Assignment 1: Official Airline Guide

This assignment was developed and written by Julie Zelenski (zelenski@cs).

The **Official Airline Guide** (OAG) is a thick book of all airline operating schedules used by travel agents and corporate travel departments to figure out how to get from A to B and back. The rainforest denuding and spotted owl mulching which has been caused by the resource-hungry yearly printing of the OAG has prompted you to write a paperless on-line version using your newest and most favorite language: C++.

The goal of this assignment is to refresh your memory of C, making sure you remember the details of dynamic allocation, pointers, etc., and give you some exposure to the new syntax and features of C++. You will make use of C++ additions such as `const`, reference parameters, `iostreams`, `new/delete`, etc. as well as getting a chance to start working with objects. We give you a few pre-defined objects (our `SymbolTable` and `FlightList` classes), and you will write a few simple objects of your own (the `Airport`, `Flight`, and `Trip` classes) to coordinate with them.

Due Monday, October 16 at 11:59 p.m.

A Brief Overview

You begin by reading and processing the airport and flight information. We supply text data files containing the list of cities and airport codes and all the known flights between airports, with facts such as arrival and departure time and flight number. You will create `Airport` and `Flight` objects to encapsulate the data you read and wire up a graph where each flight points to its origin and destination airports and each airport keeps a list of flight pointers which depart from there, making it easy to go back and forth between the two and traverse the flight paths.

Once you have built the data structure, you enter an interactive mode where you prompt the user for a departure and destination city and then find all the reasonable sequences of flights that one could take (with connecting flights as well). This will involve a recursive depth-first traversal of the graph starting from the origin city and searching outwards until you find the goal, hit a cycle, or give up because the path is too long. You will print each the potential paths as you find them for the user's inspection. You will loop, generating more and more itinieraries, until the user quits.

The Data Files

A database of flight schedules comes in two files: the `airports` file lists all the airports in the database. Each airport is described by its 3 character code on one line followed by its long city name. Each airport code is exactly three letters, so you can use `char[4]` to hold an airport code. While reading the city names, you can assume the long

name will never be more than 64 characters, but once read, you should store the city name without wasting extra space. You can assume the data file is well-formed—this is, you do not have to do any error-checking.

```
SFO
San Francisco, CA
BOS
Boston, MA
SJO
San Jose, CA
<end-of-file>
```

The `flights` file contains information on all the flights which connect our cities. Each line will consist of the codes for the departure and destination airports, the departure and arrival times, and the flight number. The times can be read as integers. The flight number will contain at most 5 characters.

```
SFO BOS 1330 2145 UA94
SFO BOS 2200 0759 TW44
DEN BOS 1055 1633 CO314
SFO DEN 0700 1015 CO462
SJO SFO 0533 0559 TW124
SJO SFO 1215 1235 ID753
SJO SFO 2045 2110 OE972
<end-of-file>
```

The Data Structures

To give you some practice with objects, you will write a few classes of your own to mange the above data. The `Airport` and `Flight` classes are not much more sophisticated than C `structs`, but they serve to encapsulate the data into clean abstract packages. These classes consist of `private` data and `public` accessors to retrieve various data members, but not much behavior beyond that.

The `Flight` and `Airport` classes are designed to work together. `Flights` should track which `Airport` they go to and from (via pointers) and the `Airport` will track all of the `Flights` that depart from there (using a `FlightList` of `Flight` pointers). You should impose no upper bound on the number of flights that might leave from a particular airport. To make this easier on you, we've provided a simple `FlightList` object that will manage an unbounded collection of `Flight` pointers. This class is a bit odd in that you wouldn't likely create a collection data structure that is so specialized, but we won't get to the facilities in C++ for creating generic data structures for another few weeks yet.

We have also provided a `SymbolTable` class that can manage a key-value collection. This can be used to store an arbitrary pointer value under a string key. For this program, you will store the airport pointers into the symbol table under their shorthand airport code. This will facilitate quick lookup of airport by code when

connecting the flights up to their airports. You will also use this table later to look up origin and destination cities for the user.

