Programming Assignment: Problem Solving with `vector`s and `string`s

Learning Outcomes

- Gain experience in functional decomposition and algorithm design
- Gain experience in using abstract data types `string` and `vector` and C++ standard library to solve practical problems
- Gain experience with function pointers
- Gain experience with C++ file input/output techniques
- Gain further experience with stream-based I/O and stream manipulators

Task

Your task is to read, process and transform an input file `worldpop.txt` that has a table of names and populations of countries into four output files containing tables with countries names sorted in *increasing* and *decreasing* orders and population counts sorted in *increasing* and *decreasing* orders. Notice that output files have a "pretty" format that makes their contents "nicer" to view and easier to understand. Also notice that output files represent a country's name *exactly* as found in the input file - we don't want citizens of these countries to take umbrage at mangled or misrepresented names.

Implementation Details

Begin the solution by looking at the input file `worldpop.txt` and corresponding [correct] output files `name-asc.txt` (country name in *ascending* order), `name-des.txt` (country name in *descending* order), `pop-asc.txt` (country population in *ascending* order), and `pop-des.txt` (country population in *descending* order). These files will give you a general idea of how your implementation must transform the input data to an output format. Pencil an algorithm that takes the input and transforms it to the corresponding output.

On each line, input text file `worldpop.txt` lists a country's name and its population. Here are a few opening lines from the file:

```
Mexico        120,286,655
Kosovo          1,859,203
Rwanda         12,337,138
Greece         10,775,557
Turks and Caicos Islands            49,070
Hungary         9,919,128
Sierra Leone          5,743,725
```

All we can say is that each line has two columns:

- The first column represents a country's name that can be a single word ***France*** or multiple words ***Solomon Islands***. Some country names may contain punctuation characters such as comma ***Gambia, The***, dash ***Guinea-Bissau*** or brackets ***Cocos (Keeling) Islands***.
- The second column represents the country's population count as a string of comma-separated numerals, as in ***38,813,722***.

Define structure `CountryInfo` in file `pa.hpp` that encapsulates a country's name and population:

```
#ifndef PA_HPP
#define PA_HPP

// necessary includes of header files except the following:
// <algorithm> and C-only headers such as <cstring>, ...

namespace HLP2 {

// structure to encapsulate a country's name and its population ...
struct CountryInfo {
  std::string name; // country's name - *exactly* as in input file
  long int pop;     // country's population
};

// other declarations ...

} // end namespace HLP2

#endif
```

Declare type name `Ptr_Cmp_Func` in file `pa.hpp`:

```
using Ptr_Cmp_Func = bool (*)(CountryInfo const&, CountryInfo const&);
```

Here, we've a `using` declaration that says "`Ptr_Cmp_Func` is a name for type *pointer to a function taking two parameters both of type read-only reference to* `CountryInfo` *and returning a* `bool`."

In traditional C++ (just as in C), storage specifier `typedef` was used to declare new names for existing types. For example, a new name `UI32` can be declared for existing type `unsigned int`:

```
typedef unsigned int UI32;
```

In modern C++, we write the same declaration in a simpler way with a `using` declaration:

```
using UI32 = unsigned int;
```

The corresponding declaration of type name `Ptr_Cmp_Func` with `typedef` would look like this:

```
typedef bool (*Ptr_Cmp_Func)(CountryInfo const&, CountryInfo const&);
```

Declare the following functions in `pa.hpp`:

```
std::vector<CountryInfo> fill_vector_from_istream(std::istream& is);
size_t max_name_length(std::vector<CountryInfo> const&);
void sort(std::vector<CountryInfo>& rv, Ptr_Cmp_Func cmp);
void write_to_ostream(std::vector<CountryInfo> const& v,
                      std::ostream& os, size_t fw);

bool cmp_name_less(CountryInfo const& left, CountryInfo const& right);
bool cmp_name_greater(CountryInfo const& left, CountryInfo const& right);
bool cmp_pop_less(CountryInfo const& left, CountryInfo const& right);
bool cmp_pop_greater(CountryInfo const& left, CountryInfo const& right);
```

A brief description of these functions is given below:

1. Define function `fill_vector_from_istream` whose declaration in `pa.hpp` looks like this:

   ```
   std::vector<CountryInfo> fill_vector_from_istream(std::istream& is);
   ```

   Using reference `is` to an input stream (presumably to a file such as `worldpop.txt`), the function will read a line of text, parse the line into a `string` specifying a country's name and a `long int` specifying the country's population, and augment a `vector<CountryInfo>` container with this information. The pseudocode might look like this:

   ```
   define object of type std::vector<CountryInfo>
   for each line in input stream:
     a) extract string encapsulating country's name from line using
        family of string::find and std::substr functions
     b) extract string encapsulating country's population count from line
     c) scrub comma operator from this string using string::erase
     d) convert scrubbed population string to value of type long int using
        family of string::sto??? functions
     e) define and initialize object of type CountryInfo with extracted
        country name and population count
     f) add new CountryInfo object to end of vector
   return vector object to caller
   ```

2. Define function `max_name_length` whose declaration in `pa.hpp` looks like this:

   ```
   size_t max_name_length(std::vector<CountryInfo> const&);
   ```

   This function returns the length of the longest country's name in the `vector`. The function's pseudocode might look like this:

   ```
   set max_length to zero
   iterate through each element of vector using vector::operator[]:
     a) access name member of each element using structure-member operator
     b) get length of country name using string::size
     c) if length is larger than max_length, set length as max_length
   return max_length
   ```

3. The third function you must define has prototype:

   ```
   void sort(std::vector<CountryInfo>& rv, Ptr_Cmp_Func cmp);
   ```

Function `sort` sorts all elements in `vector<CountryInfo>` object referenced by `rv` using a sorting criterion specified by the comparison function pointed to by `cmp`. Refactor any of the sort algorithms studied in the previous programming module. The *selection sort* algorithm is explained here.

