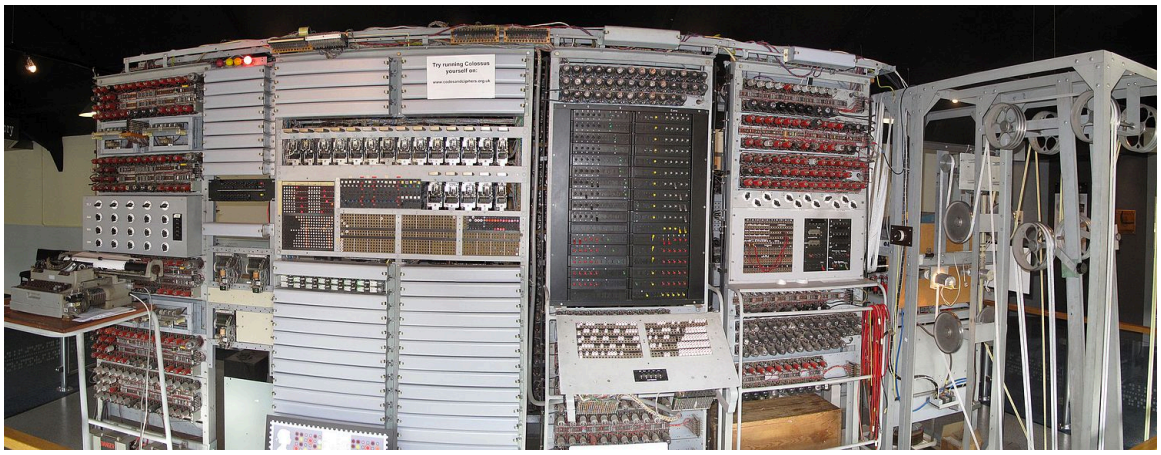
Project 4
Cracked

Time due: 11 PM Thursday, March 15



Before writing a single line of code, you MUST first read AND THEN RE-READ the *Requirements and Other Thoughts* section.

Introduction

Did you read the text in red on the previous page?

Before writing a single line of code, you **must** first read **and then re-read** the *Requirements and Other Thoughts* section. Print out that page, tape it to your wall or on the mattress above your bunk bed, etc. And read it over and over (it's nearly as exciting as Carey's novel, The Florentine Deception).

The NachenSmall Software Corporation has been contacted by the U.S. National Security Agency (NSA) and asked to create a software program that can crack encrypted messages sent by rogue groups around the world. These rogue groups have always encrypted their secret messages using advanced public-key cryptography programs, but recently have decided to go retro, ditching their complex encryption schemes (which may be vulnerable to quantum computers) for a Simple Substitution Cipher (which has been used for thousands of years).

If you've never heard of a Simple Substitution Cipher, it's pretty simple. To use it, you start by creating a mapping from each letter of the alphabet to a letter in the alphabet, like follows:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
C	B	Y	Z	F	Q	I	H	V	D	P	O	M	L	N	K	E	R	X	T	S	G	A	U	W	J

This table indicates that 'A' maps to 'C', 'B' maps to 'B', 'C' maps to 'Y', and so on. Given a message we want to encrypt, we can look up each letter of our unencrypted (*plaintext*) message on the top row and replace it with the associated letter on the bottom row. To decrypt a message, its recipient can look up each letter of the encrypted (*ciphertext*) message on the bottom row and replace it with its original plaintext letter from the top row.

In such a mapping, each plaintext letter on the top row **must** map to a unique letter on the bottom row, and each ciphertext letter on the bottom row **must** map to a unique letter on the top. If two different plaintext letters mapped to the same ciphertext letter, then the recipient would not know which plaintext letter that ciphertext letter should decrypt to. Similarly, if two different ciphertext letters were associated with the same plaintext letter, then with only 26 possible letters, some plaintext letter would not have a corresponding ciphertext letter. Notice that some letters, like B above, may map to themselves, so with the mapping above, the letter B in a plaintex message would be represented as B in the ciphertext message. Also notice that there is no symmetry requirement – for example, in the mapping above you can see that plaintext C on the top row maps to ciphertext Y on the bottom row, but in this particular mapping, plaintext Y on the top row does not map to ciphertext C on the bottom row.

Once you have created such a mapping, you then distribute this secret mapping to two people who want to secretly communicate: the sender (we'll call her Alice) and the

recipient (we'll call him Bob). Once both participants have a copy of the mapping, they can exchange secret messages without being spied upon.

To encrypt a message like "I LOVE UCLA CS", Alice simply looks up each letter of her message in the top row of the table, translates it to the corresponding letter in the bottom row, and writes this ciphertext letter. Thus, to encrypt the message "I LOVE UCLA CS" Alice would look up 'I' on the top row, and see that it maps to 'V' in the bottom row. She'd then look up 'L' in the top row, and see it maps to 'O' on the bottom row, and so on. The resulting encrypted message would be "V ONGF SYOC YX".

Alice then transmits this encrypted message to Bob, keeping her secret from the NSA.

To decrypt an encrypted message, Bob simply takes the same table and looks each ciphertext letter up in the bottom row to find its corresponding plaintext letter in the top row. So, first Bob looks up 'V' (the first letter of the encrypted message he received) in the bottom row, and sees it maps to I in the top row. Then he looks up O in the bottom row, and sees it translates to L in the top row, etc. He processes each such letter in the encrypted message in this way to arrive at the Alice's original, plaintext message.

Now if you think about it, there are 26! (about 4×10^{26}) possible mapping tables that Alice and Bob can pick from to transmit their secret messages! Why? Because there are 26! ways of shuffling all 26 letters on the bottom row of the mapping table.

For your final project of CS32, Winter '18, your job is to write a set of C++ classes capable of cracking an arbitrary encrypted message, like "V ONGF SYOC YX" and printing out the original message: "I LOVE UCLA CS". Since some encrypted messages have more than one English translation (since without the encryption key, there may be two or more possible translation tables that all produce English translations), your solution will be required to print out all possible English translations, sorted alphabetically.

If you're able to prove to NachenSmall's reclusive and bizarre CEO, Carey Nachenberg, that you have the programming skills to build the encryption-cracking tool described in this specification, he'll hire you and you'll be famous… at least among the people who work at the top secret National Security Agency.

How Do You Crack a Simple Substitution Cipher?

With 26! possible mapping tables, it would seem like it would take centuries to take an encrypted message like this

```
Vxgvab sovi jh pjhk cevc andi ngh iobnxdcjnh cn bdttook jb
pnio jpfnicvhc cevh vha nceoi nho cejhy.
```

and try all of the possible mappings to find the right one to decrypt it to its original form:

```
Always bear in mind that your own resolution to succeed is
more important than any other one thing.
```

(Abraham Lincoln wrote this.)

But, as it turns out, cracking a message encrypted with a simple substitution cipher is easier than you might think… especially if you have access to a list of valid words and use some clever data structures and algorithms.

To make things even easier, in this assignment, we will guarantee that every word in the encrypted messages you have to crack is contained in a list of English words that we will give you. We'll also make things easier for you and ensure that every encrypted word is separated by a space, digit or limited set of punctuation marks so it is clear where one word ends and the next starts. Imagine if you had to crack this instead of the version above with spaces:

```
Vxgvabsovijhpjhkcevcandinghiobnxdcjnhcnbdttookjbpnio
jpfnicvhccevhvhanceoinhocejhy.
```

So how do you actually crack a simple substitution cipher? **Below is one possible algorithm that we require you to use in your project.** It relies upon a list of English words that we will provide you.

