

Project 1 - Ciphers

CS 251, Fall 2024

Copyright Notice

© 2024 Ethan Ordentlich, University of Illinois Chicago

This assignment description is protected by [U.S. copyright law](#). Reproduction and distribution of this work, including posting or sharing through any medium, such as to websites like [chegg.com](#) is explicitly prohibited by law and also violates [UIC's Student Disciplinary Policy](#) (A2-c. Unauthorized Collaboration; and A2-e3. Participation in Academically Dishonest Activities: Material Distribution). Material posted on [any third party](#) sites in violation of this copyright and the website terms will be removed and your user information will be released.

Projects in CS 251

Projects in CS 251 are **solo**. You may discuss high-level concepts with other students, but any viewing, sharing, copying, or dictating of code is forbidden. Additionally, you may not receive programming help from anyone other than CS 251 course staff. This includes the use of AI tools, such as ChatGPT or Copilot.

Projects will have a large design component where **you** have to decide how to set things up. We will ramp up slowly on this; so, for this project, we will still give you a lot of guidance. However, ***if you get stuck for more than 30 minutes on a bug or understanding something, you should come to help hours or ask a question on Piazza***. We want this experience to be an enjoyable one, and nobody really enjoys fruitless debugging for hours on end.

Goals

This project introduces you to some of the basic features of C++. By the end of this project, you'll:

- Use strings, vectors, files, references, and streams to write a medium-length C++ program
- Implement a complicated algorithm according to an English description

Restrictions

This project is intended as a “warmup” for a (re)introduction to C++ after break. As such, we have the following restrictions:

- **Don't use any additional libraries** beyond the ones directly included in `ciphers.cpp`, which are:
 - `<cctype>` (functions on chars)
 - `<fstream>` (reading files)
 - `<iostream>` (console input, output)

- `<sstream>` (string parsing, if needed)
- `<string>`
- `<vector>`
- **Don't use libraries that are included only in `utils.h`, such as `<unordered_map>`.**
- **Don't use structs, pointers, `new`, or `malloc`.**
- **Don't add global variables.** Define “global” variables as local variables in `main`, and use pass-by-reference.
 - The one exception to this is the given `ALPHABET` constant in `include/caesar_enc.h`.

If something isn't clear, please ask! You can ask questions in instructor or TA help hours or in lab.

Logistics

- Due: 11:59 PM Tuesday, September 17
- Submit to Gradescope
 - `ciphers.cpp`
 - `plaintext.txt`
- Use grace tokens:
 - https://script.google.com/a/macros/uic.edu/s/AKfycbz23ZyuvPtyODju-o3eLJNVZP6DELy cE1YHwsleMV_pmEMin-ZqmAgO3V5bhjNv5eNm/exec
 - This form requires your UIC Google Account. The permissions it requests are to know who you are, and for the script to edit our spreadsheet database in the back end.
 - The form will become active **once the project autograder closes**.
 - See [\[FA24\] CS 251 Course Info](#) for details.

The beginning of this project will give you a lot of explicit guidance. The end of the project will give less guidance, and more closely resemble later CS 251 projects.

Setup

If you have not set up your computer to work on CS 251 projects, you should do so now:

[\[FA24\] CS 251 Software Setup](#)

Starter Code


[proj1-ciphers-starter.zip](#)

Use the **make** commands to run and test your project, not the run button or typing the **clang++** command manually. We'll tell you the relevant **make** commands at various places in the project guide when you're ready to use them.

Since this is the first project, we'll explain the purpose of each file or folder in the starter code.

