Lab: Transitioning from C to C++ [Part 1 - File I/O]

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

This lab is concerned with cloning the Linux utility wc. You can obtain more information about this utility from the Linux system reference manuals by running the following command in your Linux console:

```
$ man wc
```

You call wc with one or more command-line arguments that specify the text files for which you want to print newline, word, and byte counts for each file, and a total line if more than one file is specified. Additionally, wc takes a variety of command-line options to print specific information [such as the length of the longest line]. This assessment doesn't require the implementation of these command-line options.

> ***You should assume only Linux text files will be used to test your submission. Recall that Linux uses a newline [line feed] character `'\n'` to indicate the end of a line in a text file. On the other hand, Windows uses the carriage return character `'\r'` followed by a newline [line feed] character `'\n'` to signify the end of a line in a text file.***

> ***Because of the different end of line characters, it is important that you only use Linux tools.***

You first task is to understand wc's behavior by reading its manual pages and by examining its output for both an individual file or a variety of input files. You should also test wc's behavior when it is given files that don't exist in the disk. wc doesn't seem to have a consistent format for printing the counts to the standard output stream. Therefore, you must exactly mimic the pretty tabular data that is printed by my clone of wc.

The driver source file wc-driver.cpp indicates that you must declare [in header file wc.hpp] and define [in source file wc.cpp] function `wc` like this:

```
namespace hlp2 {
  void wc(int argc, char *argv[]);
}
```

where `argc` and `argv` are the command-line parameters passed to function `main`.

Implementation Details

Header file wc.hpp

You must provide a declaration of function `wc` in namespace `hlp2` in header file wc.hpp. See the incomplete wc.hpp for more details.

Source file wc.cpp

You must provide a definition of function `wc` in namespace `hlp2` in source file wc.cpp. See the incomplete wc.hpp for more details.

> ***You cannot use any functions from the C standard library including I/O functions such as `printf`, `fprintf`, `scanf`, `fscanf`, `puts`, `gets` and character string functions such as `strlen`. See the incomplete wc.hpp and wc.cpp provided to you for more details.***

> ***You cannot use any types and functions from the C++ standard library other than those exposed by headers `<iostream>`, `<iomanip>`, and `<fstream>`.***

Algorithm

Outline

The [rough] pseudocode for an algorithm to be implemented by `wc` would look like this:

```
void wc(int argc, char **argv) {
  initialize total line, word, and byte counters
  for each file specified in argv
    initialize line, word, and byte counters
    create and initialize file stream that connects disk file to program
    for each line in file [that has been read into a static array]
      (see incomplete wc.hpp and wc.cpp for more details)
      a) if line is terminated by end of line character,
         increment line counter
      b) increment byte counter with number of characters in line
      c) split line into words and increment word counter by
         line's word count
    end for each line in file
    close file stream
    print line, word, and byte counts in pretty tabular format
    update total line, word, and byte counters
  end for each file specified in argv
  print total line, word, and byte counters in pretty tabular format
}
```

You should review lecture presentation deck and source code to learn about writing to standard output stream, stream manipulators, and opening, closing, reading from, and writing to text files. However, there are two things missing: how to read a file line-by-line and how to split a line into words.

> ***You cannot use any functions from the C standard library including I/O functions such as `printf`, `fprintf`, `scanf`, `fscanf`, `puts`, `gets` and character string functions such as `strlen`. See the incomplete wc.hpp and wc.cpp provided to you for more details.***

> ***You cannot use any types and functions from the C++ standard library other than those exposed by headers `<iostream>`, `<iomanip>`, and `<fstream>`.***

Reading entire line from text file

Luckily for us, class `std::istream` defines member function `get` to extract one or more characters from the input file stream and member function `getline` to extract an entire line of text. Carefully read the man pages for these member functions and gain some experience in using them by implementing small toy programs [such as a file copy program]. It is extremely useful to gain exposure and experience to individual functions by exercising them individually in toy programs before using them in more complex environments.

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 line-by-line to the standard output stream:

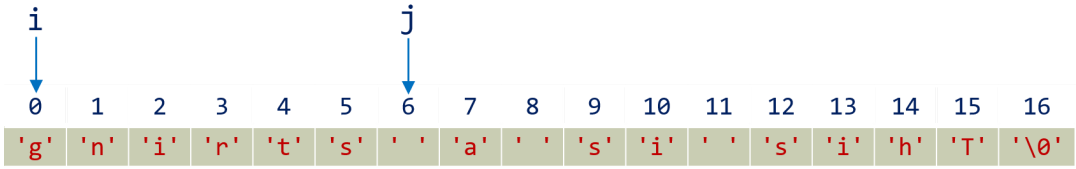
```
1  char const *filename = "some-text-file";
2  std::ifstream ifs{filename};
3  // assuming some-text-file exists in current directory ...
4  constexpr size_t MAX_LINE_LEN {2048};
5  char line[MAX_LINE_LEN];
6  while ( ifs.getline(line, MAX_LINE_LEN-1) )
7    std::cout << line << '\n';
```

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_LEN - 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_LEN` 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.

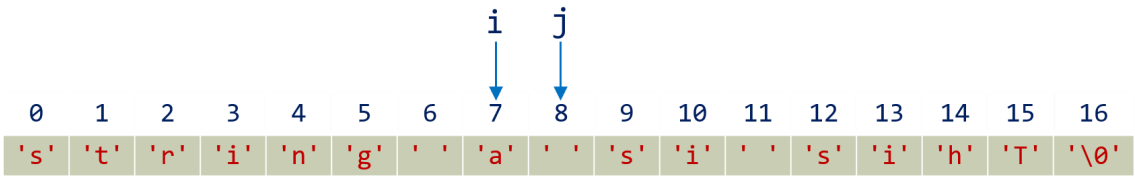
Splitting a line into words

How can we split a character string containing a line of text from a text file into words, separated from each other by whitespace [space or tab] characters? Because breaking a line of input into words that are separated from each other by whitespace is an operation that might be generally useful in many scenarios, you should write a function to perform the split. Given a line of text stored in a static array `str` as a character string `gnirts a si sihT`, two subscripts `i` and `j` are required to demarcate a word: `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. Notice that `str+i` and `str+j` define a half-open range [the concept of a half-open rang ewas explained in HLP1]. The individual steps of the splitting process 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  \n"`.
    3. With subscript `i` identifying where the 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 [that constitute the word] until the first 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"`:



- To find `i` and `j` for the second word, assign the value of `j` to `i` 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"`.



- 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 with null character `'\0'`].

Driver wc-driver.cpp

The driver simply forwards to function `wc` the file name(s) specified as command-line parameters to function `main`.

Compiling, executing, and testing

Download wc-driver.cpp, and text files from the assessment's web page. Create the executable program by compiling and linking directly on the command line:

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

You should continually check your program's output to the correct output generated by wc and my clone of wc. Remember, you should format the output exactly as implemented in my clone.

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.