As you'll see, this algorithm is recursive and may require a helper function to implement.

string Crack(string ciphertext_message, vector<string>& output):

1. Start with an empty mapping (in this example, we'll show the ciphertext letters on the top row and the plaintext letters on the bottom for clarity – this is the opposite of the table shown in the introduction):

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

   This is our initial input mapping – initially we know nothing about which plaintext letter each ciphertext letter maps to, which is why we show ? for each plaintext letter.

2. Break up ("tokenize") the ciphertext message into separate words and pick a ciphertext word w from the message that (a) has not yet been chosen and (b) has the most[1] ciphertext letters for which we don't have any translation. (When your algorithm first starts, this results in your picking the longest ciphertext word.)
3. Translate the chosen encrypted word using the current mapping table to get a partial decrypted translation. If there's no translation for a ciphertext letter on the

[1] This algorithm will work even if the chosen word does not have the *most* unknown letters, but choosing the one with the most increases the chance that an incorrect mapping will be rejected sooner.

top of your mapping table, then simply use a ? for the plaintext character. (So when the algorithm first starts, your translation of the word will be all ?s.)

4. Using our provided English word list and some clever data structures, create a collection C of all words in the word list that could possibly match the ciphertext word compatibly with the partially decrypted version of the word. For example:
    a. If the ciphertext word were *qbemme* and your current mapping table is empty, then C would contain the words *apollo, blotto, career, chilli, djinni, fusees, grotto, guerre, indeed, piazza, pierre, quagga, steppe, troppo.* All of these words have the same third and sixth letter (different from the other letters), the same fourth and fifth letter (different from the others), and first and second letters different from all others. Since we have no mappings in our mapping table, all of these choices are potential translations for *qbemme*.
    b. If the ciphertext word were *qbemme* and your current mapping table contains the two ciphertext-to-plaintext mappings e→o and b→r, then C would contain the words *grotto,* and *troppo.* Both words have the same pattern as above, and are consistent with our mappings from e→o and b→r. C does not contain *piazza*, since b→r, not b→i.
    c. If the ciphertext word were *qbemme* and your current mapping table has only the mapping e→i, then C would contain the words *chilli* and *djinni.* Both words have the pattern as above and are consistent with our mapping from e→i.
    d. If the ciphertext word were *qbemme* and your current mapping table has the mapping e→q, then C would be empty, since there are no words in the list that have a q in the third and sixth letter positions.
5. If there are no words in the list that could match the ciphertext word compatibly with the partially decrypted version of the word, so C is empty, then your current mapping table *must* be wrong, and your function can throw it away and return to the previous recursive call.
6. For each candidate plaintext word p in your collection C for ciphertext word w:
    a. Create a temporary mapping table by making a copy of the input mapping table and then updating the temporary mapping table by adding the mapping that would occur if your chosen encrypted word w actually did translate into the candidate English word p. Specifically:

       For each letter w[i] in your original ciphertext word w:
        i. Determine the letter p[i] in p that w[i] should map to.
        ii. If your current temporary mapping table does not yet contain a mapping between w[i] and p[i], then add one.
        iii. Otherwise, if the temporary mapping table already contains the same mapping between w[i] and p[i], then do nothing.
        iv. Otherwise, if the temporary mapping table already contains a conflicting mapping from w[i] to some letter that is not p[i], or from some letter that is not w[i] to p[i], then there is no way that the candidate word can work with the currently selected encoding,

so throw away your temporary mapping, and go back to step 6 to try another candidate p.

This now gives you a partial mapping (which may or may not be the right one) between the ciphertext alphabet (on the top row), and your proposed plaintext translations (on the bottom row).

b. Next use this partial proposed translation table to translate your entire ciphertext message into a proposed plaintext message.
c. Evaluate your just-decrypted message to see if all fully-translated words (those with no ?s in them) are in the word list.
    i. If at least one fully-translated word cannot be found in the word list, then the temporary mapping we chose is wrong. So the chosen word p can *not* be a valid candidate for the encrypted word w. Throw away your temporary mapping, and go back to step 6 to try another candidate p.
    ii. If all of the fully-translated words are found in our word list, but the message has *not* been completely translated (some words still have ?s in them that were not translated), then this partial solution is promising, as is the current proposed mapping table. Recursively call step 2, passing in our temporary mapping as the new input mapping (we will build upon this new mapping as we recurse deeper and deeper).
    iii. If every single word of the decrypted message was fully translated and in the word list, record this as a valid solution (potentially one of many) for eventual output to the user. Then discard the current temporary mapping and go back to step 6 to try other candidate translations for the chosen word w and their mappings.

In a nutshell, this algorithm works as follows: we pick a word from our ciphertext message, identify all of the candidate English translations for that word, then we update our mapping table based on each candidate and then apply that translation to the rest of the encrypted message. If a translation due to a given candidate results in all valid (or possibly valid) decrypted words, then we're on the right track, so we pick another word from our encrypted message, find all candidate English translations for it (that are consistent with our current mapping table), then update our mapping table based on each candidate (so now the mapping table reflects both the first candidate and this new one) and then apply that translation to the rest of the encrypted message. If this results in all valid (or possibly valid) decrypted words, then we're still on the right track, so we pick yet another encrypted word, identify all of the candidate translations for that word, then we update our mapping table based on that candidate (so now the mapping table reflects all three candidates) and then apply that translation to the rest of the encrypted message. Eventually we'll either reach a dead end because our translation will generate a word not in the word list (and we'll backtrack, trying other candidates), or we'll arrive at a completely translated sentence of valid words. In the latter case, we save the valid solution for later output, then continue hunting for other possibilities.

Let's trace through our algorithm for the following sentence and see how it might work:

```
y qook ra bdttook yqkook.
```

Each bulleted item is prefixed with its [step number] in the proposed algorithm above so you can see which step of the algorithm is doing what. Further, each step of the algorithm is listed as either *top-level*, *inside recursive call*, or *inside second recursive call* to help you understand where you are in the recursion.

- [Step 1, top level] Start with an empty mapping:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

- [Step 2, top level] Pick the word with the most untranslated letters. We haven't translated any words yet, so that would be *bdttook*, which has 7 untranslated letters.
- [Step 3, top level] Translate the chosen word, *bdttook*, to its decoded equivalent using the current mapping table. We get ?????? since we have no translations for b, d, t, o, or k in our mapping table.
- [Steps 4 and 5, top level] Locate all valid English words that might match *bdttook* and which are consistent with our current translation of ???????. For this example, let's assume there are only two possible matches: *balloon* and *succeed.*
- [Step 6a, top level] First try *balloon*, and assume that *bdttook* maps to *balloon*. Update our mapping table appropriately, mapping b to b, d to a, t to l, o to o, and k to n and leaving the rest of the letters untranslated:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	B	?	A	?	?	?	?	?	?	N	?	?	?	O	?	?	?	?	L	?	?	?	?	?	?

- [Step 6b, top level] Use the new mapping table to decode the input message and see what we get:

  Ciphertext message: *y qook ra bdttook yqkook.*
  Partially-decrypted message: *? ?oon ?? balloon ??noon.*

- [Steps 6c, top level] Check all fully-translated words to make sure they're in the word list. Right now, only *balloon* has been fully translated, and it's in the word list, so we'll continue with the current mapping table and recursively call our function with this mapping table.
- [Step 2, inside recursive call] Pick the word with the most untranslated letters. That would be a tie between *ra* and *yqkook*. Let's assume algorithm picks *yqkook*, although either choice would work.

