HW: Paris Olympics

Objectives

- Work with one dimensional arrays
    - Traversing
    - Accessing
    - Staying within array bounds
- Work with parallel arrays
- Validation of data
    - Using stream states
    - Using other more traditional means (that you've probably done before)

Overview

The Paris 2024 Olympics will hold their track and field events at the Stade de France. The track has 9 lanes. We have been asked to determine the results of heats using C++ parallel **arrays (not vectors)**.

Example Run of the Track Result

```
[1]  32.70 Moore              (USA)  +0.00
[2]  33.40 Munson             (TKY)  +0.70
[3]  36.50 Polsley            (RUS)  +3.80
[4]  38.00 Reardon            (ARG)  +5.30
[5]  45.80 Taele              (ENG)  +13.10
[6]  50.10 Darlington         (ICE)  +17.40
[7]  52.34 Nemec              (CHN)  +19.64
[8]  60.34 Da Silva           (NIC)  +27.64
[9]  76.45 Lupoli             (ITY)  +43.75
```

The output shown above is formatted and the function for that display is given. You will not have to make any edits or changes. The data will be stored in a text file and will look like below.

```
32.7 USA 12 Moore
36.5 RUS 35 Polsley
45.8 ENG 73 Taele
52.34 CHN 14 Nemec
76.45 ITY 23 Lupoli
33.4 TKY 82 Munson
38.0 ARG 88 Reardon
50.1 ICE 41 Darlington
60.34 NIC 50 Da Silva
```

Notice the data will come in this form and order on each line:

- Time Completed
- Country
- Jersey number
- Last name

Several sample files are available with the start code.

Your program will pull data from the text file and place each piece of data into separate but parallel arrays:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
time	32.7	36.5	45.8	52.34	76.45	33.4	38	50.1	60.34
country	USA	RUS	ENG	CHN	ITY	TKY	ARG	ICE	NIC
number	12	35	73	14	23	82	88	41	50
lastname	Moore	Polsley	Taele	Nemec	Lupoli	Munson	Reardon	Darlington	Da Silva

Using the data in the parallel arrays, you will rank each person based on their time from low to high.

- *There is absolutely **NO ORDERING** or **SORTING** of the data.*
- *You put each person's rank into a parallel array which will be used to print the results in the correct order.*

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
time	32.7	36.5	45.8	52.34	76.45	33.4	38	50.1	60.34
country	USA	RUS	ENG	CHN	ITY	TKY	ARG	ICE	NIC
number	12	35	73	14	23	82	88	41	50
lastname	Moore	Polsley	Taele	Nemec	Lupoli	Munson	Reardon	Darlington	Da Silva
rank	**1**	**3**	**5**	**7**	**9**	**2**	**4**	**6**	**8**

Think of the rank function as finding the lowest number first, marking the appropriate rank array index with 1, then continuing to find the next value as long as the rank is not already determined. It will need to check the rank position to see if it has been marked and not consider those.

Roadmap

This is a guide on how to tackle doing this homework. Specific requirements for functions are in the Requirements section below.

1. Get the starter code
    a. Code Files
        - `main.cpp`
        - `parallel_tracks.h` (you should read this file)
        - `parallel_tracks.cpp`
    b. Test Files
        - Various text files to help you test your code (.txt)
    c. **Read the header file!** It gives details about each function you will implement. The source files (.cpp files) have comments marked // TODO where you will need to write code.
2. Review the code.
    a. Compile and run it before making any changes.
        - It won't do anything but it also won't crash.
            - So **all** errors were introduced by you!
    b. Read the header file.
        - You are required to implement all functions listed in parallel_tracks.h except for the print_results and trim functions.
        - Read the comments for more information.
        - DO NOT MAKE CHANGES TO print_functions or trim!
        - It gives details about each function you will implement.
        - The source files (.cpp files) have comments marked // TODO where you will need to write code.
3. Implement functions
    a. **Write your code so that it is easy to understand!**
        - Conventions like meaningful variable names and commenting will make it easier for you to understand your own code and implement your algorithms!
        - Use descriptive (long) naming conventions for variables and functions.
        - **Add comments** to the code to describe anything which is not obvious from the code.
        - Use whitespace (indentation, newlines) to visually organize code.

        - Use functions to reduce code duplication and increase abstraction.

    b. Writing all of your code and then going back to debug can double, triple, quadruple or more the time it takes you to implement your solution.

    c. You should attempt to implement functions based on their dependencies.