- **.vscode/**: contains settings so that VSCode knows how our project is set up
 - This is a “hidden folder”. You may not be able to see it in your file explorer. Google for how to make hidden files visible.
- **ciphers.cpp**: write all your code in here, and submit this file at the end of the project!
- **include/**: contains function descriptions and documentation for everything you'll need to implement. You should read these files, but shouldn't modify them.
- **tests/**: contains our testing code. You can read these files to learn more about how we check that your implementation is correct.
- **utils.h**: contains code to do some of the tasks that are out-of-scope for this project. You won't need to read this file.
- **dictionary.txt**, **english_quadgrams.txt**, **cryptogram.txt**: data files for the program
- **Makefile**: contains “shortcuts” for compiling and running the program.
 - This lets us type **make COMMAND** in the terminal, for several different **COMMANDs**. We'll tell you which commands are relevant at various points.
- **build/**: a folder we use to store files that are created by the compiler. You won't need to look in here, but it needs to exist.

Sample Execution

 [FA24] Ciphers Sample Execution

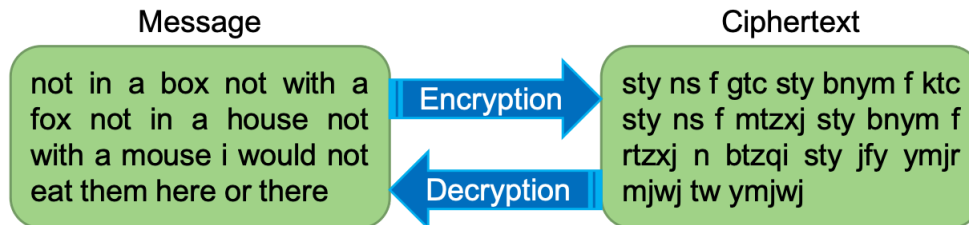
FAQ

 [FA24] Ciphers FAQ

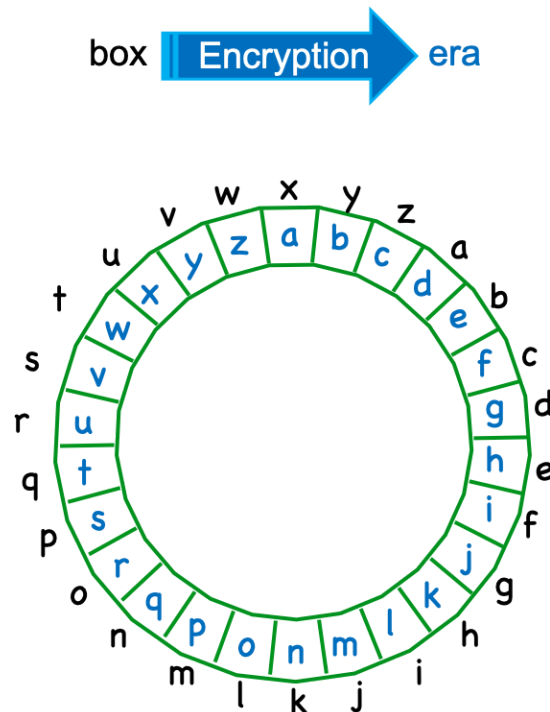
Tasks

Task: Caesar Cipher Encoder

A **cipher** is an algorithm for encrypting or decrypting text. We call the unencrypted text “plaintext” and the encrypted version “ciphertext”. Here is an example of a **Caesar cipher**, which you will implement in this project.



Notably, the original text is readable, but the encrypted version is not. A Caesar cipher is computed by “shifting” each letter of the text forward in the alphabet by a constant number, as if it was in a circle. For example, the following “ring” shows the mapping for a Caesar cipher where each letter is shifted forward by 3.



Ultimately, the goal for this part is to take in a piece of text (the “plaintext”) and a number of letters to rotate by (the “key”), and output the encrypted text (the “ciphertext”).

Subtask: Caesar Utility Functions

Like most programs, Caesar ciphers can be broken down into smaller operations. We'll start small with C++ by first implementing these operations as individual functions.

In some programming languages, such as Python or C, you can't have two functions named the same thing, even if they take different arguments. In languages like C++, this is allowed and called **function overloading**. First, you will implement two versions of the `rot` function (short for rotate): one for if we want to rotate a character and another for rotating a whole string.