- [Step 3, inside recursive call] Translate the current word, *yqkook*, to its decrypted equivalent using the current mapping table. We get *??noon* since we have translations for k → n and o → o, but not for the other letters y and q.
- [Steps 4 and 5, inside recursive call] Locate all words in the word list that could match yqkook **and** that are compatible with our current translation of *yqkook*: *??noon*. Wait! As it turns out, the word list contains NO English words that are six letters long and end with the suffix "noon". So our current translation **must** be invalid. Return from our function, discarding our current translation map. We're now back to an empty translation table:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

- [Step 6a, top level] Ok, we just tried *balloon*, so now let's try our second candidate, *succeed*, and assume that *bdttook* maps to *succeed*. Update our mapping table appropriately:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	S	?	U	?	?	?	?	?	?	D	?	?	?	E	?	?	?	?	C	?	?	?	?	?	?

- [Step 6b, top level] Use the new mapping table to translate the input message based on our proposed mapping table:

  Ciphertext message: *y qook ra bdttook yqkook.*
  Partially-decrypted message: *? ?eed ?? succeed ??deed.*

- [Steps 6c, top level] Check all fully-translated words to make sure they're in the word list. Right now, only *succeed* has been fully translated, and it's in the word list, so we'll continue with the current mapping table and recursively call our function with this mapping table.
- [Step 2, inside recursive call] Pick the word with the most unknown/untranslated letters. That would be a tie between *ra* and *yqkook*. Let's assume our algorithm picks *yqkook*, although either choice would work.
- [Step 3, inside recursive call] Translate the current word, *yqkook*, to its decrypted equivalent using the current mapping table. We get *??deed* since we have translations for k → d and o → e, but not for the other letters y and q.
- [Steps 4 and 5, inside recursive call] Locate all words in the word list that could match yqkook **and** that are compatible with our current translation of *yqkook*: *??deed*. Searching our word list results in only one word that matches this pattern: *indeed.*
- [Step 6a, inside recursive call] Update our mapping table appropriately, setting y to i and q to n:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	S	?	U	?	?	?	?	?	?	D	?	?	?	E	?	N	?	?	C	?	?	?	?	I	?

- [Step 6b, inside recursive call] Use the new proposed mapping table to decode the input message (newly discovered letters shown in red):

  Ciphertext message: *y qook ra bdttook yqkook.*
  Partially-decrypted message: *i need ?? succeed indeed.*

- [Steps 6c, inside recursive call] Check all fully-translated words to make sure they're in the word list. We look up *i*, *need, succeed* and *indeed* in the word list and find all of them. This is promising! So we'll continue with the current mapping table and recursively call our function with this mapping table.
- [Step 2, inside second recursive call] Pick the word with the most untranslated letters. That would be the only untranslated word left: *ra*
- [Step 3, inside second recursive call] Translate the current word, *ra*, to its decrypted equivalent using the current mapping table. We get *??* since we have no translations for r and a.
- [Steps 4 and 5, inside second recursive call] Locate all words in the word list that could match ra **and** that are compatible with our current translation of *ra*: ??. That results in many words like *it*, *to, at, of, go,* etc.
- [Step 6a, inside second recursive call] For each such candidate word, create a new temporary mapping table and update it appropriately. Let's assume our algorithm starts with *ra* → *it*, so let's map r to i and a to t. Whoops! We have a problem – our table

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	S	?	U	?	?	?	?	?	?	D	?	?	?	E	?	N	?	?	C	?	?	?	?	I	?

  already has a mapping from y to i, so it's impossible for r to also map to i; having two encrypted characters (y and r) map to the same plaintext character (i) is incompatible with the way the substitution cipher works. Let's abandon this mapping (*ra* → *it*) and continue back with step 6a.
- [Step 6a, inside second recursive call] For the next candidate word, create a new temporary mapping table and update it appropriately. Let's assume our algorithm continues *with ra* → *to*, so let's map r to t and a to o. This yields the following updates to the mapping table.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
O	S	?	U	?	?	?	?	?	?	D	?	?	?	E	?	N	T	?	C	?	?	?	?	I	?

- [Step 6b, inside second recursive call] Use the new mapping table to translate the input message based on our proposed mapping table (newly discovered letters shown in red):

  Ciphertext message: *y qook ra bdttook yqkook.*
  Partially-decrypted message: *i need to succeed indeed.*

- [Step 6c, inside second recursive call] Check all fully-translated words to make sure they're in the word list. We look up *i*, *need, to, succeed* and *indeed* in the word list and find all of them. Every single word has been translated, so we have a full translation! Produce the current translation (*i need to succeed indeed*) as a valid solution, discard our current translation table, and then proceed to the next candidate word.
- [Step 6a, inside second recursive call] For each such candidate word, create a new temporary mapping table and update it appropriately. We'll continue with *ra* → *at*, so let's map r to a and a to t:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
T	S	?	U	?	?	?	?	?	?	D	?	?	?	E	?	N	A	?	C	?	?	?	?	I	?

- [Step 6b, inside second recursive call] Use the new mapping table to translate the ciphertext message (newly discovered letters shown in red):

  Ciphertext message: *y qook ra bdttook yqkook.*
  Partially-decrypted message: *i need at succeed indeed.*

- [Step 6c, inside second recursive call] Check all fully-translated words to make sure they're in the word list. We look up *i*, *need, at, succeed* and *indeed* in the word list and find all of them. Every single word has been translated, so we have a full translation! Produce the current translation as a valid solution. Note that this translation may not make much sense, but it *is* a valid translation as far as our program is concerned – every translated word is in the word list. Now proceed to the next candidate word.
- [Step 6a, inside second recursive call] For each such candidate word, create a new temporary mapping table and update it appropriately. We'll continue with *ra* → *of*, so let's map r to o and a to f:
- And so on….

As you can see, by using the above algorithm, your program can quickly home in on the proper mapping. Our program might output something that looks like this:

i need ab succeed indeed.
i need ah succeed indeed.
… *items skipped*
i need go succeed indeed.
… *items skipped*
i need to succeed indeed.
… *items skipped*

There are many other approaches that can be used to crack substitution ciphers such as letter and bigram (e.g., qu, th, er, …) frequency analysis, and simulated annealing. Feel free to look them up. But for this problem, you **must** use the algorithm described above.

What Do You Need to Do?

Question: So, at a high level, what do you need to build to complete Project #4?

Answer: You'll be building five complete classes that together can be used to crack a simple substitution cipher. These classes are detailed below:

Class #1: You need to build a map class template named *MyHash* that implements a templated, resizable, open hash-table-based map:

You need to create a new map class template, based on a resizable *open hash table*. By resizable, we mean if the hash table's load factor gets too high (because you've inserted too many items) the hash table will automatically resize itself (that is, allocate a new, bigger dynamic array and move all of the entries over to the new table, then delete the old dynamic array). Since the hash table is templated, it can be used to map any type of data to any other type of data. You **must not** use any STL container classes to implement your *MyHash* class.

Class #2: You need to build a class named *Tokenizer*:

You need to build a "tokenizer" class that can be used to chop up a string (e.g., a sentence) into a vector of tokens (e.g., words in the sentence) - that's basically what tokenizing means. Typically when you tokenize a string, you must specify what characters separate tokens (e.g., spaces, periods, commas, quotes, etc.). Then the tokenizer cuts up the string by gathering only the tokens, excluding the separators.

