Lab: Transitioning From C to C++ [Part 2 - Lvalue References, Dynamic Memory, File I/O]

Learning Outcomes

- Gain experience with namespaces, references, dynamic memory allocation/deallocation expressions, file I/O techniques, formatting output
- Gain experience in defining simple user-defined types in C++ and learn about C++ standard library type `std::string`
- Practice overall problem-solving skills, as well as general design of a program
- Reading, understanding, and interpreting C++ code written by others

Task

The objective of this lab is to only use facilities provided by the C++ standard library in headers `<iostream>`, `<iomanip>`, `<fstream>`, and `<string>` to develop a program that:

1. Reads information about an unknown number of tsunami events from a text file into a dynamically allocated array of structures,
2. Processes the data in the array, and
3. Prints a report and summary statistics about these tsunami events to an output text file.

The summary statistics reported by the program include the largest maximum wave height, the average wave height of all tsunami events, and the location of all tsunamis with a wave height higher than the average height. All heights in the data file are measured in meters.

Brief Overview of `std::string`

The C++ standard library provides a flexible and convenient way to represent a character string consisting of `char`s using an object of class `std::string` that is defined in header `<string>`. You can define and manipulate variables that hold character strings like this:

```
#include <string>

std::string pest {"Bart"};
pest += " Simpson"; // or, pest = pest + " Simpson";
std::cout << pest << " is a secret genius!!!";
```

In C++, you'll need to distinguish between string variables [such as the variable `pest` defined above] and string literals [character sequences enclosed in quotes, such as `"Bart"`]. The string stored in a string variable can change. A string literal denotes a particular string, just as a number literal [such as 2] denotes a particular number.

Unlike number variables, string variables are guaranteed to be initialized even if you do not supply an initial value. By default, a string variable is set to an empty string: a string containing no characters `""`. The definition

```
std::string pest;
```

has the same effect as

```
std::string pest = "";
std::string pest {};   // ditto as above
```

`std::string` is a user-defined class type composed of built-in types, possibly other user-defined types, and functions. The parts used to define the class are called *members*. Members can be of various kinds. Most are either *data members* which define the representation of an object of the class, or *member functions* which provide the operations on such objects. Our objective of using a user-defined type such as `string` is that we don't care how it has been implemented; instead, we're only interested in doing useful work by using its member functions. We access a member function using the `object.member` notation.

Member function `length` returns the number of characters in the string. For example:

```
std::string s1 {"Bart"};
std::cout << s1 << " has " << s1.length() << " characters.\n";
```

Member function `length` return a `string::size_type` value. What is this type? Class `string` - and most other library types - defines several companion types so that these library types can be used in a machine-independent manner. Type `size_type` is one of these companion types. It is unsigned type `size_t` [declared in the C++ standard library header `<cstddef>`] which is a type big enough to hold the size of any `string`. Any variable used to store the result from member function `length` should be of type `string::size_type`.

Because class `string` overloads subscript operator `[]`, you can access characters in a `string` object in the same way that you access array elements. A `string` knows its size, so we can print the elements of a `string` like this:

```
std::string s {"Lisa Simpson"};
for (std::string::size_type i{0}; i < s.length(); ++i) {
  std::cout << s[i];
}
```

The call `s.length()` gives the number of elements of `string` variable `s`. In general, `s.length()` gives us the ability to access elements of a `string` without accidentally referring to an element outside the `string`'s range. The range of elements for `s` is $[0{:}s.length())$. That's the mathematical notation for a half-open sequence of elements. The first character of the string encapsulated by `s` is `s[0]` and the last character is `s[s.length()-1]`. If `s.length()==0`, `s` has no elements, that is, `s` is an empty `string`. This notion of half-open sequences is used throughout the C++ standard library.

C++ takes advantage of the notion of a half-open sequence to provide a simple range-based `for` loop over all the elements of a string:

```
std::string s {"Lisa Simpson"};
for (char ch : s) {
  std::cout << ch << ' ';
}
```

You can use a range-based `for` loop to convert all characters in the string to upper case:

```
std::string s {"Lisa Simpson"};
for (char &ch : s) {
  ch = std::toupper(ch); // toupper is declared in <cctype>
}
```

You can create copies of `string` objects

```
std::string pest{"Bart Simpson"}, s1 = pest, s2 {s1};
```

assign to `string` variables

```
std::string one {"othello"}, two; // two is an empty string
two = one;          // assign "othello"
two = "two\nlines"; // assign a C-style string
two = 'x';          // assign a single character
```

compare `string` objects

```
std::string first {"aaa"}, second {"aaaa"};
std::cout << (first < first) << "\n";  // false
std::cout << (first <= first) << "\n"; // true
std::cout << (first < second) << "\n"; // true
```

Given two strings, such as `"Lisa"` and `"Simpson"`, you can concatenate them to one long string using operator `+`. The result consists of all characters in the first string, followed by all characters in the second string. For example,

```
std::string first_name = "Lisa", last_name = "Simpson";
std::string smart_cookie = first_name + last_name;
```

results in the string `"LisaSimpson"`. What if you'd like the first and last name separated by a space? No problem:

```
std::string smart_cookie = first_name + " " + last_name;
```

This statement concatenates three strings: `first_name`, the string literal `" "`, and `last_name`. The result is `"Lisa Simpson"`.