You can find documentation and examples for these functions in the file `include/caesar_enc.h`. Put your implementation of these functions in `ciphers.cpp`.

`char rot(char c, int amount)`

We recommend converting the input character into a number using `string::find`. Then, add the rotation amount to that number, making sure to wrap the number around so it is between 0 and 25. Then, convert it back to a character. Assume that the input character is an uppercase letter.

`string rot(const string& line, int amount)`

You should convert each letter to uppercase, then rotate it by the rotation amount and return a new string with all the letters rotated. Non-alphabetic characters, except for spaces, are removed.

You'll find some functions from the `<cctype>` library to be helpful here, such as `isalpha`, `toupper`, and `isspace`.

Subtask: Running the Tests

The next step is testing the code you've written! There's several ways to do this, one of which is writing a separate `main` that calls these functions on various inputs. You're welcome to do this, but we've provided tools that do this automatically.

Since we're working in C++, we have to make sure that our program is up to date before we can run it. The compilation command is a bit annoying, so we use a `Makefile` to help manage this process. You're welcome to look in the `Makefile` itself, but you don't need to understand what's in there for this class.

Use the `make` commands to run and test your project, not the run button or typing the `clang++` command manually. We'll tell you the relevant `make` commands at various places in the project guide when you're ready to use them.

In a terminal, navigate to your Project 1 directory that contains the file `ciphers.cpp` and the `Makefile`. Then, run the command `make test_caesar_enc`. This command makes sure that our test program is up to date with whatever changes we've made, then executes only the tests for the Caesar Cipher Encoder portion.

If you've run the tests and written the code correctly, you should see something like this:

```
[=====] Running 7 tests from 3 test suites.
[-----] Global test environment set-up.
[-----] 1 test from CaesarEnc_RotChar
[ RUN      ] CaesarEnc_RotChar.Chars
[          OK ] CaesarEnc_RotChar.Chars (0 ms)
[-----] 1 test from CaesarEnc_RotChar (0 ms total)

[-----] 3 tests from CaesarEnc_RotString
[ RUN      ] CaesarEnc_RotString.AllUpperAlpha
[          OK ] CaesarEnc_RotString.AllUpperAlpha (0 ms)
...

```

There will be some failing tests from `CaesarEnc_FullCommand`. That's ok – we haven't implemented that part, so we expect it to fail.

Run the Caesar Encoder tests as described above, with `make test_caesar_enc`. If you've correctly implemented everything so far, the tests in `CaesarEnc_RotChar` and `CaesarEnc_RotString` should all pass with a green OK! If a test is failing, take a look at the test name and console output to help debug!

The test output can get pretty long, so remember that you can scroll up to see earlier output, which includes the expected and actual values for the test. The source code for all our tests is in the `tests` folder, and you can read them to see examples.

Don't move on until all the tests in `CaesarEnc_RotChar` and `CaesarEnc_RotString` pass.

Subtask: Encrypt with Caesar Shift

Since the "Caesar shift" logic and the "command loop" logic are separate and not really related, we can continue to apply *function decomposition*. The main idea behind function decomposition is breaking functions down to accomplish smaller independent tasks (such as rotating), then

combining them to accomplish larger and larger tasks (such as Caesar shifting or the whole command).

In `main`, you'll add a case for the `c` or `C` command, which should immediately call our helper function. For this command, you should prompt for input twice in that helper function:

- "Enter the text to Caesar encrypt: "
- "Enter the number of characters to rotate by: "

Then, perform the actual Caesar shift and output the encrypted text.

Implement the "Encrypt with Caesar Cipher" helper function, `runCaesarEncrypt()`, as described above. Also modify `main` in `ciphers.cpp` to call this helper function when prompted.

Then, run the Caesar Encoder tests again and debug until they all pass. You can also use `make run_ciphers` to interact with your program and manually check the `c` command.