Class #3: You need to build a class named *WordList*:

You need to build a word list class which is used to load up the contents of a text file consisting of English words and allow the user to efficiently search for words in that word list (e.g., "Is the word "onomatopoeia" in the word list", or "Give me all the words in the list that follow the letter pattern 122134, like 'GOOGLE' or 'TOOTHY'"). Your *WordList* implementation **must** use your *MyHash* class template in the implementation of its data structures and **must not** use any STL containers in its data members.

Class #4: You need to build a class named *Translator*:

You need to build a class that maintains a partially or completely filled mapping table, like this:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	S	?	U	?	?	?	?	?	?	D	?	?	?	E	?	?	?	?	C	?	?	?	?	?	?

which can be used to translate a ciphertext message such as:

y qook ra bdttook yqkook.

to a (possibly partially) decrypted message like this:

? ?eed ?? succeed ??deed.

The *Translator* class allows the user to specify one or more new character mappings, e.g., that Q should map to N, which will update the table:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	S	?	U	?	?	?	?	?	?	D	?	?	?	E	?	N	?	?	C	?	?	?	?	?	?

Class #5: You need to build a class named *Decrypter*:

Finally, you need to build a decrypter class that uses your other classes to process a ciphertext message and produce all possible valid decryptions of that ciphertext to the user. It **must** present these decryptions in alphabetical order. Your decrypter class **must** use the algorithm described above to produce these decrypted messages. This class **must not** use the STL map or unordered_map classes.

What Will We Provide?

We'll provide a simple *main.cpp* file that lets you test your overall decoder implementation. You can compile this *main.cpp* file (and our other provided header file) with your own source files to build a complete working test program. You can then run this program from the Windows command line or in a macOS Terminal window. You can read about our provided *main.cpp* in the *Test Harness* Section of this document.

We'll also provide you with a text file listing English words as well as some abbreviations (e.g., CA for California), one per line.

We'll also provide you with a header file named *provided.h* that will declare and implement the four classes *Tokenizer*, *WordList*, *Translator*, and *Decrypter*. Stop and think about that sentence. Is it saying that we're writing most of this project's code for you? Well, no. The code in each of these classes simply delegates work to a corresponding class that you will write. Let's see how will work, and then explain why we're doing it this way.

Part of the *provided.h* file will be (edited to make this example simpler):

```
class WordListImpl;

class WordList
{
public:
    WordList();
    ~WordList();
```

```
    bool loadWordList(std::string filename);
    bool contains(std::string word) const;
private:
    WordListImpl* m_impl;
};
```

This class contains only one data member, a pointer to a *WordListImpl* object (which you can define however you want in *WordList.cpp*). As shown in the skeleton *WordList.cpp* that you will complete and turn in, the member functions of *WordList* simpy delegate their work to functions in *WordListImpl*[2]:

```
#include "provided.h"

    // You may add additional headers, using statements, helper functions or
    // classes, etc.

class WordListImpl
{
    // You may implement this however you want, provided that WordList objects
    // behave as required by the spec.
};

  // Implementation of WordList that delegates everything to WordListImpl.
  // You will not want to change anything below this line.

WordList::WordList()
{
    m_impl = new WordListImpl;  // create a new implementation object
};

WordList::~WordList()
{
    delete m_impl;  // destroy the implementation object
}

bool WordList::loadWordList(string filename)
{
    return m_impl->loadWordList(filename);  // delegate work to the
                                            // implementation object
}

bool WordList::contains(string word) const
{
    return m_impl->contains(word);  // delegate work to the implementation object
}
```

There's an important restriction we will place on you: Other than *WordList.cpp* itself, no source file that you turn in may contain the name *WordListImpl* or that of any new helper functions or classes you might introduce in *WordList*. So some other file whose code wants to use a word list **must not** have

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

```
void f()
{
    WordListImpl wl;  // No! Other files must not use the name WordListImpl
    ...
}
```

but instead should have

```
void f()
{
    WordList wl;
    if ( ! wl.loadWordList("wordlist.txt"))
        ...
}
```

When this code constructs a *WordList*, the skeleton code creates a *WordListImpl* object and has code you write construct it. When this code calls *WordList::loadWordList*, the skeleton code calls the *WordListImpl::loadWordList* you write, and returns the value that you have *WordListImpl::loadWordList* return.

Why are we having you do this instead of just implementing *WordList* directly? For two reasons: one that restricts you and one that helps you.

The restriction: We want to make sure you don't introduce a public function into one of the required classes, e.g. *WordList*, that you then call in another, e.g. *Decrypter*; we want you to use the interface we provided. By having none of your implementation go in *WordList*, we can put the *WordList* class in a file *provided.h* that you will not turn in. We will build our tester using the *provided.h* we gave you, so it's useless for you to modify it, since we'll never use or even see those modifications. All of your implementation related to *WordList* will go into *WordList.cpp*. By forbidding you to mention *WordListImpl* outside of *WordList.cpp* (which is easy for us to detect), code in other files can't use a *WordListImpl* (whose interface you might have extended) instead of a *WordList*.

The help: Suppose you cannot figure out how to write a correct *WordListImpl*. If your *Decrypter* code would work correctly if it used a correctly implemented *WordList*, but fails with your incorrectly implemented one, would you lose points on *Decrypter*, too? No, because to test your implementation of one class, such as *Decrypter*, we will build *your* implementation (*DecrypterImpl* in this example) together with *our* correct implementations of the other classes. This is much easier for us when a class's interface and implementaion are separated the way we're doing it.

The *MyHash* class template isn't subject to this separation, since template implementations aren't put in a separate .cpp file; they go in the header file. Your other classes will use the *MyHash* class template directly; there's no need for a separate implementation class.

Details: The Classes You MUST Write

You **must** write correct versions of the following classes to obtain full credit on this project. Your classes **must** work correctly with our provided code, and you **must not** modify our provided *main.cpp* or *provided.h* to make them work with your code. Doing so will result in a **zero score** on that part of the project.

MyHash Class Template

You must write a class template named *MyHash* that implements an open hash table that resizes its internal hash table size (the number of buckets) when its load factor gets too high.

Your *MyHash* class template **must** have the following public interface:

```
template <class KeyType, class ValueType>
class MyHash
{
public:
    MyHash(double maxLoadFactor = 0.5);
    ~MyHash();
    void reset();
    void associate(const KeyType& key, const ValueType& value);
    const ValueType* find(const KeyType& key) const;
    ValueType* find(const KeyType& key);
    int getNumItems() const;
    double getLoadFactor() const;
};
```

You **must not** add any additional public member functions or data members to this class. You may add as many *private* member functions or data members as you like.

You **must not** use any STL container classes to implement your *MyHash* class.

MyHash(double maxLoadFactor) and ~MyHash()

You **must** implement a constructor and destructor for your *MyHash* class:

The constructor **must** initialize your hash table, setting the size of its initial dynamic array to 100 buckets. The user specifies the maximum load factor of the hash table by passing it into the constructor, and the default maximum load factor must be 0.5 if no parameter is passed in. This method **must** run in O(B) time where B is the number of buckets in the table. If the user passes in a zero or negative value for the maximum load factor, you must use a maximum load factor value of 0.5. If the user passes in a load factor greater than 2, then you must use 2.0 for the maximum load factor.

The destructor **must** free all memory associated with hash table. This method **must** run in O(B) time where B is the number of buckets in the table.

void reset()

Your hash table's *reset()* method **must** free all of the memory associated with the current hash table, then allocate a new empty hash table of the default size of 100 buckets. The maximum load factor is unchanged. This method **must** run in O(B) time where B is the number of buckets in the table to be reset.

void associate(const KeyType &key, const ValueType &value)

The *associate()* method adds a new association to the hash table associating the specified *key*, e.g. "122134", with the associated *value*, e.g., "google". If you pass in a key which is already in the hash table, then this function will update the association in the hash table, discarding the old value that the key mapped to and replacing it with the new value. So a given key in the hash table may map to only one value.

