Before you ask questions about this specification, see if your question has already been addressed by the [Project 5 FAQ](http://cs.ucla.edu/classes/winter14/cs31/Projects/5/faq.html). And read the FAQ before you turn in this project, to be sure you didn't misinterpret anything.

Many encryption systems have been developed over the years to keep messages secret. One of the oldest is the *simple substitution cipher*. In this scheme, each letter in the original plaintext message is consistently replaced by a letter to produce the ciphertext message (e.g., every E is replaced by K). To be reversible, different letters are not replaced by the same letter (e.g., if every E is replaced by K, no other letter will also be replaced by K). It is allowable for a letter to be replaced by itself (e.g., every H is replaced by H). If the sender and receiver have agreed on the substitution scheme (the *key*), the receiver can easily decrypt the encrypted message. As an example, suppose the key is this:

ABCDEFGHIJKLMNOPQRSTUVWXYZ *plaintext letters*

NRWZKXCHFBOIMTGVJLYADEPQSU *corresponding ciphertext letters*

Then the plaintext message ATTACK AT DAWN would be encrypted as NAANWO NA ZNPT.

Simple substitution ciphers are very insecure; their cryptanalysis (recovering the plaintext message from the ciphertext message without knowing the key) is not difficult. It's even easier if the cryptanalyst can use a *known plaintext attack*, one in which there is a word or phrase that is known (or strongly suspected) to occur in the message that was encrypted. This known word or phrase is a *crib*.

For this project, you will write a function that will take a set of ciphertext messages, all encrypted with the same key, and a crib that occurs in one of the messages. The output will be the set of ciphertext messages, with plaintext letters substituted for ciphertext letters to the extent that they can be determined from the crib. As an example, suppose there are three ciphertext messages:

Tftdn lxwdqkoft lhgzztr of Ltezgk zvtfzn-ltctf.

O ziofa zit Zktqlxkt gy Doeiossofrq ol wxkotr lgdtvitkt of dn wqeanqkr.

Ror ngx itqk ziqz Sgktzzq ol ugofu gxz vozi Dqkexl?

and the crib is treasure of michillinda. The only ciphertext fragment that could possibly be an encryption of the crib (because it's the only phrase that has words of the right length with the right pattern of repeated letters) is zktqlxkt gy doeiossofrq. That implies that ciphertext z corresponds to plaintext t, k decrypts to r, etc. Your program would output

enemN suWmarine sHotted in sector tVentN-seCen.

i thinA the treasure of michillinda is Wuried someVhere in mN WacANard.

did Nou hear that loretta is UoinU out Vith marcus?

The case of letters in ciphertext and the crib is irrelevant. (Notice, for example, that Zktqlxkt matched treasure; the fact that Z is upper case and t is lower case is irrelevant.) However, when you output the (partially) decrypted messages, all plaintext letters must be written in lower case, while remaining ciphertext letters that could not be determined from the crib must be written in upper case.

The function you implement to do this must have the following prototype. (To learn about istream and how to read from a file, read the [tutorial on file I/O](http://cs.ucla.edu/classes/winter14/cs31/Projects/5/fileio.html).)

bool decrypt(istream& cipherstream, const char crib[]);

The parameter cipherstream is an already-opened input source you will read from (probably a file the caller opened). That input source contains the ciphertext messages, one message per line. You may assume (and thus don't have to check) that the input source will contain no more than 50 lines, and that no line of the input source will be longer than 80 characters (not counting the newline at the end of the line).

The parameter crib is a C string that denotes the crib, the sequence of one or more words that appear consecutively in order in at least one of the ciphertext messages. One or more blanks separate words in the crib; non-letter characters in crib are to be treated as if they were blanks. Thus, the crib "D-Day is June 6, 1944" should be treated the same as "d day is june", as indicating the sequence consisting of those four words. There is no limit to the possible length of the crib string.