You can read a string from the console:

```
std::cout << "Enter your name: ";
std::string name;
std::cin >> name;
```

When a string is read with operator `>>`, only one word is placed into the string variable. For example, suppose the user types

```
Lisa Simpson
```

as the response to the prompt. This input consists of two words. After the call `std::cin >> name`, the string `"Lisa"` is placed into variable `name`. Use another input statement to read the second word.

Once you have a string, you can extract substrings by using member function `substr`. The member function call

```
s.substr(start, length)
```

returns a string that is made from the characters in string `s`, starting at index `start`, and containing `length` characters. Here is an example:

```
std::string greeting = "Hello, World!";
std::string sub = greeting.substr(0, 5); // sub is "Hello"
```

The `substr` operation makes a string that consists of five characters taken from string `greeting`. Indeed, `"Hello"` is a string of length 5 characters starting at index 0 that occurs inside `greeting`.

Let's figure out how to extract substring `"World"`. Counting characters starting at index 0, you find that `'W'` has index 7. The string you want is 5 characters long. Therefore, the appropriate substring command is:

```
std::string w = greeting.substr(7, 5); // w is "World"
```

If you omit the length, you get all characters from the given position to the end of the string. For example,

```
greeting.substr(7)
```

is the string `"World!"` [including the exclamation point].

Here is a simple program that puts basic concepts to work. The program asks for your name and that of your significant other and then prints out your initials. Operation `first.substr(0, 1)` makes a string consisting of one character, taken from the start of `first`. The program does the same for `second`. Then it concatenates the resulting one-character strings with string literal `"&"` to get a string `initials` of length 3:

```cpp
#include <iostream>
#include <string>

int main() {
  std::cout << "Enter your first name: ";
  std::string first;
  std::cin >> first;
  std::cout << "Enter your significant other's first name: ";
  std::string second;
  std::cin >> second;
  std::string initials = first.substr(0, 1) + "&" + second.substr(0, 1);
  std::cout << initials << '\n';
}
```

The C++ standard library allows you to use objects of type `string` anywhere you can use character strings. Last week we learnt to open a file whose name is specified as a character string:

```cpp
std::cout << "Enter file name:";
char filename[80];
std::stream_size old_width = std::cin.width(sizeof(filename));
std::cin >> filename;
std::cin.width(old_width);
std::ifstream in_file{filename};
```

It is much simpler to open a file using a `string`:

```cpp
std::cout << "Enter file name:";
std::string filename;
std::cin >> filename;
std::ifstream in_file{filename};
```

Class `string` has many more member functions and the standard library provides a variety of non-member functions.

Implementation Details

Begin the solution by looking at the input and corresponding output files to understand what needs to be done. By studying the format of data in input text file tsunamis1.txt, you'll notice that each *line* of the text file contains information for each tsunami event in the following order:

- integer value representing the month when the tsunami occurred,
- integer value representing the day when the tsunami occurred,
- integer value representing the year when the tsunami occurred,
- integer value representing the number of fatalities caused by the tsunami,
- double-precision floating-point value representing maximum wave height during the tsunami, and
- a sequence of characters [that may contain whitespace characters] representing the tsunami's location.

Examine text file output1.txt, where you'll find the correct and expected output that should be generated by your program for the tsunami events in tsunamis1.txt.

The next step is to examine the requirements from your client with respect to the specific types and functions that you must implement.

Header file q.hpp

In header file q.hpp, define type `Tsunami` that will encapsulate heterogeneous information relevant to a tsunami event and declare two functions to manipulate objects of type `Tsunami`:

```
#ifndef Q_HPP
#define Q_HPP

// Never include headers that will be required by the implementation
// in this file. Instead, you must include only those standard library
// headers required by declarations in this file ...

#include <string> // required when using standard library type std::string

namespace hlp2 {

  struct Tsunami {
    // data members
  };

  // declaration of interface functions ...
  Tsunami* read_tsunami_data(std::string const& file_name, int& max_cnt);
  void     print_tsunami_data(Tsunami const *arr,
                              int size, std::string const& file_name);

} // end namespace hlp2
#endif

```

The data members encapsulated in `Tsunami` must include the month, day, and year of the tsunami's occurrence [represented by `int`s]; its maximum wave height [represented by a `double`]; the number of fatalities [represented by an `int`]; and the geographical location [represented by a `std::string`].