If inserting a new key → value association into the hash table would cause the hash table's load factor to exceed its maximum load factor (as specified during construction), then your function **must** allocate a new dynamic array that has double the number of buckets the current dynamic array has, and must move all of the items from the old array to the new array. It **must** delete the contents of the old array.

Be careful: Items that hashed to bucket X in the old array may not hash to the same bucket in the new array because its number of buckets is different! You'll have to re-compute the proper location for every key → value association in the new array.

Assuming items in your hash table are roughly uniformly distributed across all the buckets, your *associate()* function **must** run in O(1) time if the number of buckets does not change. (In a pathological case, such as when most keys hash to just a few buckets, this function may run in O(X) time where X is the number of items in the hash table.) In those cases where the *associate()* call results in the number of buckets changing, your *associate()* function **must** run in O(B) time, where B is the number of buckets in the hash table.

const ValueType* find(const KeyType& key) const
ValueType* find(const KeyType& key)

The *find()* method attempts to locate the item with a key equal to *key* in the hash table, and if successful, returns a pointer to the value associated with that key. If the key can not be found within the hash table then this function returns *nullptr*. If the hash table is modifiable, the value pointed to by the return value will be modifiable; if the hash table is not, the value is not. The skeleton we provide does a little C++ magic so that the second overload is implemented in terms of the first one, which is the only one of the two that you will implement.

Assuming items in your hash table are roughly uniformly distributed across all the buckets, your *find()* function **must** run in O(1) time. (In a pathological case, such as when most keys hash to just a few buckets, this function may run in O(X) time where X is the number of items in the hash table.)

int getNumItems() const

The *getNumItems()* method returns the number of unique key → value associations that the hash table holds. Note, that overwriting an old association k1→v1 with a new association k1→v2 does not increase the number of items in the hash table. It simply replaces the old association with the new one. Your *getNumItems()* function **must** run in O(1) time. Note: This function does not return the number of buckets in the table, but the actual number of stored associations.

double getLoadFactor() const

The *getLoadFactor()* method **must** return the actual current load factor of the hash table based on its current number of buckets and the number of associations it holds. Your *getLoad()* function **must** run in O(1) time.

Hash Functions

We are providing you with three hash functions (for common types like *string*, *int*, and *char*) which you may call from your *MyHash* class to obtain an unsigned int value between 0 and roughly 4 billion:

```
unsigned int hash(const std::string& s)
{
	return std::hash<std::string>()(s);
}

unsigned int hash(const int& i)
{
	return std::hash<int>()(i);
}

unsigned int hash(const char& c)
{
	return std::hash<char>()(c);
}
```

These hash functions leverage the STL's *hash* class which is declared in the header `<functional>`. You'll need to transform the values returned by our hash functions to produce a bucket number within the range of your array.

You **must** put these implementations in one of your .cpp files, not in *MyHash.h*. It will simplify our grading scripts if, when you turn in your project, the .cpp file you put the hash*()* implementations in is *WordList.cpp*.

To avoid getting certain strange compilation or linker errors, in the implementation of any member function of *MyHash* that calls *hash()*, you **must** have a prototype declaration for the hash function, such as:

```
unsigned int hash(const KeyType& k);
```

Here's a little hint for this:

```
template<typename KeyType, typename ValueType>
class MyHash
{
    ...
    unsigned int getBucketNumber(const KeyType& key) const
    {
        unsigned int hash(const KeyType& k);  // prototype
        unsigned int h = hash(key);
        ...
    }
    ...
}
```

While some older compilers are forgiving if you don't declare the prototype in the implementation of a class template member function, Standard C++ has arcane "two-phase lookup" rules for names used in templates that often surprise even experienced C++ programmers.

If you'd like to define your own hash function, you **must** define it in one of your .cpp files, **not** in *MyHash.h*. Each such hash function **must** be named *hash()* and **must** have the signature:

```
unsigned int hash(const YourTypeHere& k);
```

However, for this project, it's unlikely your design would require a hash function for any type other than string, int, and char.

Tokenizer Class

You must implement a class named *Tokenizer* that can be used to tokenize strings. Your *Tokenizer* class **must** have the following public interface:

```
class Tokenizer
{
public:
   Tokenizer(std::string separators);
```

```
   std::vector<std::string> tokenize(const std::string& s) const;
};
```

You **must not** add any additional public member functions or data members to this class other than a destructor, should you need one. You may add as many *private* member functions or data members as you like.

Tokenizer(std::string separators)

The *Tokenizer* constructor must initialize a new Tokenizer object. When you construct a Tokenizer object, you pass in a list of separators, e.g., `" ,.$-!;"`. This function **must** run in O(P) time, where P is the number of separators.

```
void f()
{
    Tokenizer t(" ,.!");  // spaces, commas, periods, and exclamation points will
                          // be separators
    …
}
```

std::vector<std::string> tokenize(const std::string &s) const

The *tokenize()* method is responsible for breaking the input string *s* into a set of non-empty token strings based on the separators provided in the constructor, returning them in a vector. This function **must** run in O(SP) time where S is the number of characters in the input string *s* and P is the number of different separators (you can implement it to run in O(S) time if you want a small challenge).

Here's an example of how it might be used:

```
void f()
{
    TokenizerImpl t(" ,.!");
    vector<string> v = t.tokenize("This,, is a    test! It's the… best!");
      // v now contains "This", "is", "a", "test", "It's", "the", and "best"

    string s = "!!!!!";
    v = t.tokenize(s);
      // v is now empty
}
```

WordList Class

A *WordList* object is responsible for loading all of the words from our provided word list and storing these words in data structures that enable the object to be used to efficiently look up words and word patterns. Your *WordList* class **must** have the following public interface:

```
class WordList
{
public:
    WordList();
    bool loadWordList(std::string filename);
    bool contains(std::string word) const;
    std::vector<std::string> findCandidates(std::string cipherWord,
                                            std::string currTranslation) const;
};
```

You **must not** add any additional public member functions or data members to this class other than a destructor, should you need one. You may add as many *private* member functions or data members as you like.

Your *WordList* implementation **must** use your *MyHash* class template in the implementation of its data structures and **must not** use any STL containers as data members, although you **may** use *string*, *vector*, *list*, and *array* in template arguments to your *MyHash* class template. Within the implementations of your *WordList* member functions, you **must not** use any STL containers other than *string*, *vector*, *list*, and *array*. You **must** use your *MyHash* class template for any map-like data structures.

WordList() and possibly ~WordList()

The *WordList* constructor must initialize the *WordList* object. You probably won't need to write much code to do this. This spec imposes no requirements on its time complexity.

Should you need to write one to properly dispose of all the memory a *WordList* object uses, you must declare and implement a destructor that does so. This spec imposes no requirements on its time complexity.

bool loadWordList(std::string filename)

Your *loadWordList()* method **must** load the contents of the indicated word list text file into data structures that enables your other methods to meet the requirements stated in the sections below. Every time *loadWordList()* is called, it **must** discard the old contents of the *WordList* data structures and start fresh (so any words contained in the data structures before are thrown out, and it reverts to empty data structures before loading the new words). If this function is able to open the file and load the words, it must return true; otherwise it must return false.