If the crib string has no words, or if no ciphertext fragment in any message could possibly be an encryption of the crib, the decrypt function returns false without writing anything tocout. Otherwise, it writes to cout the (partially) decrypted messages as described above and returns true. The decrypt function must not cause any other output to be written to cout. If more than one ciphertext fragment is a possible encryption of the crib, then choose any one of those matching fragments as the match for the crib. For example, if the set of encrypted messages consisted only of the message Rzy pkr and the crib were "dog", then the output would be exactly one of dog PKd or gZY dog, your choice.

A crib word must match an entire ciphertext word. The crib word "aba" matches "cdc" in "cdc ef", but not in "cdcef" or "efcdc". A match for the crib does not span multiple messages. For example, if there were two consecutive lines in the input file, "bwra wmwt" and "qeirtk spst", and the crib were "alan turing", the "wmwt" from the first message and the "qeirtk" from the second are not considered a match for the crib.

A word is a sequence of letters only, so the crib "dog" would not match anything in the ciphertext "ew'q p-aj", but the crib "he" could match either ew or aj in that ciphertext. As another example, the crib "s cloak and" matches something in the ciphertext "Kpio't dmpbl-boe-ebhhfs opwfm"; the partially decrypted plaintext would be written as "KoIn's cloak-and-daHHFS noWFl".

All the preceding rules imply that all of these crib strings should be treated the same way:

"D-Day is June 6, 1944"

" d???dAy--- ---iS JunE !! "

"d day is june"

and would match something in the ciphertext string

"DhaiiA, zyxZYXzyx--A aBc dE## $$fGhi6437 wvuWVUwvu"

causing the partially decrypted plaintext of that string to be written as

"indeed, ZYXZYXZYZ--d day is## $$june6437 WVUWVUWVU"

Your decrypt function and any functions you write that it calls must **not** use any std::string objects. If you need to use a string, you must use a C string. (Although the program you turn in must not use any C++ strings, only C strings, you might want to consider this development strategy: Ignore this restriction at first, and develop a working solution that uses C++ strings. After you've ironed out the issues in writing the decryptor, save a backup, and then convert your using C++ strings to using C strings instead. This approach helps you avoid confusing the mistakes in your use of C strings with the mistakes in your decryption algorithm, so makes debugging easier.)

(Note: Some algorithms that you might consider for your decrypt function may appear at first to require that you assume a limit on the length of the crib string. We prohibited that. But we gave you permission to assume that the maximum length of any ciphertext message is 80, so you know that a crib string that could possibly match only messages longer than 80 characters could not possibly match any of the ciphertext messages; for crib strings like that, you can return false without any further analysis. Thus, if you think about it a little, you can determine maximum limits for any auxiliary arrays and C strings you might want your decrypt function to create.)

The decrypt function is the only function you are required to write. You may write additional functions as part of your solution if you wish. While we won't test those additional functions separately, their use may help you structure your program more readably. Of course, to test all your functions, you'll want to write a main routine that calls your decryptfunction. During the course of developing your solution, you might change that main routine many times. As long as your main routine compiles correctly when you turn in your solution, it doesn't matter what it does, since we will rename it to something harmless and never call it (because we will supply our own main routine to throroughly test yourdecrypt function).

Your decrypt function and any functions it calls must not cause anything to be read from cin. They must not cause anything to be written to cout other than the (partially) decrypted messages required by this spec. If you want these functions to write things out for debugging purposes, write to cerr instead of cout. When we test your program, we will cause everything written to cerr to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

The correctness of your program must not depend on undefined program behavior. Your program could not, for example, assume anything about t's value, or even whether or not the program crashes:

int main()

{

char t[6];

strcpy(t, "Enigma"); // too long: 7 chars including '\0'

…

Here's an example of a main routine that performs some simple tests of the decrypt function:

void runtest(const char filename[], const char crib[])

{

cout << "====== " << crib << endl;

ifstream cfile(filename);

if (!cfile)

{

cout << "Cannot open " << filename << endl;

return;

}

bool result = decrypt(cfile, crib);

cout << "Return value: " << result << endl;

}

int main()

{

cout.setf(ios::boolalpha); // output bools as "true"/"false"

runtest("mytest.txt", "my secret");

runtest("mytest.txt", "shadow");

}

If the file mytest.txt contained

Hirdd ejsy zu drvtry od.

