Programming Assignment: Transitioning from C to C++ [Part 3]

Learning Outcomes

- To gain experience with file input/output techniques
- To gain experience with formatting output
- To gain experience with namespaces
- To practice overall problem-solving skills, as well as general design of a program
- Reading, understanding, and interpreting C++ code written by others

Task

Cryptography is a mathematical science that deals with transforming data to hide its meaning and preventing it from being read or tampered with by unauthorized parties so that messages can be securely exchanged between two participants even in the presence of adversaries. Much of this is accomplished through *encryption* and *decryption* of data. Encryption makes data incomprehensible in order to ensure its confidentiality. Encryption uses an algorithm called a *cipher* and a secret value called the *key* that is known only by the sender and legitimate receiver(s) to change *plaintext* [the data to be kept secret] into *ciphertext* [the resulting garbled and therefore unreadable message]. The art of cryptography has been used to code messages for thousands of years and continues to play a primary role in global ecommerce, financial transactions, computer passwords, and for securing databases.

Like cryptography, steganography is a data securing technique. Rather than transforming data to hide its meaning, steganography is the practice of concealing information within other nonsecret text or data to prevent eavesdropping. Steganography is focused on hiding the presence of information, while cryptography is more concerned with making sure that information cannot be accessed. When steganography is used properly, no one – apart from the intended recipients – should be able to detect there is any hidden communication taking place. This makes it a useful technique for situations where obvious contact between participants is unsafe. If you were an imprisoned mafia don and you wish to secretly pass orders to your minions without being detected by your jailers, steganography is an ideal choice. Why? Since the authorities will be checking every message you send from your cell and it will be obvious you're sending a secret message if the message is encrypted. Instead, you'll have to hide the fact that you're sending a secret message by obscuring it within a plain message. In contrast, cryptography tends to be used in situations where the participants aren't concerned if anyone finds out that they are communicating, but they need the message itself to be hidden and inaccessible to unauthorized parties. Suppose you were a diplomat [a far better career choice than a mafia don] based in an enemy country. It is expected that diplomats discuss secrets with their governments and it is also expected that host governments will intercept such diplomatic messages. Therefore, communications between you and your government would use cryptography rather than steganography because it is critical the host government doesn't uncover these secret messages when they're intercepted.

Text steganography is a mechanism of creating a covering message that hides a secret text message. The task is to implement function `extract` with declaration

```
void extract(char const *filename, char const **keywords);
```

that decodes and prints to standard output stream a secret message hidden *steganographically* in the following way:

1. Parameter `filename` specifies the name of a text file that contains a *cover message*. The cover message is a sequence of characters broken into *words*, where words are separated from each other by whitespace [space, tab, or the newline].
2. Parameter `keywords` points to the first element of an array of elements of type `char const*`. Each array element points to a *keyword* except the last element which is a null pointer and has value `nullptr`.

   > ***Keyword `nullptr` for the null pointer is new in C++11. In old C++ code and C code, people often use `0` (zero) or `NULL` instead of `nullptr`. Both older alternatives can lead to confusion and/or errors, so prefer the more specific `nullptr`. This is the "modern C++" way.***

3. If a file specified by parameter `filename` [say, the file is named cover-message.txt] doesn't exist, function `extract` must print to standard output stream the following error message and return:

   ```
   File cover-message.txt not found.

   ```

4. If the text file contains the cover message

   ```
   we ask you
   please don't feed my shark
   panic may ensure
   please manage your fear

   ```

   with shark and please as the keywords specified by parameter `keywords`, function `extract` must print to standard output stream the *secret message*:

   ```
   don't panic manage

   ```

   > ***Note the secret message is printed with its words separated by a space character [including the last word in the message] followed by a newline. This will be the consistent manner in which your code should write the secret message to the standard output stream.***

   The algorithm for extracting the secret message is straightforward:

   - For each word `word` in the cover message
     - If `word` matches a keyword, write to standard output stream the next word in the cover message [and a whitespace character]
   - Write newline to standard output stream

Confirm that the order of keywords specified by parameter `keywords` does not affect the secret message extracted from the cover message.

5. To improve data security, the basic extraction algorithm described above requires a minor tweak. Consider a slightly amended cover message:

```
we ask you
don't feed my shark please
panic may ensure
please act very calm to avoid being eaten

```

With shark and please as the keywords specified by parameter `keywords`, function `extract` must print the secret message:

```
please act

```

and NOT this secret message:

```
please panic act

```

Why? If the cover message contains two adjacent keywords [as with the last two words shark and please of the cover message's second line: don't feed my shark please], the second keyword please *doesn't act as a keyword and is instead considered as part of the cover message*. Thus, when keyword shark in the cover message is encountered, the next word please is extracted as part of the secret message. However, since adjacent word please in the cover message matches a keyword, it is now considered as a part of the cover message and not as a keyword.

