector with the currently loaded spam list. When the user loads a

new spam list, clear the vector before loading the new spam entries.

    Your program is required to work properly if the input or output files cannot be opened, the user specifies an unknown command, the user inputs a badly-formed email address, or the user tries to display / check / filter before loading a spam list. Examples with expected output:

```
** Welcome to spam filtering app **

Enter command or # to exit> display

Enter command or # to exit> check user@domain
user@domain is not spam

Enter command or # to exit> filter emails1.txt output3.txt
# emails processed: 21
# non-spam emails:  21

Enter command or # to exit> load spamlist.txt
**Error, unable to open 'spamlist.txt'

Enter command or # to exit> load spamlist1.txt
Loading 'spamlist1.txt'
# of spam entries: 12

Enter command or # to exit> filter x y
**Error, unable to open 'x'

Enter command or # to exit> check fred
fred is not spam

Enter command or # to exit> list
**invalid command

Enter command or # to exit> help
**invalid command

Enter command or # to exit>
```

Keep in mind that our Gradescope submission system will be auto-grading your output, so you need to match the responses shown above *exactly*.

You may be new to C++, or perhaps new to the features of C++ needed in this assignment. Homework #01 and #02 will help you prepare, so we strongly recommend you complete those HW exercises before starting.

You'll need some string processing, namely finding characters within a string, and extracting a substring. While you can certainly write these functions yourself, it's expected that you'll use the .find() and .substr() functions built into the **string** class provided by C++: http://www.cplusplus.com/reference/string/string/. Don't forget to #include <string>.

Your solution is required to store all data in a **vector<T>** class --- to be precise, in an vector-compatible implementation we are providing: **ourvector<T>**. For the purposes of this assignment, always start with an empty vector, and then add data to the vector by calling the **push_back()** member function. When you need to access an element of the vector, use the **.at()** function, or the more convenient and familiar **[ ]** syntax. To empty a vector in order to load a new spam list, use the **.clear()** function. For more info on vector<T>: http://www.cplusplus.com/reference/vector/vector/. Don't forget to #include "ourvector.h", which is available on the course dropbox under Projects, project01-files.

Finally, to work with files, #include <fstream>. To read from a file, use an **ifstream** object, and use the **>>** operator when inputting a single value (e.g. integer or single word). When you need to input 1 or more words into a single string variable, such as the email subject, use **getline(infile, variable)**. Example of reading a file that contains one string per line, where each string is just one word (i.e. no spaces):

```
string infilename;
infilename = ...;

ifstream infile(infilename);  // use infile object to read from file

if (!infile.good())  // unable to open input file:
{
  ...
}
else
{
   string oneWord;

   infile >> oneWord;
   while (!infile.eof())  // until we hit the end-of-file:
   {
      .
      .  // process
      .

      infile >> oneWord;
   }

   infile.close();
}
```

To write to an output file, use an **ofstream** object and the **<<** operator. Here's an example of writing the string "hello world!" to a file:

```
string outfilename;
outfilename = ...;

ofstream outfile(outfilename);  // use outfile object to write to file

if (!outfile.good())  // unable to open output file:
{
    ...
}
else
{
    outfile << "hello world!" << endl;

    outfile.close();  // make sure contents are written by closing file:
}
```

Programming Environment

You are free to program on whatever platform you want, using whatever compiler / programming environment you want. Input files and "ourvector.h" are available in the course dropbox under Projects, project01-files. If you do not have a C++ programming environment that you like, we recommend **Codio**; see the appendix at the end of this document for getting started with Codio.

Be aware that we are using **Gradescope** as our grading platform, and it's common for C++ programs to "work on my platform" but fail on Gradescope. This is due to logic errors in *your* program, not an error with Gradescope. The most common mistake is a memory-related error, e.g. using an uninitialized variable or accessing memory outside the bounds of an array or via an invalid pointer. These errors are hard to find; the tools **valgrind** and **cppcheck** (available on Codio) can help.

Gradescope is running on Ubuntu Linux, and we are compiling via **g++** with **-std=C++11**. Do not ask us to change the C++ version; we are compiling against C++ 11.

Requirements

1. You must use **ourvector<T>** ("ourvector.h") for storing all data. No other data structures may be used.

2. Each input file may be opened and read exactly once; store the data in ourvector<T> if need be.

3. You must use binary search to search the spam list, and you must write the algorithm yourself; do not call the built-in binary search function provided by C++.