O'z fodvtrry.

then the output of running the program with this main routine would be

====== my secret

HIess EJSt my secret Os.

O'm FOscreet.

Return value: true

====== shadow

Return value: false

What you will turn in for this assignment is a zip file containing these two files and nothing more:

1. A text file named **decrypt.cpp** that contains the source code for your C++ program. Your source code should have helpful comments that tell the purpose of your data structures and program segments, and explain any tricky code.
2. A file named **report.doc** or **report.docx** (in Microsoft Word format), or **report.txt** (an ordinary text file) that contains:
   1. A brief description of notable obstacles you overcame.
   2. A description of the design of your program. You should use [pseudocode](http://www.cs.ucla.edu/classes/winter14/cs31/pseudocode.html) in this description where it clarifies the presentation.
   3. A list of the kinds of test data that could be used to thoroughly test the function, along with the actual data itself. You must note which test cases your program does not handle correctly. (This could happen if you didn't have time to write a complete solution, or if you ran out of time while still debugging a supposedly complete solution.)

By ~~February 28~~ March 2, there will be links on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above. Give yourself enough time to be sure you can turn something in, because we will not accept excuses like "My network connection at home was down, and I didn't have a way to copy my files and bring them to a SEASnet machine." There's a lot to be said for turning in a preliminary version of your program and report early (You can always overwrite it with a later submission). That way you have something submitted in case there's a problem later. Notice that most of the test data portion of your report can be written from the requirements in this specification, before you even start designing your program.

// Project 5 solution

#include <iostream>

#include <cstring>

#include <cctype>

#include <fstream>

#include <sstream>

#include <string>

#include <cstdlib>

#include <cassert>

using namespace std;

const int OKTEXT = 3;

bool decrypt(istream& cipherstream, const char crib[]);

int dotest(const char\* ciphertext, const char\* crib, const char\* plaintext)

{

istringstream cipherstream(ciphertext, ios::in | ios::out | ios::app);

{

ostream xcs(cipherstream.rdbuf());

xcs << endl;

}

ostringstream oss;

streambuf \*sb = cout.rdbuf(oss.rdbuf());

bool result = decrypt(cipherstream, crib);

cout.rdbuf(sb);

if (plaintext == NULL)

return result ? 0 : 1; // 1 point if returns false when should

int retval = 0;

if (result)

retval++; // 1 point if returns true when should

string s = oss.str();

if (!s.empty() && s[s.length()-1] == '\n')

s.erase(s.length()-1);

if (s == plaintext)

{

retval += OKTEXT; // plus OKTEXT points for correct output text

}

else

{

if (!s.empty() && s[s.length()-1] == '\n')

s.erase(s.length()-1);

if (s == plaintext)

retval += OKTEXT; // plus OKTEXT points for correct output text

}

return retval;

}

void testone(int n)

{

int res;

switch (n)

{

default: {

cout << "Bad argument" << endl;

} break; case 1: {

res = dotest("", "", NULL);

} break; case 2: {

res = dotest("", "a", NULL);

} break; case 3: {

res = dotest("a", "", NULL);

} break; case 4: {

res = dotest("ab", "b", NULL);

} break; case 5: {

res = dotest("abc", "de f", NULL);

} break; case 6: {

res = dotest("abc", "de-f", NULL);

} break; case 7: {

res = dotest("abc", "ded", NULL);

} break; case 8: {

res = dotest("aba", "def", NULL);

} break; case 9: {

res = dotest("agbhc\nabcd", "def", NULL);