        - Since get_runner_data depends on having initialized arrays, you should do all of the **prep_*_array** functions before doing get_runner_data.

            - This can be done before or after updating the

        - Since the get_ranking function relies on arrays populated with data, you should do **get_runner_data** next.

        - Finally, you should do **get_ranking** next since print_results depends on it.

    d. Recompile and rerun after completing each function.

        - Note: You can test and debug before you add all functionality to a function. Get one aspect to work and then add functionality in measured increments until you complete the full functionality.

        - Check for errors.

        - If no errors, move on

        - Else, start debugging

    e. Once you have some functionality, submit to Gradescope.

        - If the basic tests for that function pass, move on

        - Else, start debugging

    f. Continue by picking new tests and writing just enough code to pass them, adding more functionality each time.

Requirements

Allowed Includes

- `<iostream>`
- `<string>`
- `<sstream>`
- `<fstream>`
- `<iomanip>`
- `<stdexcept>`
- `<limits>`

- `<cctype>`

main function

- Create arrays for time, country, jersey number, name and rank.
    - They should use the SIZE constant in parallel_tracks.h for their size.
- The program should prompt the user for the name of a valid file (does not throw any of the above exceptions) using “Enter file name: “
    - You can do this before dealing with exceptions so you can test your program.
- Exceptions
    - The program should catch all of the invalid_argument and domain_error exceptions and continually re-prompt until a file name is provided that is ~~not~~ valid.
    - All exceptions (for invalid data, missing data, and bad files) should be caught in the main function and the error message printed out, preceded by the message “Invalid File: “.
    - This might be easier to do, once you’ve written enough code to be able to throw exceptions.

Prep functions

These functions initialize all of the values in each array before they are used.

1. `prep_double_array` elements should all be set to 0.0
2. `prep_unsigned_int_array` elements should all be set to 0
3. `prep_string_array` elements should all be set to “N/A”

get_runner_data

This function loads data from a text file into the parallel arrays. You can only process data from a file that has valid data. If you have a problem with the data, you will throw the appropriate type of exception.

- When reading from the file, it is recommended to read an entire line into a stringstream and process the individual values from there.

File validation:

- If the file fails to open, throw an invalid_argument exception with message: “Cannot open file”
    - Note: If opening the file fails, is_open will be false.

Data validation:

- If there is no data on a line where input is expected, throw an domain_error exception with message: “File missing data”
    - If you use getline and the resulting string is empty.
    - Do not throw an exception if it is an empty last line.

    - You should trim whitespace before checking for valid characters.
- All of the following should throw an domain_error exception with message: “File contains invalid data” followed by which piece of data is invalid {time, country, number, name}:
    - The exception thrown should correspond to the **first** piece of invalid data in the file (reading left to right, top to bottom).
    - For example, if the time is -15.01, an exception with the message “File contains invalid data (time)” should be thrown.
- The double containing **time** information:
    - Be a valid floating point number
        - Stream states can help
    - Be a non-zero positive number.
- The string containing **country** information:
    - Contains only capital letters ‘A’ - ‘Z’
    - Contains exactly 3 characters
        - Note that an empty string is not valid
- The unsigned int **number** information:
    - Be a valid unsigned integer
        - Stream states can help
    - Contains 1 or 2 digits
- The string containing **name** information:
    - Contains only alphabet characters ‘A’ - ‘Z’, ‘a’ - ‘z’ and ‘ ‘ (space)
        - You should trim whitespace from the beginning and end before checking for valid characters.
    - Contains more than one character
        - Note that an empty string is not valid
        - You should trim whitespace from the beginning and end before checking if it is long enough.

get_ranking

This function looks at the time in each element of the parallel array. The corresponding element in the rank array is assigned the appropriate rank. The lowest time’s rank is one and the highest times rank is 9.

Useful Functions

- **`getline(istream, string, char delim)`** - extracts characters from the input stream and stores them in the string until the char parameter delim is reached (delim is extracted and discarded)
- **`>>`** gets a value and stops when it gets to whitespace (i.e. a space, at tab, and return, etc.)
- **`isupper(char)`**
- **`isalpha(char)`**
- **`isspace(char)`**

Submission

The source files to submit to Gradescope are named:

1. `main.cpp`
2. `parallel_tracks.h`
3. `parallel_tracks.cpp`