4. The # of inserts must be O(N), where N is the # of spam list entries in the input file(s).

5. The # of accesses must be O(MlgN), where M is the # of emails in the input file(s).

6. Your main.cpp program file must have a header comment with your name and a program overview. Each function must have a header comment above the function, explaining the function's purpose, parameters, and return value (if any). Inline comments should be supplied as appropriate; comments like "*declares variable*" or "*increments counter*" are useless. Comments that explain non-obvious assumptions or behavior *are* appropriate.

7. Each command (load, display, check, filter) must be implemented using a function; this implies a complete solution must have at least 4 functions.

8. No global variables; use parameter passing and function return.

9. The **cyclomatic complexity** (CCN) of any function may not exceed 15, including main(). This will be reported when you submit on Gradescope, and you'll be warned if you exceed this threshold. In short, cyclomatic complexity is a representation of code complexity --- e.g. nesting of loops, nesting of if-statements, etc. You should strive to keep code as simple as possible, which generally means encapsulating complexity within functions (and calling those function) instead of explicitly nesting code. Here's an example of simpler code with low CCN:

```
while (…)
{
   if (searchFunctionFindsWhatWeNeed(…))
     doSomething();

   next value;
}
```

Here's an example of complex code with high CCN:

```
while (…)
{
   for (…)  // loop to do search
   {
      if (search condition is met)
      {
         for (…) // now do something:
         {
            …
         }
      }
   }

   next value;
}
```

As a general principle, if we see code that has **more than 2 levels** of explicit looping --- an example of which is shown above --- we will score that code as 0, even if it's technically correct. The solution is to move one or more loops into a function, and call the function.

Have a question? Use Piazza, not email

As discussed in the syllabus, questions must be posted to our course Piazza site — questions via email will be ignored. Remember the guidelines for using Piazza:

1. _Look before you post_ — the main advantage of Piazza is that common questions are already answered, so search for an existing answer before you post a question. Posts are categorized to help you search, e.g. "HW".

2. Post publicly — only post privately when asked by the staff, or when it's absolutely necessary (e.g. the question is of a personal nature). Private posts defeat the purpose of piazza, which is answering questions to the benefit of everyone.

3. Ask pointed questions — do not post a big chunk of code and then ask "help, please fix this". Staff and other students are willing to help, but we aren't going to type in that chunk of code to find the error. You need to narrow down the problem, and ask a pointed question, e.g. "on the 3rd line I get this error, I don't understand what that means…".

4. Post a screenshot — sometimes a picture captures the essence of your question better than text. Piazza allows the posting of images, so don't hesitate to take a screenshot and post; see http://www.take-a-screenshot.org/ .

5. Don't post your entire answer / code — if you do, you just gave away the answer to the ENTIRE CLASS. Sometimes you will need to post code when asking a question --- in that case post only the fragment that denotes your question, and omit whatever details you can. If you must post the entire code, then do so privately --- there's an option to create a private post ("visible to staff only").

Grading, electronic submission, and Gradescope

Your score on this project is based on two factors: (1) correctness as determined by your Gradescope submission, and (2) manual review by the TAs for commenting, style, and approach (e.g. use of functions and efficiency of solution). The entire project is worth 150 points: 100 points for correctness, and 50 points for commenting, style, and approach. In this class we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness. Example: if you submit to Gradescope and your score is a reported as a 0, then that's your correctness score. The only way to raise your correctness score is to re-submit and obtain a higher score by passing one or more test cases. You have unlimited submissions on this project assignment.

Note that the TAs will also review for adherence to requirements; breaking a requirement can result in a final score of 0 out of 150. We take all requirements seriously.

A bonus of 10% is earned for submitting by the "Bonus" deadline on page 1. Bonus points are accumulated and can be applied to a future project. To earn bonus points, your submission must post before the Bonus deadline, earn a correctness score of 100, *and* meet all project requirements --- in this case required # of functions, efficiency, etc. In other words, you cannot submit a correct but inefficient solution and expect to earn bonus points. Bonus points are reserved for early and well-written submissions.

To submit to Gradescope, you must first create an account; check your UIC email for an invitation to

Gradescope. If you cannot find this invitation email, post privately on Piazza and we will send another invite; you cannot register yourself. Gradescope is running on Ubuntu Linux, and we are using **g++** with **-std=C++11**. Do not ask us to change the C++ version; we are compiling against C++ 11. Our grading script assumes your program will be found in a single "main.cpp" file; do not create additional .h, .hpp, or .cpp files. You may submit as many files as you want (e.g. the .zip produced by Codio's Project >> Export as Zip command), but we extract and run only your "main.cpp" file.