When reading from `cin`, you should always use `getline`, since it reduces the number of possible bugs when dealing with input streams. Since `getline` reads input as strings, you'll find `stoi` useful. We aren't going to worry about error-handling in this project, so you don't have to handle the case where input that should be a number isn't.

Later in the project, you'll likely find some of the smaller functions to be useful again, and you'll find at least one other place where writing a new helper function will be useful!

In this part, you might have noticed that a lot of information was duplicated between this guide and comments in the code. Rather than repeating the exact same thing in the guide and the comments, **we will often split the information across the two sources**. Usually, the high-level requirements and hints will be in the guide and the details and examples will be in the code and tests itself.

We recognize that you may not be used to this, but this is the way the real world works: there is no one source of information in projects. Navigating multiple sources of information is a useful skill to practice.

Task: Caesar Cipher Decoder

In the first part, we **encrypted** a piece of text using a Caesar cipher. Now, we're going to go in the opposite direction – decrypting the ciphertext back into the plaintext, without knowing which

value for the shift amount was used. To decrypt text that was encrypted by a Caesar cipher with shift k , we shift the encrypted text *again* by $26 - k$. This gives us a total rotation of 26, which is equivalent to not rotating at all!

Since the number of possible rotations is small, we can just try all of them and see which one “works best” – this is called **brute forcing** a solution. How do we know which shifts work “better”, though? We’ll compare against a dictionary and check how many words in the decrypted text are actually words in the dictionary.

Our general strategy for implementing the “Decrypt Caesar Cipher” command, which you should write in a helper function, will be the following:

- Load the dictionary (`dictionary.txt`) **once** in `main`.
- Ask for text to decrypt ("Enter the text to Caesar decrypt: ")
- Break the text into a vector of words (separated by spaces), converting lowercase to uppercase and removing non-letters.
- For each possible decryption rotation, starting from 0:
 - If more than half ($>1/2$) of the words in the rotated text are also in the dictionary:
 - Print out the entire decrypted text on its own line, with words separated by a single space.
- If no lines were output, print "No good decryptions found"

[FA24] Ciphers FAQ

Implement the “Decrypt Caesar Cipher” command as described above.

In `include/caesar_dec.h`, we have provided a *required* “function decomposition” for this part. We strongly recommend you implement the functions in the order they appear in the file. The documentation for each function in this file contains specifications for each of the functions we want you to write. As usual, put all your code in `ciphers.cpp`.

Remember to run the tests once you’re done! The command to run this section’s tests is `make test_caesar_dec`. Since there’s significantly more tests this time, you might want to run your program manually to only focus on a specific scenario. You can use `make run_ciphers` to run your `main` and interact with your program!

The file `dictionary.txt` is a bit weird in what it does or doesn’t contain, and may not output all decryptions that you think are valid.

Task: Substitution Cipher Encoder

Now, we'll write a solver for a more complicated cipher where the mapping between plaintext letters and ciphertext letters can be more than just a shift. In particular, we'll look at "simple monoalphabetic substitution ciphers" in which the letters are shuffled to create a **substitution alphabet key** and each letter in the source text is replaced with its corresponding letter in the substitution alphabet. For example:

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

The bottom row is a rearrangement of the alphabet: every letter appears, and no letter is repeated. Unlike with Caesar Ciphers, where we can just try every possible key, there are far too many possibilities for substitution ciphers to try them all¹. Instead, we're going to use *randomness* to only try keys that are likely to be better.

Subtask: Applying a Substitution Cipher

We'll represent a substitution cipher as a single `vector<char>` of uppercase letters, where the char at index 0 is what 'A' turns into; the char at index 1 is what 'B' turns into, etc. This is the second row of the table above. You can get a random substitution cipher by calling `genRandomSubstCipher`, which is included from `utils.h`.