If a line of the word list file contains a character that is not a letter or an apostrophe, ignore that line; otherwise, that line is to be considered one of the words in the word list.

If the word list text file contains W unique words, then this method **must** run in O(W) time. (For the purpose of this requirement, we're assuming that there is some constant such that no word in the file has a length that exceeds that constant.)

For more information on how to open and read data from a text file, see the File I/O writeup on the class web site.

bool contains(std::string word) const

The *contains()* method is used to determine if the specified word is in the word list. The method **must** return true if the specified word is in the currently-loaded word list, or false otherwise. This function **must** be case-insensitive, so searching for "nerd" and "NeRd" should result in the same result (true, if the word list file contained the word "nerd" or "NERD" or "nERd", for example).

This this method **must** run in O(1) time regardless of the size of the word list. (For the purpose of this requirement, we're assuming that there is some constant such that no word has a length that exceeds that constant.)

std::vector<std::string> findCandidates(std::string cipherWord, std::string currTranslation) const

The *findCandidates()* method returns a vector of those words in the loaded word list that have the same letter pattern as the *cipherWord* and are consistent with *currTranslation*, the current plaintext translation of the *cipherWord*. The *cipherWord* parameter should contain only letters (a-z or A-Z) and apostrophes, and the *currTranslation* parameter should contain only letters, apostrophes and ? characters, and be the same length as *cipherWord*; if not, this function must return an empty vector.

Consider each word *w* in the word list. For *w* to be returned in the vector, it must have the same letter pattern as *cipherWord* (e.g., "indeed" has the same letter pattern as "abcddc", but not "cbcddc"), and for each position *j* of *currTranslation*:

- If *currTranslation[j]* is a letter, this indicates that we believe that the plaintext translation of *cipherWord[j]* is that letter, and *w[j]* must be that letter, in upper or lower case. (Thus, g matches G or g, and G matches G or g.) If *cipherWord[j]* is not also some letter, this function must return an empty vector.
- If *currTranslation[j]* is a ?, this indicates that we don't know the plaintext translation of *cipherWord[j]*, but *w[j]* must be a letter. If *cipherWord[j]* is not a letter, this function must return an empty vector.
- If *currTranslation[j]* is an apostrophe, *w[j]* must be an apostrophe. If *cipherWord[j]* is not an apostrophe, this function must return an empty vector.

This method **must** be case-insensitive with respect to *cipherWord* and *currTranslation*; that is, it must work regardless of the case of the letters in either string.

For instance, let's assume we provide this function with a *cipherWord* of "xyqbbq" and a *currTranslation* of "??????" indicating that we have no idea what plaintext letters any of the ciphertext letters (b, q, x or y) translate to. Assuming you loaded our provided word list, this method returns a vector consisting of the following words:

apollo
blotto
career
chilli
djinni
fusees
grotto
guerre
indeed
piazza
pierre
quagga
steppe
troppo

These are all of the words in our word list that are potential matches for xyqbbq. Notice that all of these words have the same letter pattern as the ciphertext word xyqbbq. Specifically, they consist of two unique letters, followed by a letter that's repeated in the third and sixth positions, and a different letter that's repeated in the fourth and fifth positions.

Here's another example: Suppose you provided this function with the *cipherWord* "xyqbbq" and a *currTranslation* of "??o??o", indicating that you believe that q translates to o, but have no idea about the translation of the other ciphertext letters. Assuming you loaded our provided word list, this method returns a vector consisting of the following words:

apollo
blotto
grotto
troppo

Or, assume you provided this function with the *cipherWord* "xyqbbq" and a *currTranslation* of "???mm?", indicating that you believe that b translates to m, but have no idea about the translation of the other encrypted letters. Assuming you loaded our provided word list, this method returns an empty vector, since there are no words in our list with m in the fourth and fifth positions, and with the same other pattern of letters as the ciphertext word.

Your function **must** run in O(Q) time where Q is the number of words in the word list that match the letter pattern of *cipherWord.* Note that Q is much, much smaller than the number of words in the word list.

You might be asking right now: How can I implement this efficiently without having to iterate through the entire list of N words? Here's a hint: Use letter patterns to index your word list data!

What's a letter pattern? It's a way of describing the pattern of repeated letters in a word.

For example, the word "indeed" has four distinct letters: i, n, d, and e. Two of the letters, d and e, are each repeated twice in particular positions. The word "grotto" has the same pattern: four distinct letters, with two each being repeated in particular positions, the sa,e positions as in "indeed". For each of those words, we could form a pattern by labeling each distinct letter with a symbol chosen from a particular sequence, such as αβγδεζηθ..., in the order of the letter's first occurrence in the word. For "indeed", we'd label i with α, n with β, d with γ, and e with δ, yielding αβγδδγ. For "grotto", we'd label g with α, r with β, o with γ, and t with δ, yielding αβγδδγ — the same pattern! Of course, it may be more convenient to get our labels from a sequence other than αβγδεζηθ..., such as 01234567... or abcdefgh... or ABCDEFGH...
Suppose we use ABCDEFGH as the source of our labels, so that "indeed" and "grotto" have the pattern ABCDDC. Consider the ciphertext word "xyqbbq". It has the pattern ABCDDC! Using a pattern scheme and a clever hash-based data structure, we could easily locate all plaintext words that have the same pattern as a given ciphertext word. We could then further reduce the collection of viable words by eliminating those that are inconsistent with *currTranslation*.

WordList – Putting It All Together

Here's how the WordList class be used:

```
void f()
{
    WordList wl;

    if ( ! wl.loadWordList("wordlist.txt"))
    {
        cout << "Unable to load word list" << endl;
        return;
    }
    …
    if (wl.contains("onomatopoeia"))
        cout << "I found onomatopoeia!" << endl;
    else
        cout << "Onomatopoeia is not in the word list!" << endl;

    string cipher = "xyqbbq";
    string decodedSoFar = "?r????";
    vector<string> v = wl.findCandidates(cipher, decodedSoFar);
    if (v.empty())
        cout << "No matches found" << endl;
    else
    {
        cout << "Found these matches:" << endl;
        for (size_t k = 0; k < v.size(); k++)
            cout << v[k] << endl;  // writes grotto and troppo
    }
}
```

Translator Class

The *Translator* class is responsible for translating a ciphertext message into either a partially- or fully-decoded English version.

To perform a translation, first you provide a *Translator* object with a group of ciphertext-to-plaintext character mappings, e.g., D→E, H→R, L→D:

```
void f()
{
    Translator t;  // Define a translator object
      // Submit the mapping D→E, H→R, L→D
    t.pushMapping("DHL", "ERD");
    ...
}
```

The call to *pushMapping()* tells your *Translator* object that the translation of each ciphertext letter in the first string is the corresponding plaintext letter in the second string. The *Translator* object **must** maintain a mapping table (or, as you'll see, multiple tables) that represents this collection of character mappings after the above code runs.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	?	?	E	?	?	?	R	?	?	?	D	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Once you have pushed one or more character mappings into the mapping table, you can then use the object to translate a message:

```
void f()
{
    Translator t;
    t.pushMapping("DHL", "ERD");  // D→E, H→R, L→D
    string secret_msg = "Hdqlx!";
    string translated_msg = t.getTranslation(secret_msg);
    cout << "The translated message is: " << translated_msg;
}
```

The output from the above program would be:

```
The decrypted message is: Re?d?!
```