By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can **activate** an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it will be considered academic misconduct.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you <u>extend or receive any kind of unauthorized assistance</u>. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .

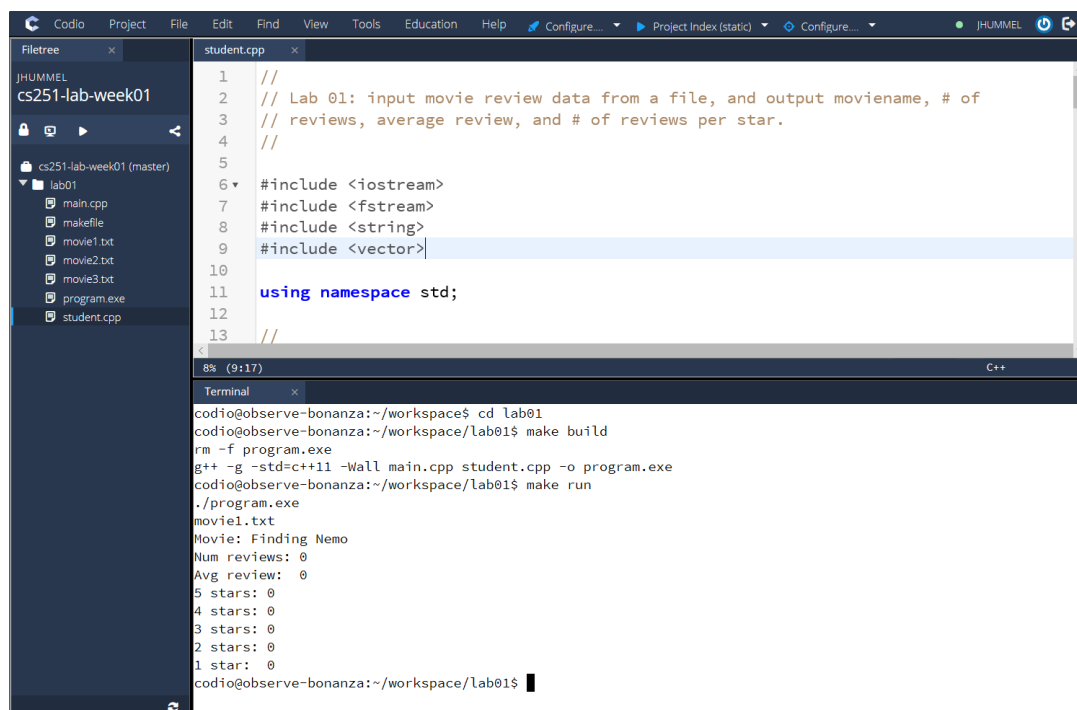# Appendix: Codio Programming Environment

Here's a quick summary of how to get started with Codio. Codio is cloud-based, and accessible via a web browser. It's platform-neutral and works from any platform, but you must be online to use it. The first step is to create a Codio account and join the class ("CS 251 Spring 2020"):

   https://codio.com/p/join-class?token=forum-jamaica

Be sure to register using your UIC email address, especially since you may be using this account in future CS classes. The above link will provide access to Codio, and the resources associated with CS 251.

Once you successfully login to Codio, you'll see the project "**cs251-project01-spam-filter**" pinned to the top of your dashboard. This represents a container-based C++ programming environment --- think light-weight virtual machine (VM). When you are ready to start programming, click "Ready to go" and Codio will startup the VM and within a few seconds you'll have a complete Ubuntu environment at your disposal. In particular, you'll have access to C++ via the GNU g++ compiler. You also have super-user (root) access, so you can install additional software if you want ("sudo apt-get install XYZ").

Once I login, I normally split the right-side into 2 windows: the top as my editor pane, and the bottom as my terminal window. This can be done via the View menu, Panels, Split Horizontal. Then click on the bottom window, drop the Tools menu, and select Terminal. Here's a snapshot showing "student.cpp" open in the editor, and the terminal window open below it:

For the purposes of project #01, nothing is provided other than a makefile and some input files for testing purposes. Use the File menu to create a new "main.cpp" file, and then open a terminal windows and use the provided makefile to compile and run: type "make build" to compile, and assuming no compilation errors, type "make run" to execute.