Implement the functions from `include/subst_enc.h`, and invoke the "Apply Random Substitution Cipher" command in `main`.

The command takes in one line of text ("Enter the text to substitution-cipher encrypt: "), encrypts it using a random substitution cipher, and outputs the ciphertext. Convert all letters to uppercase and leave non-alphabetic characters alone.

Run `make test_subst_enc` to check your work.

¹ There's $26! \sim 4.03 \cdot 10^{26}$ possible ciphers. If we checked 1 billion (10^9) per second, this would still take about as long as the universe has existed. Computers are fast, but not that fast!

Task: Substitution Cipher Decoder

Just like we can decrypt a Caesar cipher by applying the correct shift, we can decrypt a substitution cipher by applying the correct substitution. That is, the same function `applySubstCipher` can also decrypt them!

Subtask: Scoring

To decide whether a candidate decryption is better, we need a way of comparing the “english-ness” of two phrases. We call this idea **scoring**. To get a score for a phrase, we will split it into rolling chunks of four letters and evaluate the probability that a chunk came from an English word. For example, if we were working with the text `SCHOOL`, we would take the chunks `SCHO`, `CHOO`, `HOO`²; and get the probability of each of these. Each of these four letter pieces is called a **quadgram**. The higher the score of the entire text, the more “english-like” it is.

Dealing with very small numbers is difficult; so, we will deal with “log probabilities” (also called “log likelihoods”) instead². One thing that appears a lot in programming is using code that someone else has written, without necessarily understanding how it works. Working with log probabilities is difficult, so we’ve provided a `QuadgramScorer` class. You do not need to understand how this class works, only how to use it³. You construct one with the following code:

```
QuadgramScorer scorer(quadgrams, counts);
```

`quadgrams` is a `vector<string>` containing quadgrams, and `counts` is a `vector<int>` containing the number of occurrences of the corresponding quadgram in some representative English text. These vectors should be in the same order; that is, `counts[0]` is the number of occurrences of `quadgrams[0]`. The result is a variable `scorer` of type `QuadgramScorer`.

We’ve given you a file (`english_quadgrams.txt`) that contains the number of occurrences of quadgrams in some representative English text in a comma-separated value (CSV) format. The first line in this file is “TION,13168375”. This means that the quadgram “TION” occurs 13168375 in the text we processed, while a later line “OUBO,5480” means that the quadgram “OUBO” occurs only 5480 times.

To use `scorer`, call `scorer.getScore(quadgram)`. This returns a `double` holding the score for that quadgram. The total score of a piece of text is the sum of the score of all its quadgrams, so the score of `SCHOOL` is the sum of the scores of `SCHO`, `CHOO`, `HOO`:

Text	Score
<code>SCHO</code>	-3.5456

² CS 261 talks about “floating point representation” in more detail, and why this is hard.

³ If you’re curious, hop into `utils.h`. We’ll cover most of these later in the term.

CHOO	-3.5687
HOOO	-3.6050
SCHOOL	-10.7195

Compare `SCHOOL` (an English word) with something like `ZNCHHQ`, which looks nothing like an English word and scores lower at -22.4587.

Parse `english_quadgrams.txt` and construct a `QuadgramScorer` outside the command loop in `main`, and implement the “Compute English-ness Score” command (with the prompt “Enter a string to score: ”). Before you score the text, remove all non-letter characters (including spaces) and convert all lowercase letters to uppercase.

Subtask: Substitution Cipher Decrypter (Console Input)

To find a “good solution”, we will put together all the previous pieces into an algorithm called **hill climbing**, which successively improves on a random starting cipher. You should implement it as follows:

1. Start with a random substitution cipher key.
2. While fewer than 1500 trials in a row have not resulted in a replacement...
 - a. Randomly swap 2 letters to create a new key.
 - b. If the new key gets a higher score...
 - i. Replace the old key with the new key.
3. Output the result of the decryption.

