Fall 2021 CS 31

Programming Assignment 5
Secret Plans

Time due: 11:00 PM Monday, November 15

Part 1

Go through the following sections of the class zyBook, doing the Participation Activities and Challenge Activities. We will be looking at whether you have ever successfully completed them; it does not matter how many attempts you make before a successful completion (or how many attempts you make after a successful completion if you want to experiment).

- 6.6 through 6.9
- 7.2 and 7.3 (Part 2 does not depend on these)
- 8.1 and 8.2 (Part 2 does not depend on these)

Part 2

Before you ask questions about this specification, see if your question has already been addressed by the Project 5 FAQ. And read the FAQ before you turn in this project, to be sure you didn't misinterpret anything.

In the course of their investigation into a series of bank robberies, the FBI discovered encrypted messages sent between some of the suspected perpetrators. While experts are working on decrypting most of the messages, some of the messages appear to have been sent between people who know almost nothing about cryptography, so the Bureau has decided to save money by hiring a student intern to work with those messages. That intern is you.

The encrypted message you have been given have been encrypted using one of the oldest known encryption schemes: a *simple substitution cipher*. In this scheme, each letter in an original plaintext message is consistently replaced by a letter to produce a ciphertext message (e.g., every A is replaced by N). To be reversible, different plaintext letters are not replaced by the same ciphertext letter (e.g., if every A is replaced by N, no other letter will also be replaced by N). It is allowable for a letter to be replaced by itself (e.g., every M is replaced by M). 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 `BANK OF AMERICA` would be encrypted as `RNTO GX NMKLFWN`.

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 four ciphertext messages:

```
Rswjo qgx Tmeuo sgjsy jds vqgf vo jds vqzf xbby.
Udbyjo iqcju cg wybgj cg jds esjqiqo zqy
Xbg'j rsj jds jsrrsy jycn jds ucrsgj qrqyt.
ZU 31 cu zdqrrsgecge!
```

and the crib is `silent alarm`. 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 `ucrsgj qrqyt`. That implies that ciphertext `u` corresponds to plaintext `s`, `c` decrypts to `i`, etc. Your program would output

```
LEwTo ANx MmeSo ENTER TdE vANf vo TdE vAzf xbbR.
SdbRTo iAITS IN wRbNT IN TdE eETAiAo zAR
xbN'T LET TdE TELLER TRIn TdE SILENT ALARM.
zS 31 IS zdALLENeINe!
```

The case of letters in ciphertext and the crib is irrelevant. (For example, ciphertext `UcRSgJ` matches crib `sILent`; the fact that the ciphertext's `U` is upper case and the crib's `s` is lower case is irrelevant.) However, when you output the (partially) decrypted messages, all plaintext letters must be written in upper case, while remaining ciphertext letters that could not be determined from the crib must be written in lower case. All non-letter characters (punctuation, digits, blanks, newlines, etc.) must be written unchanged.

The function you implement to do this must have the following prototype:

```
bool decrypt(const char ciphertext[], const char crib[]);
```

The parameter `ciphertext` is a single C string with all the encrypted messages, separated by newline characters. (There may or may not be a newline character after the last message.) For example, a caller who wanted to decrypt the two messages

```
F lgr rntoy rkwndyk ahna'y
phklk ahk mgtks fy.
```

could pass as the first argument `"F lgr rntoy rkwndyk ahna'y\n phklk ahk mgtks fy."` or `"F lgr rntoy rkwndyk ahna'y\n phklk ahk mgtks fy.\n"`. You may assume (and thus don't have to check) that the ciphertext will contain no more than 70 newline characters, and that no message within the ciphertext will be longer than 90 characters (not counting a newline at the end of the message). In other words, there will never be more than 90 characters between two newlines in the ciphertext or before the first newline or after the last newline. It is possible that a message has no words (e.g., is empty, or has digits and spaces but no letters).

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. (For this spec, we define a *word* to be a sequence of one or more letters.) 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 `"hush-hush until November 25, 2021"` should be treated the same as `"hush hush until november"` would be, as indicating the sequence consisting of those four words. You must **not** assume any particular limit to the possible length of the crib string argument that is passed to the function.

only the non-letters within letters are treated as blanks; otherwise not included in the crib

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 to `cout`. 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 ciphertext string were `"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 the ciphertext string were `"bwra wmwt\nqeirtk spst\n"`, 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:

```
"hush-hush until November 25, 2021"
"   hush???hUSh---     --- until    NovemBER !!  "
"hush hush until november"
```

and would match something in the ciphertext string

```
"F gspt fe! zyxZYXzyx--Abca abCa    bdefg## $$dsptrqtj6437 wvuWVUwvu\n\n8 9\n"
```

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

```
"I LOVE IT! zyxzyxzyx--HUSH HUSH    UNTIL## $$NOVEMBER6437 wvuwvuwvu\n\n8 9\n"
```

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.