> ***This assignment will be one of the few instances where you will not be allowed to use any functions declared in `<algorithm>`. Therefore, don't include the text `<algorithm>` in your source file - even in a comment. The server will not accept your submission.***

4. Define following comparison functions declared in `pa.hpp`:

```
bool cmp_name_less(CountryInfo const& left, CountryInfo const& right);
bool cmp_name_greater(CountryInfo const& left, CountryInfo const& right);
bool cmp_pop_less(CountryInfo const& left, CountryInfo const& right);
bool cmp_pop_greater(CountryInfo const& left, CountryInfo const& right);
```

   Function `cmp_name_less` returns `true` if the country name referenced by `left` is *lexicographically less* than the country name referenced by `right`. Otherwise, the function returns `false`.

   Function `cmp_name_greater` returns `true` if the country name referenced by `left` is *lexicographically greater* than the country name referenced by `right`. Otherwise, the function returns `false`.

   Function name `cmp_pop_less` returns `true` if population of object referenced by `left` is *numerically less* than population of object referenced by `right`.

   Function name `cmp_pop_greater` returns `true` if population of object referenced by `left` is *numerically greater* than population of object referenced by `right`.

   These functions will be used as callback functions by function `sort` to provide specific sorting criteria that will enable the program to transform the contents of container of type `vector<CountryInfo>` into four different forms.

5. The final function you must define has prototype:

```
void write_to_ostream(std::vector<CountryInfo> const& v,
                      std::ostream& os, size_t fw);
```

   This function will write the contents of the container referenced by `v` into the output stream referenced by `os`. Each element of the container must be separated by a newline. The two data members of `CountryInfo` must be *left justified*. Parameter `fw` specifies the field width to be used when writing the country name. Research `std::left` to specify text alignment and `std::setw` to specify the field width of a value to be written to the output stream. Make sure that your insertions to the output stream match the sample output.

Begin by implementing *stub functions* for the functions that you must define. A stub is a skeleton of a function that is called and immediately returns. It is syntactically correct - it takes the correct parameters and returns the proper values. Although a stub is a complete function, it does nothing other than to establish and verify the linkage between the caller and itself. But this is a very important part of coding, testing, and verifying a program. At this point, the program should be compiled, linked, and executed.

Driver `pa-driver.cpp`

In addition to the program's name, the driver requires the name of the input file [you're given `worldpop.txt`]. The program will use your definitions to generate *four* output files: `your-name-asc.txt` (country name in *ascending* order), `your-name-des.txt` (country name in *descending* order), `your-pop-asc.txt` (country population in *ascending* order), and `your-pop-des.txt` (country population in *descending* order). You can then compare the input generated by your definitions to the correct output.

Carefully read the code in `pa-driver.cpp` to understand how the functions you're defining in `pa.cpp` and declaring in `pa.hpp` are being used by your client.

Submission Details

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

Header file

Submit `pa.hpp`.

Source file

Submit `pa.cpp`.

Compiling, executing, and testing

Running `make` with target `all` will bring executable `pa.out` up to date. Running `make` with target `test` will compile, link, and execute your program to generate *four* output files: `your-name-asc.txt` (country name in *ascending* order), `your-name-des.txt` (country name in *descending* order), `your-pop-asc.txt` (country population in *ascending* order), and `your-pop-des.txt` (country population in *descending* order); and then run `diff` to compare the output of these file with the correct output. Your source file `pa.cpp` is ready for submission only if the `diff` command in the `test` rule is silent. Otherwise, one or more of your function definitions is incorrect and will require further work.

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.

Selection Sort algorithm

One of the most common applications in computer science is sorting - the process through which data are arranged according to their values. We're surrounded by data. If data were not ordered, we would spend hours trying to find a single piece of information. Imagine the difficulty of finding someone's telephone number in your cellphone's contact list that was not ordered! In this document, we present a sorting algorithm called the selection sort that is simple to understand and straightforward to code in a function.

Although the type of elements in the array is of no significance to the algorithm itself, assume an array whose elements are assigned arbitrary integral values. The selection sort algorithm begins by finding the minimum value and exchanging it with the value in the first position in the array. Then the algorithm finds the minimum value beginning with the second element, and it exchanges this minimum with the second element. This process continues until reaching the next-to-last element, which is compared with the last element; the values are exchanged if they are out of order. At this point, the entire array of values will be in ascending order. This process is illustrated in the following sequences:

Original order:

5	3	12	8	1	9

Exchange minimum with value in subscript 0:

1	3	12	8	5	9

Exchange next minimum with value in subscript 1:

1	3	12	8	5	9

Exchange next minimum with value in subscript 2:

1	3	5	8	12	9

Exchange next minimum with value in subscript 3:

1	3	5	8	12	9

Exchange next minimum with value in subscript 4:

1	3	5	8	9	12

Array values are now in ascending order:

1	3	5	8	9	12

A hand implementation of the algorithm for a deck of cards can be visualized here. An algorithm for the selection sort can be designed like this:

Algorithm Selection Sort

Input: Array A of n elements
Output: Array A sorted in ascending order

1. $i = 0$
2. while $(i <= n - 2)$
3. find min : the index of smallest element in unsorted subarray $A[i]$ through $A[n-1]$
4. **if** $i \neq min$
5. $\text{swap}(A[i], A[min])$
6. $i = i + 1$
7. endwhile