Lab: Transitioning from C to C++

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

The task is to implement a function `q` with prototype

```
void q(char const *filename, char const **key_words);
```

that decodes and prints to `stdout` a secret message hidden *steganographically* in the following way:

1. Parameter `filename` specifies the name of a text file with a sequence of words containing a hidden message.
2. Parameter `key_words` points to the first element of an array of elements of type `char const*`. Each element of the array points to a *keyword* except the last element which is a null pointer and has value `nullptr`.

   > ***The name `nullptr` for the null pointer is new in C++11, so 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.***

   The hidden message consists of every word in the file that immediately follows a keyword. So, if the keywords are `"please"` and `"shark"`, the text `we ask you please don't feed my shark panic may ensure` yields the message `don't panic` because the text contains the word `"don't"` after keyword `''please"` and it also contains the word `"panic"` after keyword `"shark"`.
3. If the file with name `filename` doesn't exist, the function must print an error message and return. If the nonexistent file name is `"message0.txt"`, the function must print out the following error message:

   ```
   File message0.txt not found.

   ```
4. If the file contains the text

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

   ```

and the keywords specified by parameter `words` are `"please"` and `"shark"`, the function must print the following to `stdout`:

```
don't panic

```

Note that there is a space character after the last word followed by a newline. This will be the consistent manner in which your code should write the secret message to `stdout`.

5. If the file contains the text

```
the rain in spain
falls mainly in the plain
or so eye ear

```

and the keywords specified by parameter `words` are `"so"`, `"rain"`, and `"in"`, the function must print the following to `stdout`:

```
in the eye

```

and NOT

```
in spain the eye

```

Why? If the file contains two keywords in a row, the second keyword doesn't act as a keyword, instead it is considered as part of the message. Suppose the file contains text `the rain in spain falls mainly in the plain or so eye ear`. With keywords `"so"`, `"rain"`, and `"in"`, the correct secret message is `in the eye` and not `in spain the eye`. This is because the file contains the word `"in"` after keyword `"rain"` and the secret message begin with the word `"in"`. However, even though `"in"` itself is a keyword, the word `"spain"` following `"in"` is not considered a part of the secret message because `"in"` is the second keyword that appears in a row following the first keyword `"rain"`.

Details

Header file `q.hpp`

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

```cpp
#ifndef Q_HPP_
#define Q_HPP_

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

#endif
```

Source file `q.cpp`

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

```cpp
// 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>

// Important note: You cannot make use of any I/O functions from the
// C standard library such as printf, fprintf, scanf, fscanf, puts, gets,
...
// If you do so, it'll be a violation of the Academic Integrity Policy.
// You're warned!!!

namespace hlp2 {
  // provide definition of q here ...
}

```

Driver `q-driver.cpp`

The driver implements five units tests: the input messages are in files `message?.txt` and the corresponding correct secret message is in `secret-message?.txt`. That is, the unit test defined in `test1()` uses a list of hard-coded keywords (that you can see) to generate a secret message `secret-message1.txt` from a message contained in file `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.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Header file

Submit `q.hpp`.

Source file

Submit `q.cpp`.

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
lab1.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 zero, run the program like this:

```
$ ./lab1.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 documentation

Every source and header file *must* begin with a *file-level* documentation block. 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. Here is a sample for a C++ source file:

```
/*!*****************************************************************************
\file    scantext.cpp
\author  Prasanna Ghali
\par     DP email: pghali\@digipen.edu
\par     Course: HLP2
\par     Section: student - provide your section here
\par     Programming Assignment #6
\date    11-30-2018

\brief
  This program takes strings and performs various tasks on them via several
  separate functions. The functions include:

  - mystrlen
      Calculates the length (in characters) of a given string.

  - count_tabs
      Takes a string and counts the amount of tabs within.

  - substitute_char
      Takes a string and replaces every instance of a certain character with
      another given character.

  - calculate_lengths
      Calculates the length (in characters) of a given string first with tabs,
      and again after tabs have been converted to a given amount of spaces.

  - count_words
      Takes a string and counts the amount of words inside.
*******************************************************************************/
```

Function-level documentation

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. Here is a sample for function `substitute_char`:

```
/*!****************************************************************************
****
\brief
   Replaces each instance of a given character in a string with
   other given characters.

\param str
  The string to walk through and replace characters in.

\param old_char
  The original character that will be replaced in the string.

\param new_char
  The character used to replace the old characters

\return
  The number of characters changed in the string.
*******************************************************************************
***/
int substitute_char(char *in, char old_char, char new_char) {
  // ...
}
```

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary files.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - F grade if your submission doesn't compile with the full suite of `g++` options.
   - F grade if your submission doesn't link to create an executable.
   - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will assign 50% of the grade based on the input and output files given to you. The remaining 50% of the grade will be awarded based on the additional tests implemented by the auto grader.

- The auto grade will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if your output matches correct output of auto grader.
- A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is 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 two documentation blocks are missing, your grade will be later reduced from C to E.