} break; case 10: {

res = dotest("ab cd", "ef ge", NULL);

} break; case 11: {

res = dotest("ab ca", "ef gh", NULL);

} break; case 12: {

res = dotest("ab\ncd", "ef gh", NULL);

} break; case 13: {

res = dotest("a", "b", "b");

} break; case 14: {

res = dotest("ab", "ba", "ba");

} break; case 15: {

res = dotest("aba", "ded", "ded");

} break; case 16: {

res = dotest("ab a", "c", "cB c");

} break; case 17: {

res = dotest("abc ab bca", "de",

"deC de eCd");

} break; case 18: {

res = dotest("abc ab bc a", "de",

"deC de eC d");

if (res < OKTEXT)

{

int res2 = dotest("abc ab bc a", "de",

"Ade Ad de A");

if (res2 > res)

res = res2;

}

} break; case 19: {

res = dotest("abccd abccc abccb abcca", "xyzzy",

"xyzzD xyzzz xyzzy xyzzx");

} break; case 20: {

res = dotest("abc abc", "def",

"def def");

} break; case 21: {

res = dotest("abc", "dEf", "def");

} break; case 22: {

res = dotest("abC", "dEf", "def");

} break; case 23: {

res = dotest("aba", "dED", "ded");

} break; case 24: {

res = dotest("abA", "ded", "ded");

} break; case 25: {

res = dotest("abc", "def ", "def");

} break; case 26: {

res = dotest("abc", "def!32", "def");

} break; case 27: {

res = dotest("abc", "32!def", "def");

} break; case 28: {

res = dotest("abc!", "def", "def!");

} break; case 29: {

res = dotest("@@abc!", "def", "@@def!");

} break; case 30: {

res = dotest("abc\nagbhc", "def",

"def\ndGeHf");

} break; case 31: {

res = dotest("agbhc\nabc", "def",

"dGeHf\ndef");

} break; case 32: {

res = dotest("abc de fab cde", "gh ijk",

"jkC gh ijk Cgh");

} break; case 33: {

res = dotest("abc de fabc ef ab cde fg", "gh ijk",

"ghi jk Fghi kF gh ijk FG");

} break; case 34: {

res = dotest("abc de fab de abd de abc", "gh ijg",

"ijC gh Fij gh ijg gh ijC");

} break; case 35: {

res = dotest("ab cd", "ef#$% gh", "ef gh");

} break; case 36: {

res = dotest("ab#$% cd", "ef gh", "ef#$% gh");

} break; case 37: {

res = dotest("ab\nbc", "de",

"de\neC");

if (res < OKTEXT)

{

int res2 = dotest("ab\nbc", "de",

"Ad\nde");

if (res2 > res)

res = res2;

}

} break; case 38: {

string cs;

for (int k = 0; k < 40; k++)

cs += " a";

cs += '\n';

string ct("ab");

for (int k = 0; k < 39; k++)

ct += " a";

ct += '\n';

string ps;

for (int k = 0; k < 40; k++)

ps += " c";

ps += '\n';

string pt("cd");

for (int k = 0; k < 39; k++)

pt += " c";

pt += '\n';

for (int k = 0; k < 49; k++)

{

ct += cs;

pt += ps;

}

ct.erase(ct.size()-1);

pt.erase(pt.size()-1);

res = dotest(ct.c\_str(), "cd", pt.c\_str());

} break; case 39: {

string crib;

crib.reserve(1000003);

crib.append(500000,':');

crib += "def";

crib.append(500000,':');

res = dotest("abc", crib.c\_str(), "def");

} break; case 40: {

string crib;

crib.reserve(1000082);

for (int k = 0; k < 40; k++)

crib += "b ";

crib.append(1000000, ' ');

crib += "b ";

string ct;

for (int k = 0; k < 40; k++)

ct += "a ";

res = dotest(ct.c\_str(), crib.c\_str(), NULL);

res \*= (2+OKTEXT);