You can generate a random number with possible values $\{0, 1, \dots, A\}$ with

`Random::randInt(A)`.⁴ This is a function we provide in `utils.h`. If the two indices you randomly select are the same, reroll only the second one until they are different.

Because we make one swap at a time, we might end up in a situation we call a *local maximum*, where making any one swap would result in a lower score, but this isn’t the best possible decryption. Reaching a better one would require multiple swaps – decreasing our score before finding something even better. As an analogy, we can walk to the top of a short hill, but have to go down before we can walk up a taller hill. To handle this situation, we run the hill-climbing algorithm in a *loop*:

⁴ Why can’t we use `rand() % (A+1)`? We want to get deterministic behavior out of the RNG for testing purposes. Unfortunately, `rand` isn’t guaranteed to have the same behavior across different machines, even when we seed with `srand_mt19937` has a stricter definition and does have this guarantee, but is also more complicated to use, so we wrap it up in a nice function for you.

1. Run the hill-climbing algorithm 20 times.
2. Take the best-scoring decryption out of the 20 runs.

Since we use a different random starting cipher each time we hill-climb, we're likely to hit a different local maximum each iteration, helping us find the best overall.

Implement the “Decrypt Substitution Cipher (Console)” command, which reads one line of input from the console and applies the hill-climbing algorithm in a loop, as described. The prompt is “Enter text to substitution-cipher decrypt: “. Before decrypting, convert lowercase letters to uppercase (but leave all non-letter characters alone).

This command will take longer, since we're doing a significant amount of calculation!⁵ However, it shouldn't take more than 1 minute per execution on our test cases (Ethan's computer takes ~10 seconds each). If `make test_subst_dec` doesn't finish in a few minutes, look for optimization opportunities (see our [\[FA24\] Ciphers FAQ](#)) or come to help hours!

Task: Substitution Cipher on Files

One more command!

Implement the “Decrypt Substitution Cipher (File)” command, which is in response to “f” or “F”. This command reads two lines of input from the console: an input filename (which contains ciphertext encrypted with a substitution cipher), and an output filename. It reads the input file, decrypts the contents, and writes the decrypted text to the given output file. As usual, lowercase letters are converted to uppercase and non-letter characters are left alone.

Use `make run_ciphers` to run your program. Use the newly implemented command to decrypt `cryptogram.txt`, and output to a file called `plaintext.txt`.

This command will only be checked on Gradescope – you **must** check there to make sure your command is correct. You can resubmit as many times as you want on Gradescope until the deadline.

For quicker feedback from the autograder, we'll run the command on a smaller file. We won't run the command to create `plaintext.txt`, but will check its contents. Ethan's computer takes a

⁵ There's definitely optimization opportunities beyond what our reference solution does! They just use things out of scope for the first project... but things like `string_view` or clever reuse of already-allocated `strings/vectors` would help, along with swapping `unordered_map` for a better implementation.

bit over 2 minutes to decrypt `cryptogram.txt`. You should create smaller files to test this command yourself by running your program manually.

Submitting

Upload `ciphers.cpp` and `plaintext.txt` to the Gradescope assignment.

Grading Breakdown

In CS 251, project grading is based on **milestones**. To complete a milestone, you'll need to both:

- Complete **every** milestone before it, and
- Pass *all* of its tests

Every project will be out of 100 points.

Milestone	Total Points
Caesar Cipher Encoder <code>make test_caesar_enc</code>	10
Caesar Cipher Decoder <code>make test_caesar_dec</code>	60
Substitution Cipher Encoder <code>make test_subst_enc</code>	65
Substitution Cipher Decoder <code>make test_subst_dec</code>	90
Substitution Cipher on Files <code>plaintext.txt</code> , Gradescope only	100

You can run all tests for the entire project with `make test_all`, except for the last command.

Acknowledgements

Assignment by Adam Blank, adapted for CS 251 by Ethan Ordentlich.