Notice how the *Translator* object was able to translate H into R, d into e, and l into d. However, also notice that since you never submitted a translation for the Q or X characters, the *Translator* object doesn't know how to translate those characters into their plaintext equivalents, so it instead it translates both to ? characters. Also notice that the translator maintains the case of the original message, so the upper case H is translated into a capital R, the lower case d is translated to a lower case e, and so on. Also notice that non-letters (including spaces, punctuation, numbers, and the ? sign, etc.) are all copied as is from the argument string to the result string. **These are all requirements for your implementation of the class.**

You are allowed to submit more than one collection of character mapping to a *Translator* object. For example, we could do the following:

```
void f()
{
    Translator t;

      // Submit the first collection of character mappings
    t.pushMapping("DHL", "ERD");  // D→E, H→R, L→D

    string secret = "Hdqlx!";
    cout << t.getTranslation(secret) << endl;  // writes Re?d?!

      // Submit a second collection of character mappings
    t.pushMapping("QX", "AY");  // Q→A, X→Y

    cout << t.getTranslation(secret) << endl;  // writes Ready!
}
```

Submitting an additional collection of character mappings to a *Translator* object causes:

- your current mapping table to be saved onto the top of a stack,
- a copy of the current mapping table to be made,
- your new collection of character mappings to be added to the copy of the mapping table, and
- the updated copy then becoming the current mapping table.

So the new current mapping table looks like this:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	?	?	E	?	?	?	R	?	?	?	D	?	?	?	?	A	?	?	?	?	?	?	Y	?	?

resulting in the second call to *getTranslation()* above to return `Ready!`.

You can push as many such additional collections of character mapping as you like. And of course, if you can push a collection of character mappings, you can pop them, too:

```
void f()
{
    Translator t;

      // Submit the first collection of mappings
    t.pushMapping("DHL", "ERD");  // D→E, H→R, L→D

    string secret = "Hdqlx!";
    cout << t.getTranslation(secret) << endl;  // writes Re?d?!

      // Submit a second collection of mappings
    t.pushMapping("QX", "AY");  // Q→A, X→Y

    cout << t.getTranslation(secret) << endl;  // writes Ready!
```

```
    // Pop the most recently pushed collection
    t.popMapping();
    cout << t.getTranslation(secret) << endl;  // writes Re?d?!
}
```

Popping will discard the most recent collection of character mappings that was submitted (the top of the stack), resstoring the second most recent mapping table as the current one:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	?	?	E	?	?	?	R	?	?	?	D	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Popping once more will take you back to the empty mapping table:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

The *getTranslation()* function must always use the top mapping table on the stack to translate an encrypted message to its decrypted equivalent.

Your *Translator* class **must** have the following public interface:

```
class Translator
{
public:
    Translator();
    bool pushMapping(std::string ciphertext, std::string plaintext);
    bool popMapping();
    std::string getTranslation(const std::string& ciphertext) const;
};
```

You **must not** add any additional public member functions or data members to this class other than a destructor, should you need one. You may add as many *private* member functions or data members as you like.

Your *Translator* implementation **must not** the STL container *stack*; you must implement a stack some other way.

Translator() and ~Translator()

The *Translator* constructor must initialize the *Translator* object. This spec imposes no requirements on its time complexity. The constructor must ensure that the current mapping table represents this collection of character mappings:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Should you need to write one to properly dispose of all the memory a *Translator* object uses, you must declare and implement a destructor that does so. This spec imposes no requirements on its time complexity.

bool pushMapping(std::string ciphertext, std::string plaintext)

The *pushMapping()* function takes a collection of new mappings from ciphertext letters to plaintext letters. For each position *j* in *ciphertext*, *ciphertext[j]* maps to *plaintext[j]*. If the parameters are not the same length, or if either contains a non-letter, or if the new character mappings they specify, together with the current collection of character mappings, would be inconsistent, then this function must return false without changing the state of the *Translator* object. A collection of character mappings is inconsistent if one ciphertext letter would map to two or more different plaintext letters, or if two or more different ciphertext letters map to the same plaintext letter. Here's an example:

```
void f()
{
    Translator t;
    t.pushMapping("DHL", "ERD");  // D→E, H→R, L→D
    if ( ! t.pushMapping("QX", "RY"))  // Q→R, X→Y
        cout << "Both H and Q would map to R!" << endl;
      // The current mapping is still D→E, H→R, L→D with no
      // mapping for Q or X
    cout << t.getTranslation("HDX") << endl;  // writes RE?
    if ( ! t.pushMapping("H", "S"))  // H→S
        cout << "H would map to both R and S!" << endl;
}
```

If none of the conditions that require a return value of false hold, then this function must return true after saving the current mapping table onto a stack and updating the current mapping table so that it incorporates the collection of new character mappings specified by the parameters. These character mappings are case-insensitive, so submitting a mapping of D → e is the same as submitting d → E, or d → e, or D → E.

This function MUST run in O(N+L) time, where N is the length of the parameter strings and L is the number of letters in the English alphabet. Since typically N <= L and L is a constant, of course, this is essentially O(1).

bool popMapping()

The *popMapping()* must return false without changing the state of the *Translator* object if the stack of mapping tables is empty. Otherwise, it must pop the most-recently pushed mapping table from the stack, make it the current mapping table, and return true.

This function MUST run in O(L) time, where L is the number of letters in the English alphabet. Since L is a constant, of course, this is essentially O(1).

std::string getTranslation(const std::string& ciphertext) const

The *getTranslation()* function translates its argument string according to the current mapping table and returns the resulting string of the same length. Each character in the ciphertext argument results in a character appearing in the corresponding position of the result string according to the following:

- If the ciphertext character is a letter that maps to a plaintext letter in the current mapping, then that plaintext letter appears, in the same case as in the ciphertext string. (Thus, if D→E in the current mapping, ciphertext D results in E, while ciphertext d results in e.
- If the ciphertext character is a letter with an unknown translation in the current mapping, a ? appears.
- If the ciphertext character is not a letter, that character appears, unchanged.

Decrypter Class

The *Decrypter* class is responsible for cracking a ciphertext message and printing out all plaintext translations of that message. Your *Decrypter* class **must** have the following public interface:

```
class Decrypter
{
public:
    Decrypter();
    bool load(std::string filename);
    std::vector<std::string> crack(const std::string& ciphertext);
};
```

You **must not** add any additional public member functions or data members to this class other than a destructor, should you need one. You may add as many *private* member functions or data members as you like.

Your *Decrypter* implementation **must not** the STL containers *map* or *unordered_map*.

Decrypter() and ~Decrypter()

The *Decrypter* constructor must initialize the *Decrypter* object. You probably won't need to write much code to do this. This spec imposes no requirements on its time complexity.

Should you need to write one to properly dispose of all the memory a *Decrypter* object uses, you must declare and implement a destructor that does so. This spec imposes no requirements on its time complexity.

bool load(std::string filename)

Your *load()* method **must** load the contents of the indicated word list text file, to be used during the decrypting process. (Presumably, it will delegate this task to a *WordList* object.) Every time *load()* is called, it **must** discard the old list of words and start fresh before loading the new words. If this function successfully loads the words, it must return true; otherwise it must return false.

If the word list text file contains W unique words, then this method **must** run in O(W) time. (For the purpose of this requirement, we're assuming that there is some constant such that no word in the file has a length that exceeds that constant.)

std::vector<std::string> crack(const std::string& ciphertext)

The *crack()* method is responsible for taking in an encrypted message, such as this:

> *G lbbm qfbbm GLMBBM!*

and then finding **all** valid decryptions for this message, returning them **in sorted order** in the result vector. For example, given the above message and our provided word list file, vector returned must consist of the following strings in this order:

> *I need bleed INDEED!*
> *I need breed INDEED!*
> *I need creed INDEED!*
> *I need freed INDEED!*
> *I need greed INDEED!*
> *I need speed INDEED!*
> *I need steed INDEED!*
> *I need treed INDEED!*
> *I need tweed INDEED!*

Notice that it is possible for the returned vector to be empty; this means that no plaintext message consisting only of words in our list could possibly have been encrypted to produce the given ciphertext.