} break;

}

cout << res << endl;

}

int main()

{

cout << "Enter a test number (1 to 40): ";

int n;

cin >> n;

if (n < 1 || n > 40)

{

cout << "Bad test number" << endl;

return 1;

}

testone(n);

}

const int MAX\_MESSAGES = 50;

const int MAX\_MESSAGE\_LENGTH = 80;

const int MAX\_NORMALIZED\_LENGTH = MAX\_MESSAGE\_LENGTH + 1;

const int ALPHABET\_SIZE = 'z' - 'a' + 1;

const char UNKNOWN = '?'; // any non-letter will do

bool normalizeText(const char src[], char dest[], int destSize);

bool matchCrib(const char message[], const char crib[],

char plaintextLetters[]);

bool matchCribStartingHere(const char message[], int msgpos, const char crib[],

char plaintextLetters[]);

void writeDecryption(const char msgs[][MAX\_MESSAGE\_LENGTH+1], int nMsgs,

const char plaintextLetters[]);

////////////////////

// decrypt

// If originalCrib matches any message in cipherstream, write the (partially)

// decrypted message and return true; otherwise, write nothing and return false.

bool decrypt(istream& cipherstream, const char originalCrib[])

{

// Normalize crib. If resulting crib would be too long to match any

// message, or has no words, return false. (Length of array accounts

// for one space after each word, including the last, and the '\0' byte.)

char crib[MAX\_NORMALIZED\_LENGTH+1];

if (! normalizeText(originalCrib, crib, MAX\_NORMALIZED\_LENGTH+1))

return false;

if (crib[0] == '\0')

return false;

// Save messages for later processing, since we have to make two passes

// through the messages.

char messages[MAX\_MESSAGES][MAX\_MESSAGE\_LENGTH+1];

int nMessages = 0;

char line[MAX\_MESSAGE\_LENGTH+1];

while (cipherstream.getline(line, MAX\_MESSAGE\_LENGTH+1))

{

if (nMessages == MAX\_MESSAGES)

{

cerr << "The number of messages exceeds the spec's guarantee!"

<< endl << "Processing only the first " << MAX\_MESSAGES

<< " messages." << endl;

break;

}

strcpy(messages[nMessages], line);

nMessages++;

}

// Try to match crib against each message.

for (int m = 0; m < nMessages; m++)

{

// Normalize message.

char normalizedMsg[MAX\_NORMALIZED\_LENGTH+1];

if (! normalizeText(messages[m], normalizedMsg, MAX\_NORMALIZED\_LENGTH+1))

{

cerr << "BUG: Normalized message is too long!" << endl;

return false;

}

// For a matching crib, this array will eventually contain the

// decryptions of ciphertext letters. For example, if ciphertext

// 'e' maps to plaintext 't', then plaintextLetters['e'-'a'] == 't'.

// If a mapping is not determined, the element value is UNKNOWN.

char plaintextLetters[ALPHABET\_SIZE];

// If crib matches somewhere in normalizedMsg, write (partial)

// decryption of all messages and return

if (matchCrib(normalizedMsg, crib, plaintextLetters))

{

for (int k = 0; k < ALPHABET\_SIZE; k++)

plaintextLetters[k] = tolower(plaintextLetters[k]);

writeDecryption(messages, nMessages, plaintextLetters);

return true;

}

}

// No message matched.

return false;

}

////////////////////

// normalizeText

// Transform src string into dest string. If src has no words, dest will be

// the empty string. Otherwise dest will contain all the words in src (all

// lower cased), with one space character following each word (including the

// last). Return true if this fits in dest (as determined by destsize, which

// includes accounting for the trailing '\0'). Otherwise, return false (and

// dest may be changed).

