Lab 4 – File I/O

Overview

For this assignment, you are going to load a series of files containing data and search the loaded data for specific criteria. That might sound a bit dry, but the files contain information about LEGO sets, and everyone loves LEGO! The structure of this assignment is a bit open-ended; you can solve this problem in any way that you see fit.

Description

First things first, the files: There are **3** main data files that you will be loading in this assignment:

- lego1.csv
- lego2.csv
- lego3.csv

The data that you will be loading is information about a LEGO set:

1. Its set number (really a string, something like: 10195-1)
2. The theme it comes from (City, Technic, Star Wars, etc.)
3. The name of the set
4. How many parts and minifigures it contains (if any)
5. Its price in US dollars – for this variable, use a **double** instead of a float

Your goal is to read this from 1 of or more of these files, store the data, and then search it based on a few different criteria.

Main.cpp is the only file required for this assignment, but you are free to write any class/functions that you see fit to assist you solve this problem. Main.cpp has some structure to it already to help you get started. Take some time to **think** about the problem, and how you might go about this before diving in.

Searches

The different searches you will perform will be based on a menu that might look like this:

1. Most expensive
2. Largest piece count
3. Search for set name containing…
4. Search themes...
5. Part count information
6. Price information
7. Minifigure information
8. If you bought one of everything...

Most Expensive From the sets that were loaded, which is the most expensive?	The most expensive set is: Name: Super Awesome Building Set Number: 99923 Theme: City

	Price: $21.99 Minifigures: 4 Piece count: 286
Largest piece count From the sets that were loaded, which has the most parts?	The set with the highest parts count: Name: Really Big Set Number: 22231 Theme: Technic Price: $249.99 Minifigures: 0 Piece count: 5211
Search for set names containing… Get a string as input from the user. Then search all sets and their names to see if they contain the search term. There could be a lot of sets matching the search term, so show them in a more concise, list format with the set ID, name, and price. If no sets are found, report that as well.	Sets matching "Fire Station": 49281 Fire Station $19.99 9381 Big Fire Station $49.99 **-OR-** No sets found matching that search term
Search for set themes containing… Ditto[1], but for the theme of the set instead	Sets matching "City": 1234 Police Station $29.99 49281 Fire Station $19.99 // And TONS more… LEGO City is huge!
Part count information Show the average parts for all the loaded sets	Average part count for 601 sets: 492
Price information Show the average, minimum and maximum prices.	Average price information for 601 sets: $500 Set with the minimum price: [Set data goes here] Set with the maximum price: [Set data goes here]
Minifigure information Show the average number of minifigures, and the set information for the set with the most minifigures	Average number of minifigures: 8 Set with the most minifigures: Name: PG2 Lecture Number: COP3503 Theme: Education Price: $299.99 Minifigures: 310 Piece count: 938
If you bought one of everything…	If you bought one of everything...

[1] A similar thing; a duplicate

How much would it costs? How many parts and minifigures would you have?	It would cost: $9999.99 You would have 200207 pieces in your collection You would have an army of 3000 minifigures!

Reading Mixed Data From Files

When reading a text file, oftentimes the data initially gets read in as a string. If the final storage variable isn't a string (such as a person's age, or the price of something), you must convert it before storing/using it in its numeric form. In the **<string>** header file, there are a number of functions to help you convert. These function converts a STRING_TO_SOMETHING, and are named like stoi (string to integer), stof (string to float), etc.

Example:

```
ifstream someFile("Example.txt");
string someString;
someFile >> someString;

// We COULD just read directly into the int... but sometimes data doesn't start out that way.
// If you read a lot of data (say an entire line from a file), then you may break
// that one string into many smaller strings, and convert as necessary.
int convertToInt = stoi(someString);
```

The implementation of some of these functions may throw an exception of type "invalid_argument" if the conversion process fails, so you may want to encapsulate these operations in try/catch blocks, and use a default value if you catch an exception—if the number can't be converted, (in this case) it's because a value wasn't there, so what would be a good value to use in the absence of anything else? Refer back to the section on Exceptions in your textbook if you need to.

Tips

1. Choices you make at the start of a program can have a big impact on how the rest of the program gets developed. Think about how you want to store the information retrieved from the file, and how you could easily pass that data to various functions you might write.
2. If you have a process for easily loading and accessing the data, the rest of the functionality should be a lot easier to write. Make sure the loading process is all taken care of before worrying about anything else.
3. The code to load 1 file containing 1 piece of data (no matter how complex that data is) should not be much different than loading 100 files, each containing 100 elements. Start by thinking about just 1 element from the file first. Do the values you read match the values in the file? What about 2 entries, does everything add up? Then 3, 4, etc.
4. When passing containers of data, make sure you pass them by **REFERENCE**, not by value. Creating copies of massive data sets is generally a bad, bad thing… unless you specifically need to duplicate the data. If not? Then pass by reference (in C++ that means either by pointer or the reference data type)
5. There may be a fair amount of repetition in a program like this. Think about where you can create helper functions to reduce the number of times you write the exact same (or just slightly different) code.

Sample Outputs

```
Which file(s) to open?
1. lego1.csv
2. lego2.csv
3. lego3.csv
4. All 3 files
Total number of sets: 767

The most expensive set is:
Name: Death Star
Number: 75159
Theme: Star Wars
Price: $499.99
Minifigures: 27
Piece count: 4016
```

```
Which file(s) to open?
1. lego1.csv
2. lego2.csv
3. lego3.csv
4. All 3 files
Total number of sets: 710

The set with the highest parts count:
Name: Surf N' Sail Camper
Number: 6351
Theme: Town
Price: $19.00
Minifigures: 2
Piece count: 188
```

```
Sets matching "Police":
71021 Classic Police Officer $3.99
75046 Coruscant Police Gunship $49.99
2811 Fire and Police Station $50.00
30311 Swamp Police Helicopter $3.49
6533 Police 4 x 4 $4.75
30282 Super Secret Police Enforcer  $4.99
7236 Police Car $5.99
6684 Police Patrol Squad $6.50
60006 Police ATV $6.99
7279 Police Minifigure Collection $9.99
10720 Police Helicopter Chase $9.99
```

```
Average part count for 971 sets: 550
```

```
Average price for 710 sets: $18.04

Least expensive set:
Name: Swamp Police Helicopter
Number: 30311
Theme: City
Price: $3.49
Minifigures: 1
Piece count: 51

Most expensive set:
Name: Tech Machines
Number: 9203
Theme: Dacta
Price: $105.00
Minifigures: 6
Piece count: 117
```