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 message within the ciphertext string is 90, so you know that a crib string that could possibly match only messages longer than 90 characters could not possibly match any of the ciphertext messages; for crib strings like that, you could 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 declare.

done

Standard C++ requires that the number of elements in an array you declare to be known at compile time. Since the g31 command on cs31.seas.ucla.edu enforces that requirement, and your program must run under that compiler, you must meet that requirement. Thus, you must not do something like this:

```
bool decrypt(const char ciphertext[], const char crib[])
{
    char a[strlen(crib)]; // Error! strlen(crib) not known at compile time
```

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 your `decrypt` function, you'll want to write a main routine that calls it. 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 your `decrypt` 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.

Your implementation must not use any global variables whose values may be changed during execution.

Your program must build successfully under both g31 and either Visual C++ or clang++.

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 ciphertext[], const char crib[])
{
    cout << "====== " << crib << endl;
    bool result = decrypt(ciphertext, crib);
    cout << "Return value: " << result << endl;
}

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

    runtest("Hirdd ejsy zu drvtry od.\nO'z fodvtrry.\n", "my secret");
    runtest("Hirdd ejsy zu drvtry od.\nO'z fodvtrry.\n", "shadow");
}
```

The output of running the program with this main routine would be the following. (Only two of the lines below are written by `decrypt`, of course.)

```
====== my secret
hiESS ejsT MY SECRET oS.
o'M foSCREET.
Return value: true
====== shadow
Return value: false
```

What to turn in

You won't turn anything in through the CS 31 web site for Part 1; the zyBook system notes your successful completion of the PAs and CAs. For Part 2, you will turn in 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.docx** or **report.doc** (in Microsoft Word format), or **report.txt** (an ordinary text file) that contains:
    a. A brief description of notable obstacles you overcame.
    b. A description of the design of your program. You should use pseudocode in this description where it clarifies the presentation. Someone reading your description should be able to determine what the responsibilities of the functions you wrote are. If someone had to modify your code later, could they from your description readily find in your program the approximate location of the code that, for example, detect whether a crib word matches a ciphertext word?
    c. A list of the test data that could be used to thoroughly test the function, along with the reason for each test case (e.g., "crib with non-letter" or "crib letter case not same as ciphertext case"). 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 November 14, 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.

Note

Although the program you turn in must use C strings and is forbidden from using C++ strings, you can experiment with ideas for doing this project without that restriction. For example, you could create an experimental project (that you will not turn in) and pretend the required function is `bool decrypt(const string ciphertext, const string crib)`. (The prototype for this experimental version declares the parameters as `const string`s so that your experimental implementation doesn't try to modify them in any way even though they are copies of the caller's arguments. This is because the real function has `const char[]` parameters, which won't allow the caller's arguments to be modified.)

You could work out a lot of what you need to do for this project using C++ strings without the distraction of having to wrestle with C strings. Use what you learn from the experimental project when writing the real project that uses only C strings. Warning: It may not be wise to try to completely finish the experimental C++ string version before even starting the real C string version; it might take you more time than you thought to figure out how to work with C strings, so you might not have anything working in the C string version that you must turn in. Instead, when you have just a few things working in a C++ string version, try implementing them and getting them to work in the C string version, so you'll know how much time it takes you to translate from using C++ strings to C strings. Get more things working in the C++ string version, then in the C string version. Once you're comfortable with C strings, you might abandon the experimental version and continue on with just the real C string version. (Or not; maybe you prefer to continue working out each new bit with C++ strings first before implementing it with C strings.)

Notes for Visual C++ users

Microsoft made a controversial decision to issue by default an error or warning when your code uses certain functions from the standard C and C++ libraries (e.g., `strcpy`). These warnings call those functions unsafe and recommend using different functions in their place; those other functions, though, are not Standard C++ functions, so will cause a compilation failure when you try to build your program under g31 or clang++. Therefore, for this class, we want to use functions like `strcpy` without getting those warnings from Visual C++; to eliminate them, put the following line in your program *before* any of your `#include`s:

```
#define _CRT_SECURE_NO_DEPRECATE
```

It is OK and harmless to leave that line in when you build your program using g31 or clang++ and when you turn it in.

If you declare a large array in a function, Visual C++ gives a harmless warning `C6262: Function uses '`*NNNNN*`' bytes of stack: exceeds /analyze:stacksize '16384'. Consider moving some data to the heap.`, where *NNNNN* is some number. You can eliminate that warning by putting the following line in your program above the `#define _CRT_SECURE_NO_DEPRECATE`:

```
#pragma warning(disable:6262)
```

It is OK and harmless to leave that line in when you build your program using g31 or clang++ and when you turn it in, even if you get a warning about the pragma being ignored.

Alternatively, in Visual Studio, select Project / *yourProjectName* properties, then select Configuration Properties / Code Analysis / General, and then in Code Analysis's stacksize, modify 16384 to, say, 100000.

If your program dies under Visual C++ with a dialog box appearing saying "Debug Assertion Failed! ... File: ...\src\isctype.c ... expression: (unsigned)(c+1)<=256", then you called one of the functions defined by `<cctype>`, such as `isalpha` or `tolower`, with a character whose encoding is outside the range of 0 through 127. Since all the normal characters you would use (space, letters, punctuation, `'\0'`, etc.) fall in that range, you're probably passing an uninitialized character to the function. Perhaps you're examining a character past the `'\0'` marking the end of a C string, or perhaps you built what you thought was a C string but forgot to end it with a `'\0'`.