3460:209 Project Part 2

# **Project Assignment Part 2: My Airline**

**Overview**

The purpose of this part of the project is to make sure that you know how to write a program that utilizes functional programming, and utilizes files and file processing as well as random number generation. This part of the project’s task list is exclusive to what we are learning in Unit 2. Do not use advanced techniques, for example, do NOT use arrays. Any use of anything beyond Unit 2 will invalidate your program and it will not receive consideration ☹

**PROGRAM SPECIFICATION**

Your airline has been getting more and more good press, and has drawn the attention from travel experts as a great option for travelers. We continue to improve our software as the business grows. We will again pick up from the last part, and add more functionality.

The next part of our work with the airline system will be to functionally decompose what we have so far, and perform refactoring. Basically, the parts of our program that are all within main will now need to be “broken out” into separate functions. Reconstruct all the logic you have created thus far into functions. Think about the number of things your program manages, and design them to be functions. For example, the menu should now be considered as a separate process and placed into a function, and so on. Consider how many functions there are. Consider reuse. Don’t forget to add your pre and post conditions. Make certain to use a call-by-reference when you can. Make sure to consider leaf functions only, if possible.

We also will be creating brand new functions from the things that we’re building into this release. A well-considered effort here will result in a very streamlined main function.

A list of possible candidates for functions may include the following:

* Menu
* Item Entries with validations
  + Type of flight
  + Route
  + Wind Speed
* Randomizer generating flight numbers
* Track of entries (static)
* The transaction log
* Flight Calculator and displaying (could all be in one function)
  + Flight times
  + Flight time category for this trip (such as short-haul, etc…)

Some of these could possibly be further subdivided. Notice we added a few functions too. Those functions are described by our specifications that follow.

Along with the decomposition effort, we will modify part of the existing menu logic to generate a unique value for the flight numbers. This flight number will be of the type int, and will be 5 digits in length. The values are 00001 – 99999, and should appear with *all 5 digits* (leading zeroes) on the display upon generating them. Do not change the data type from that of an int. Check out the setfill manipulator as a possible helper for this. Setfill allows you to set a fill character to a value using an input parameter, such as ‘0’ for leading zeroes. Researching it, you’ll see that you will need to reset the fill character after use too, or you will be printing a lot of zeroes on the display! Your program also should guarantee an absolutely unique flight number has been generated for each new activity that is added. Think about how this works…

Additional messaging should now appear on the output screen with the new flight number, and that should remain on subsequent displays until another new flight is requested (i.e., your program cycles back to the beginning of the menu). You can place this randomly generated flight number on the screen wherever you like, such as top right hand corner, etc…

Add to your menu logic the use of a static variable to keep count of how many times that the program’s main menu has been invoked. For example, after we have added 2 turbo prop flights, and 1 heavy jet flight, the number of this static variable should be 4 (this is the 4th time we have invoked the menu), and your program will display “Number of hits today is 4” on the display (again, where is your design choice). You will create a separate function just for this task. See the sample display of what your program should be doing

Next, we are going to add file processing. You are going to add an output file that will journal entries that are made by the user. The format of the file reflects the information that you will gather, and should be as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| User ID | Type of flight | Minutes | Route | Flight Time | Status | Time in |
| 99999 | 1 - 4   1. Helicopter 2. Turbo Prop 3. Midsize Jet 4. Heavy Jet | 0-60 | CLE-ORD  ORD-JFK  DEN-JFK  JFK-SFO JFK-MIA DFW-CYYZ | Hours and minutes | One of the four levels | Grab the system time |

Notice that the system’s time will be captured and written to the log. Each and every time that your program runs, this file will be *appended* to, and each time the system time will be refreshed to current before adding. Consider how best to store the data on your file, such as the status. The name of the log file is totally up to you, but should be a .txt file. You should create a separate function(s) to handle all of this. Be sure to write *only* when the user has successfully completed a transaction. You should insert the following code examples for getting the system time into your program for this part of the processing (note we must #include ctime, which you may already have from randomizing):

// current date/time based on current system

time\_t now = time(0);

// convert now to c-string form

char\* timeOf = ctime(&now);

// we want a way to limit the size to be just 20 in length

timeOf[20] = '\0'; // this effectively truncates the c-string

This code works and can be used directly in your program. The time\_t type, the conversion to a c-string, and the idea of truncating the c-string are advanced topics. If you wish to research how this works, please do. Otherwise, we will cover c-strings in several weeks. Basically, after inserting this code in the proper place, you will use timeIn as another one of the variables to stream to your output file.