Your implementation **must** use the decryption algorithm that we provided in the introduction of this document in order to crack encrypted messages.

Every character of the *ciphertext* argument should be an upper or lower case letter, an apostrophe, a digit, a space, or one of the following punctuation characters:

```
,;:.!()[]{}-"#$%^&
```

While *ciphertext* may contain apostrophes, these are to be considered as part of a word (such as can't or won't), so must not be used as a separator when tokenizing. Digits, spaces, and the punctuation characters listed above are all separators when tokenizing.

If *ciphertext* contains a character not listed above (e.g., ? or @ or a tab), the vector returned by this function may contain whatever you want; this spec imposes no requirement in that case other than that the function must return something, as opposed to crashing or going into an infinite loop. We expect that for many people's programs there's no need to check for this situation, since the way their code works without this checking, a ciphertext with an unlisted character will be processed as containing a ciphertext word that can not be translated to a word on the word list, so the vector that *crack()* returns will be empty, which is acceptable.

We will not impose any big-O requirement on this function, since you **must** implement the algorithm we specified. Computing its big-O complexity is beyond the scope of this course.

Test Harness

We have graciously provided you with a simple test harness (in *main.cpp*) that lets you test your overall Cracked implementation. You can compile this *main.cpp* file with your source files to build a complete working test program. You can then run this program from the Windows command line or the shell running in a macOS Terminal window or under Linux.

In the *main.cpp* file is a constant WORDLIST_FILE defined as "wordlist.txt". If you leave it this way, then when you run the test harness to decrypt a message, it will use a file named *wordlist.txt* in the same directory as your executable file as the word list file. If you would like to use a different file, change the string and rebuild. If the program isn't finding the file, then change the string to be the full path name to the file, such as "C:/CS32/P4/mywords.txt" or "/Users/fred/p4/mywords.txt".

Our test harness has two different functions:

You can use our harness to encrypt a message

Once you've implemented the *Translator* class, you can build the test harness and run it with the –e option to generate a random mapping and use it to encrypt a message. If the name of your executable file is proj4, you'd say:

```
proj4 –e "Please encrypt this message for me"
```

This writes something like the following:

```
Lzdkgd dyrmjls shcg xdggkud fpm xd!!
```

If you run the program again, you'll get other versions of the encrypted message:

```
Ojvgtv vcrpxok kfwt uvttgiv byp uv!!
```

You can use our harness to decrypt an encrypted message into all possible plaintext versions

Once you've implemented all of your other classes, you can build the test harness and run it with the –d option to use it to decrypt a message:

```
proj4 –d "Lzdkgd dyrmjls shcg xdggkud fpm xd!!"
```

This writes something like the following:

```
Please encrypt this message bur me!!
Please encrypt this message for me!!
Please encrypt this message fur me!!
Please encrypt this message our me!!
Please encrypt thus message fir me!!
Please encrypt thus message for me!!
Please encrypt tows message bur me!!
Please encrypt tows message fir me!!
Please encrypt tows message fur me!!
Please encrypt tubs message fir me!!
Please encrypt tubs message for me!!
Please encrypt tubs message hor me!!
Please encrypt twos message bur me!!
Please encrypt twos message fir me!!
Please encrypt twos message fur me!!
Please entropy yuks message fir me!!
```

Notice how there may be multiple plaintexts that are consistent with the encrypted message, which must be printed in alphabetical order.

Encrypted Messages to Test With

Here are some interesting encrypted messages that you can test your program with.

```
Trcy oyc koon oz rweelycbb vmobcb, wyogrcn oecyb; hjg ozgcy tc
moox bo moya wg grc vmobck koon grwg tc ko yog bcc grc oyc trlvr
rwb hccy oecyck zon jb. -Rcmcy Xcmmcn
```

```
Jxwpjq qwrla glcu pcx qcn xkvv dw uclw ekarbbckpjwe dq jzw jzkpta
jzrj qcn ekep'j ec jzrp dq jzw cpwa qcn eke ec. -Urls Jxrkp

Xjzwq gjz cuvq xz huri arwqvudiy fuk ufjrqoq svquxiy. -Lzjk
Nqkkqcy

Axevfvu lvnelvp bxqp mvpprjv rgl bvoop Grnxvgkvuj dqupb jvbp
buvrbvl be lqggvu.
```

Requirements and Other Thoughts

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

1. You should backup your code to an online repository like dropbox or google drive frequently (e.g., creating a new function successfully). If you come to us and complain that your computer crashed and you lost all of your work, we'll ask you where your backups are.
2. 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
3. No matter what you do, and how much you finish, make sure your project builds and at least runs (even if it crashes after a while). WHATEVER YOU DO, don't turn in code that doesn't build.
4. Whatever you do, DO NOT MODIFY OUR PROVIDED HEADER FILE IN ANY WAY! YOU WILL NOT TURN IT IN, SO ANY MODIFICATIONS WILL NOT BE SEEN BY OUR GRADING TOOL!
5. The entire project can be completed in less than 550 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.
6. Be sure to make use of the C++ STL where we permit it – it can make things much easier for you!
7. If you need to define your own inline comparison operators, feel free to do so! However you MUST place these functions within one of your own source files that you will submit as part of the project. You MUST NOT modify our provided header file!!
8. Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. Plan before you program!
9. Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
10. You MUST NOT modify any of the code in the files we provide you that you will not turn in; since you're not turning them in, we will not see those changes. We will incorporate the required files that you turn in into a project with special test versions of the other files.
11. 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.

12. Try your best to meet our big-O requirements for each method in this spec. If you can't figure out how, then solve the problem in a simpler, less efficient way, and move on. Then come back and improve the efficiency of your implementation later if you have time.

If you don't think you'll be able to finish this project, then take some shortcuts. For example, implement the *MyHash* class first by using the STL's map or unordered_map class to do all of the hard work. Then use it to get your decrypter working. Once you get your decrypter working, then go back and implement your full *MyHash* class.

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., *MyHash*), 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 you couldn't get working, and we can give you credit for those parts of the project you completed correctly.

But whatever you do, make sure that ALL CODE THAT YOU TURN IN BUILDS without errors with both g32 and either Visual C++ or clang++!

What to Turn In

You will turn in **six** files:

MyHash.h	Contains your resizable hash table map class template implementation
Tokenizer.cpp	Contains your tokenizer implementation
WordList.cpp	Contains your word list implementation
Translator.cpp	Contains your translator implementation
Decrypter.cpp	Contains your decrypterer implementation
report.docx, report.doc, or report.txt	Contains your report

You **must** submit a report that describes:

1. Whether any of your classes have known bugs or other problems that we should know about. For example, if you didn't finish the *DecoderImpl::crack()* method or it has bugs, tell us.
2. A high-level description of what data structures and algorithms you chose for each of your classes' non-trivial methods and data structures. Brief means a paragraph or two description or half a page pseudocode per class.
3. Whether or not each method satisfies our big-O requirements, and if not, what you did instead and what the big-O is for your version.

Grading

- 95% of your grade will be assigned based on the correctness of your solution.
- 5% of your grade will be based on your report.

Good luck!