A brief summary of the interface follows:

1. Function `read_tsunami_data` declared as

   ```
   Tsunami* read_tsunami_data(std::string const& file_name, int& max_cnt);
   ```

   must do the following:

   - determine the number of tsunami events recorded in the input file specified by parameter `file_name` and assign this count to parameter `max_cnt`,
   - if the input file doesn't exist, the function should return a `nullptr`,
   - dynamically allocate an array [where each element is of type `Tsunami`] with `max_cnt` number of elements,
   - read data for each tsunami event from the input file and record the data in the corresponding element of the dynamically allocated array,

- return a pointer to the first element of the dynamically allocated array.

2. Function `print_tsunami_data` declared as

```
void print_tsunami_data(Tsunami const *arr,
                        int size, std::string const& file_name);
```

prints a report of the tsunami events in the array specified by parameter `arr` with `size` number of elements to an output text file whose name is specified by parameter `file_name`. Sample output text files output?.txt provide examples of the pretty table that your submission must generate. First, the function must open an output file stream that connects your program to the file name specified by parameter `file_name`. If the output file stream cannot be created in a good state, the function should just return. Next, the function should print a nicely formatted list of the information recorded in each tsunami event as specified by the output file and in the following order and format:

- month [formatted with 2 digits, left padded with zeroes],
- day [formatted with 2 digits, left padded with zeros],
- year [formatted with width of 4 digits],
- number of fatalities [right-aligned],
- maximum wave height [right-aligned with precision of exactly 2 digits], and
- the location of the tsunami [left-aligned].

Next, the function must print certain summary statistics including

- the largest maximum wave height of the tsunami events recorded in the array,
- average maximum wave height of all the tsunamis,
- a formatted table of the list of the maximum wave heights and locations of those tsunamis with higher maximum wave heights than the average maximum wave height of all the tsunamis.

Source file q.cpp

You're now in a position to devise an overall algorithm that takes the input and transforms it to the corresponding output using the specific requirements imposed on you. To implement the algorithm in C++ , begin by authoring a source file q.cpp that contains stub definitions of the necessary interface functions:

```
// include ONLY these C++ header files ...
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include "q.hpp" // compiler needs definition of type Tsunami

// anonymous namespace for any helper functions that you wish to implement
namespace {
  // ...
}

namespace hlp2 {
  // provide definitions of functions here ...
```

```
15 }
```

> ***You should implement your algorithm(s) using only the following C++ standard library headers: `<iostream>`, `<iomanip>`, `<fstream>`, and `<string>`. You cannot use C standard library headers - such headers are unnecessary and the auto grader will not accept your submission.***
>
> ***Read this again: Your implementation can include headers `<iostream>`, `<iomanip>`, `<fstream>`, and `<string>`. Addition of any other C and C++ headers [even in comments] such as `<cstring>`, `<cstdlib>` and the use of functions not declared in these headers such as `std::printf`, `printf`, `std::malloc`, `std::strlen` or just `strlen` [even in comments] will prevent the auto grader from accepting your submission.***

The first step is to determine the number of tsunamis listed in the input file. Since there are as many tsunamis as the number of lines in the input file, you must get a line count by reading each line using member function `getline` of class `std::istream` [you should know this from last week]. Once you know how many lines there are in the input file, you know how many tsunamis there are. Dynamically allocate memory for the appropriate number of `Tsunami` objects.

Before re-reading each line, you must set the next character to read from the file stream back to the top of the file using member function `seekg` of class `std::istream`. For each line, numerical data must be read using extract operator `>>`. The final portion of the line that contains the tsunami's location is then read using member function `getline` of class `std::istream`. You will notice that the tsunami's location is preceded and followed by whitespace characters. These whitespace characters must be trimmed to extract the tsunami location. The information read from each line must be stored into the corresponding element of the dynamic array.

You should think about authoring the following helper functions [in an anonymous namespace]:

1. Function `largest_max_ht` to return the largest maximum wave height in the tsunami events.
2. Function `avg_max_ht` to return the average maximum wave height of the tsunami events.

The next challenge is to write a report and summary statistics about the tsunami events to an output file in a pretty tabular format with proper alignment so that your output exactly matches the sample output files. You should look at the manipulators presented in `<iomanip>`: `left`, `right`, `setw`, `fixed`, `setprecision`, `setfill`, and so on.

Driver q-driver.cpp

The driver accepts as command-line parameters the name of the input file to analyze and the name of the output file containing the analysis. You are given three files to test: tsunamis1.txt, tsunamis2.txt, and tsunamis3.txt, and corresponding correct output files output?.txt.

Submission Details

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

Header and source files

You must submit q.hpp and q.cpp.

Compiling, executing, and testing

Download q-driver.cpp, and input and output 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 two command-line parameters to specify the name of the input file and the name of the output file.

```
$ ./q.out tsunamis1.txt your-output1.txt
```

Compare your submission's output with the correct output in output1.txt:

```
$ diff -y --strip-trailing-cr --suppress-common-lines your-output1.txt
output1.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 input files.

File-level and function-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. In addition, 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.

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 F.