Implementation Details

Header file q.hpp

You must provide a declaration of function `extract` in namespace `hlp2` in header file q.hpp. It would look like this:

```
#ifndef Q_HPP_
#define Q_HPP_

// No C++ standard library headers must be included here since the
// declaration of function extract doesn't make any references to
// names from the standard library!!!
// Any inclusion will prevent your submission from being compiled!!!

namespace hlp2 {
  // declare function extract here ...
}

#endif
```

Source file q.cpp

You must provide a definition of function `extract` in namespace `hlp2` in source file q.cpp. It would look like this:

```
// You cannot use any I/O functions from the C standard library
// such as printf, fprintf, scanf, fscanf, puts, gets, ...

// Include whatever C++ standard library headers you want ...

// Yes - you can include any headers that are part of the
// C standard library (except for I/O functions).
// For example if you want to use the C-string facilities in <string.h>,
// you must write the following preprocessor directive:
#include <cstring>

namespace hlp2 {
  // provide definition of function extract here ...
}

```

Algorithm outline

One algorithm for implementing function extract would look like this:

```
void extract(char const *filename, char const **keyword) {
  Create input file stream that opens disk file with name filename
                                                      in read mode
  if input file stream cannot be created, write error message and return

  while (input file stream is in good state) {
    read line of text from input file stream

    for each word in line
      if previous word was a keyword
        write current word to standard output stream followed by
                                                 space character
      else
        check if current word is a keyword
  }
  write newline to standard output stream
}
```

Reading entire line from text file

Class `istream` implements overloaded member functions `getline` that extract characters from an input stream as a null-terminated character string. Since input file stream class `ifstream` inherits from class `istream`, all of the member
functions and operators that you can apply to an `istream` object can also be applied to an `ifstream` object. Therefore, member function `getline` of class `istream` can be used by an input file stream object to read entire lines from a text file. The following code snippet prints the contents of a text file to the standard output stream:

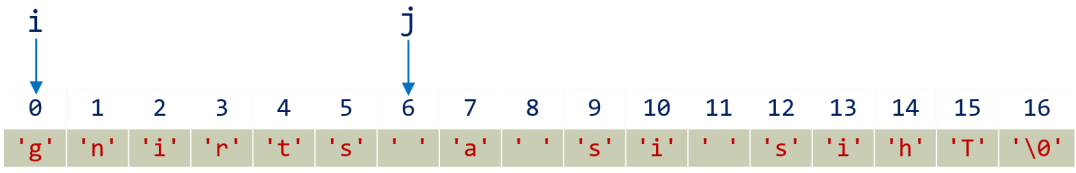
```
char const *filename = "some-text-file";
std::ifstream ifs{filename};

// assuming some-text-file exists in current directory ...
constexpr size_t MAX_LINE_LENGTH {256};
char line[MAX_LINE_LENGTH];
while ( ifs.getline(line, MAX_LINE_LENGTH-1) )
  std::cout << line << '\n';
```

Member function `getline` reads characters into the pointed-to character array `line`, reading until it has either encountered a `'\n'` or has read `MAX_LINE_LENGTH - 1` characters. It terminates the string of characters with `'\0'` so you get a valid C string regardless of how much was read. As long as the supplied `MAX_LINE_LENGTH` is less than or equal to the array length, you will not overflow the array. Conveniently, the `'\n'` character is removed from the stream and is not placed into the array, which is usually what you want in order to read and process the information in a series of lines. The returned value is a reference to the `istream` object, which can be used to test the stream state. If it fills the array without finding the `'\n'`, the input
operation fails in the same way as invalid input to let you know that a newline was not found within the size of the line you are trying to read.

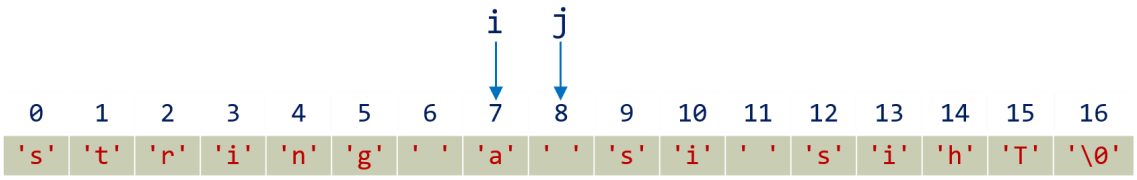
Splitting a line into words

How can we break a character string containing a line of text from a text file into words, separated from each other by whitespace [space or tab] characters? The second pass repeatedly identifies a word in the string using subscripts `i` and `j` so that `str+i` points to the first character in a word while `str+j` points to the whitespace character following the final character in the word. The second pass will then call helper function `reverse(str+i, str+j)` to reverse characters of the delimited word. The individual steps in this second pass are detailed below:

- Find subscripts `i` and `j` for the first word.
    1. Begin by setting `i` to `0`.
    2. Skip whitespace characters (if any) to set `i` with subscript at which the first non-whitespace character occurs. This is the subscript at which the first word begins. Skipping whitespace characters will also take care of the edge case where the string has whitespace characters preceding the first word in the string, as in `"  today  is a good day"`. Another edge case to consider is a string that only contains whitespace characters, as in `"  "` or `"  \t  "`.
    3. With subscript `i` identifying where a word begins, a second subscript `j` is required to identify where the word ends. Begin by setting `j` with the value of `i` and then increment `j` to skip non-whitespace characters until the first non-whitespace character is encountered. Characters in subscript range `[i, j)` will delimit the first word. The following picture illustrates the values of subscripts `i` and `j` for string `"gnirts a si sihT"`:



- Reverse the characters for the first word in the substring delimited by subscripts `i` and `j` in range `[i, j)` using helper function `reverse` like this: `reverse(str+i, str+j)`.
- To find `i` and `j` for the second word, set `i` to value of `j` and repeat the steps used for delimiting the first word. The following picture illustrates the values of subscripts `i` and `j` for the second word in string `"string a si sihT"`.



- Again, reverse characters for second word in the substring delimited by indices `i` and `j` in range `[i, j)`.
- Continue the process of identifying individual words until there are no more words left in the string - that is, when `i` has subscript value equivalent to the length of the string [that is, `i` and `j` reach element having null character `'\0'`].

Driver q-driver.cpp

The driver implements multiple unit tests: the cover messages are in files cover-message?.txt and the corresponding correct secret message is in secret-message?.txt. That is, unit test defined in function `test1` uses a list of hard-coded keywords to generate a secret message secret-message1.txt from a cover message contained in file cover-message1.txt. Note that the auto grader will perform additional tests that are not shown to you - this is done to encourage students to test their code rigorously.

Compiling, executing, and testing

Download q-driver.cpp, and message files from the assignment web page. Create the executable program by compiling and linking directly on the command line:

```
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror q.cpp q-driver.cpp -o
q.out
```

In addition to the program's name, function `main` is authored to take an additional command-line parameter that [currently] specifies a character between 0 and 4 corresponding to the unit test the user wishes to execute. To execute unit test 0, run the program like this:

```
$ ./q.out 0 > your-secret-message0.txt
```

Compare your submission's output with the correct output in secret-message0.txt:

```
$ diff -y --strip-trailing-cr --suppress-common-lines your-secret-message0.txt
secret-message0.txt
```

If the diff command is not silent, your output is incorrect and must be debugged.

You'll have to repeat this process for the remaining tests.

File-level and Function-level documentation

This module will use Doxygen to tag source and header files for generating html-based documentation. An introduction to Doxygen and a configuration file is provided on the module web page.

Every source and header file *must* begin with a *file-level* documentation block.

Every function that you declare and define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value. In team-based projects, this information is crucial for every team member to quickly grasp the details necessary to efficiently use, maintain, and debug the function. Certain details that programmers find useful include: what does the function take as input, what is the output, a sample output for some example input data, how the function implements its task, and importantly any special considerations that the author has taken into account in implementing the function. Although beginner programmers might feel that these details are unnecessary and are an overkill for assignments, they have been shown to save considerable time and effort in both academic and professional settings. Humans are prone to quickly forget details and good function-level documentation provides continuity for developers by acting as a repository for information related to the function. Otherwise, the developer will have to unnecessarily invest time in recalling and remembering undocumented gotcha details and assumptions each time the function is debugged or extended to incorporate additional features.

Submission and Automatic evaluation

1. In the course web page, click on the submission page to submit the necessary file(s).
2. Read the following rubrics to maximize your grade. Your submission will receive:
    1. F grade if your submission doesn't compile with the full suite of $\mathrm{g{+}{+}}$ options [shown above].
    2. F grade if your submission doesn't link to create an executable.
    3. $A+$ grade if the submission's output matches the correct output. Otherwise, a proportional grade is assigned based on how many incorrect results were generated by your submission.
    4. A deduction of one letter grade for each missing documentation block. Every submitted file must have one file-level documentation block. Every function that you declare in a header file must provide a function-level documentation block. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and the three documentation blocks are missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the three documentation blocks are missing, your grade will be later reduced from C to F.