//

// As an example, " aBc--dE ?? fg#@hi" normalizes to "abc de fg hi "

bool normalizeText(const char src[], char dest[], int destSize)

{

// Fill the dest string

int d = 0;

for (int s = 0; src[s] != '\0'; s++)

{

if (isalpha(src[s]))

{

// Transform each letter

if (d >= destSize)

return false;

dest[d] = tolower(src[s]);

d++;

// Only the first non-letter after a word is mapped to a space

if (!isalpha(src[s+1]))

{

if (d >= destSize)

return false;

dest[d] = ' ';

d++;

}

}

}

if (d >= destSize)

return false;

dest[d] = '\0';

return true;

}

////////////////////

// matchCrib

// If crib matches somewhere in the message, set plaintextLetters to

// map ciphertext letters to plaintext letters and return true. Otherwise,

// return false (and plaintextLetters might be changed to something useless).

// The message and the crib must be normalized.

bool matchCrib(const char message[], const char crib[], char plaintextLetters[])

{

int k = 0;

while (message[k] != '\0')

{

// Try matching starting at the word that starts at position k

if (matchCribStartingHere(message, k, crib, plaintextLetters))

return true;

// Skip to end of word

while (isalpha(message[k]))

k++;

// Skip the one space at the end of the word

k++;

}

// No match anywhere

return false;

}

////////////////////

// matchCribStartingHere

// If crib matches the message starting at the word that starts at msgpos,

// set plaintextLetters to map ciphertext letters to plaintext letters and

// return true. Otherwise, return false (and plaintextLetters might be

// changed to something useless). The message and the crib must be normalized.

bool matchCribStartingHere(const char message[], int msgpos,

const char crib [], char plaintextLetters[])

{

// This will record whether we encounter in the crib a repeat of a

// plaintext letter that appears earlier in the crib

bool cribHasRepeatOfPrior[ALPHABET\_SIZE];

// Initialize: All mapping unknown, no repeat letters encountered yet

for (int m = 0; m < ALPHABET\_SIZE; m++)

{

plaintextLetters[m] = UNKNOWN;

cribHasRepeatOfPrior[m] = false;

}

// Walk through corresponding positions of crib and message. Since

// both are normalized, a matching crib will have spaces in positions

// that correspond to spaces in the message.

for (int c = 0; crib[c] != '\0'; c++)

{

char cribch = crib[c]; // crib character

char msgch = message[msgpos+c]; // corresponding message character

// If either is a space, the other must be, otherwise no match

if (cribch == ' ')

{

if (msgch != ' ')

return false;

}

else if (msgch == ' ')

return false;

// If neither is a space and we haven't matched this ciphertext

// letter before

else if (plaintextLetters[msgch-'a'] == UNKNOWN)

{

// No other ciphertext letter can already map to this plaintext

// letter.

if (cribHasRepeatOfPrior[cribch-'a'])

return false;

// Record mapping

plaintextLetters[msgch-'a'] = cribch;

cribHasRepeatOfPrior[cribch-'a'] = true;

}

// If we have matched this ciphertext letter before, it must map

// to this plaintext letter

else if (plaintextLetters[msgch-'a'] != cribch)

return false;

}

// We got through the crib without a mismatch

return true;

}

////////////////////

// writeDecryption

// Write the ciphertext, mapping known ciphertext letters to their upper case

// case plaintext equivalent and unknown ciphertext letters to lower case,

// and leaving all other characters unchanged.

void writeDecryption(const char msgs[][MAX\_MESSAGE\_LENGTH+1], int nMsgs,

const char plaintextLetters[])

{

// Write appropriate version of each character.

for (int m = 0; m < nMsgs; m++)

{

for (int k = 0; msgs[m][k] != '\0'; k++)

{

char ch = msgs[m][k];

if (isalpha(ch))

{

ch = tolower(ch);

if (plaintextLetters[ch-'a'] != UNKNOWN)

{

ch = plaintextLetters[ch-'a'];

}

else

{

ch = toupper(ch);

}

}

cout << ch;

}

cout << endl;

}

}