Consider opening your transaction file using the append mode. The ios::app mode causes data to be written to the end of the file. This will allow you to add to it as you go along, each time someone uses your app. Consider opening your file in main(), and then passing it as a best practice. You might also consider using the fstream type too, because we will need to print from it, thus making it both an output and an input file. I have included an example of the coding for that here:

**transactionLog.open("tranlog.txt", std::ios::in | std::ios::out | std::ios::app);**

Once you have figured out your function to write out records to the transaction log, you will need to also design a function that will print this file too. This is a bit tricky, however, you can do this by using some more free code here that essentially will take us to the beginning of the file so that we can print the all of the items sequentially. We need to do this because if we have had users entering new flights then it is probable that the file will be positioned at the end, ready to write more. So we need to go to the top. Here it is:

**// reset everything and go to the beginning of the file**

**transactionLog.clear();**

**transactionLog.seekg(0, std::ios::beg);**

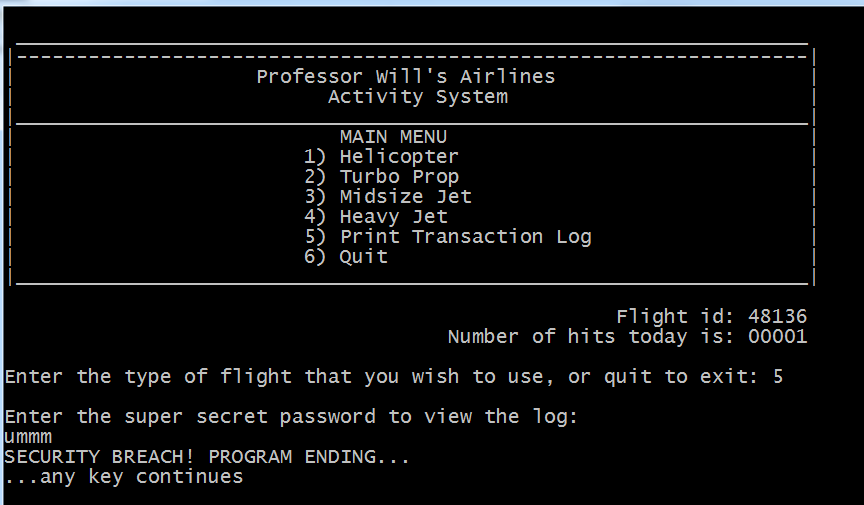
**// start the streaming >>**

File stream objects have two member functions that are used to move the read/write position to any byte in the file. They are seekp and seekg. Seekg works here for us. Notice it uses two arguments. The first one is a numeric value that represents an offset. The second argument is a mode flag. We used beg, for beginning, which is the beginning of the file. The statement basically says use the transactionLog and move to byte 1 (no surprise here but the counting of bytes starts at zero) from the beginning (beg). The seekp and seekg fall under more advanced file operations as random access methods. Feel free to study them more on your own.

One more thing… in order to reengage this file *after* processing through it for the display, the file needs to be repositioned to the end. This could be accomplished in a few ways, but close and reopen would be inefficient. If this seems less confusing then you may close it and reopen it (recommend doing this after the function call back in main). This certainly aligns with our level of understanding of files and file processing. However, there is a different method that you can use and it is going to be very simple and fast. In fact, you already used it to move the file position to the beginning. Figure it out?

The next part adds to your menu logic the ability for the file to be printed. Thus, you will add an option to the menu as “Print Transaction Log”. This will assume the option 5, and option 6 will now be added to support “End”. Don’t forget to change all parts of your program, such as validation, to accommodate this change. When the request is made to print this journal or log, you will also add logic (yet another function!) that prompts the user for a password. This password should be of the data type **string** and should equal (case sensitive) the word “Not-telling”. If the user fails the request then your program should abort, issuing a message “SECURITY BREACH” upon exit. Use the string compare operator == to check this value. Notice you can stream (>>) the input because there are no spaces… (i.e., it is not necessary to use **getline**). Importantly, when a violation is encountered consider the best way to *gracefully* end your program for full credit.

If the user enters the correct password, your function that will print the file should be invoked and it should have a heading and appropriately formatted output. The design of this display is totally discretionary, however, it should be neat and orderly and most certainly should make use of the iomanip formatting tools. An example of what your program should look like follows (new menu, number of hits, random flight number, and this example shows the print option with a security breach):



Make sure that your programs follow good documentation standards and follow the requirements for assignments. Reference the rubric standards on Brightspace. Validate data. Format output. Check calculations. Test, test, and test again before submitting.

Submission Instructions – for programming solutions

On Brightspace, go to the matching Assignments for the **PROJECT PART-#**, where # is the number or character of the project part assigned (eg., 5 for project part 5), and submit your cpp, and any hpp files (if they exist). Unless otherwise specified, they may be submitted under any name that you prefer (such as main.cpp).

*Last updated 5.22.2017 by Will Crissey.*

*Be aware that programming falls under all of the rules of plagiarism. Be careful when using any coding found in the outside world that is not your own. Any evidence of plagiarism is subject to sanctions like forfeits, suspension, and even ejection, as determined by the Department of Student Conduct and Community Standards.*

**COPYRIGHT © 2017 Will Crissey, Jr., Assistant Professor, All rights reserved.**

This document is intended exclusively for use by The University of Akron's Department of Computer Science. Do not distribute document copies to the public in any way.

***THIS PROGRAM'S CONTENT IS PROTECTED***

The document and associated content are either the property of, or used with permission by, The University of Akron, Department of Computer Science and may be protected by copyright and other restrictions. Copyrights in the site content may also be owned by individuals and entities other than, and in addition to, The University of Akron, Department of Computer Science. The use of this document by you, or anyone else authorized by you, is prohibited, except for purposes of completing the homework/project. Any unauthorized use of the document may violate copyright laws, trademark laws, the laws of privacy and publicity, and communications regulations and statutes.