When your data structure is complete, it should be possible to lookup any airport by code in the table, examine the flights that leave from that airport, follow those flights to their destination airport, and so on.

Finding Flights

In the interactive phase, the user enters the origin and destination cities (using the airport code) and your job is to find all the routes between them. It may be possible to get from the origin to the destination on one flight. There will also probably be ways that involve one or more connections at intermediate cities. So you get humane suggestions from the program, paths will be restricted as follows:

- No one would want to take a trip with more than 4 flights in its path, so you do not need to consider any paths longer than that. This number should be a constant so it can be easily changed later.

- A path should not contain any loops, i.e. a path should not visit the same airport twice.

- To avoid long waits on connections, the difference between the arrival time of one connection and the departure time of the next should never be more than 2 hours. In order to avoid rushing, so the departure flight should leave no earlier than 30 minutes after the arrival of the first flight. These two numbers should also be constants that can be easily changed. Be careful in handling times around midnight— 23:55 and 00:05 are ten minutes apart.

You will use a recursive depth-first search from the origin city and trace flights outward until you either hit the destination, end up in a loop, or have made 4 hops without arriving at you destination. After you have computed a path, print the trip information in a reasonable format. Paths should be printed as you find them (i.e. you don't need to store them or print them in any particular order)

```
Choose the origin airport ("quit" to quit): SFO
Choose the destination airport: DFT
That airport not known.  Please try again.
Choose the destination airport: DCA

From: SFO San Francisco, CA
To:   DCA Washington, DC

Dep   Arr   Flight (Connections)
-------------------------------------

21:02 07:58 UA945/UA066 (LAX)
21:02 10:00 UA945/TW072/TS120 (LAX,STL)
22:00 23:30 TW044/AA011/PA818/RZ739 (BOS,LAX,JFK)
15:01 07:58 FL264/CO317/UA066 (DEN,LAX)
22:00 18:20 UA024/NW003/CO385/PL800 (JFK,IAD,DEN)
09:55 20:10 TW122/TW026 (STL)
11:50 21:05 RC344
```

The first line of the above itinerary list means it's possible to go from San Francisco to DC with just one connection: UA945 flies from SFO to LAX; UA066 flies from LAX to DCA. Our unlucky traveler would leave San Francisco at 9:02 p.m. and arrive in DC at 7:58 a.m. Alternatively, there is one non-stop flight (RC344) and a variety of other multi-leg paths to choose from.

One last object you will create is the `Trip` class, which encapsulates the data for a trip including all of the flight legs. A `Trip` pretty much just contains only a `FlightList` to track the list of flights in the path. This is largely a convenience object to help you manage partial itineraries while doing the search. By including methods that add and remove connections, check if a new flight can connect to the trip so far, print the trip out, etc. you will make the search code cleaner and easier instead of cluttering it with `FlightList` details.

Some design guidelines

We haven't had much of a chance to get into some of the issues in effective use of C++ features and class design, but here's some basic information about our expectations about your programs:

- All class data members (instance variables) should be `private`.
- If a client needs access to a data member, you can provide a public accessor function (a "getter"). However, be very wary about handing out pointers or references into your internal data—i.e. a stack object shouldn't hand out an internal list and have the client add elements to it, but instead provide a `push` method which takes the client's element and adds it to the internal list.
- You should only provide a setter function for a data member if the client's use will require it and there is no better way to provide that functionality. Take precautions in the setter function so that it cannot be used maliciously

to corrupt the internal consistency of your object. (i.e. don't allow a client to change a count to a negative number or such things.)

- Most methods will be `public`, since they are usually intended for public use. However, any helper methods only for use of the class implementor should be `private`.
- Try to give responsibility for behavior to the object itself rather than manipulating it using setters/getters from the outside. For example, you want to include methods in the class that can read its data from a file or print the data out nicely rather than extracting the data and printing it externally.
- At this point, we haven't yet learned about friend access, so we would prefer that you not use it. There are some problems this "back-door" can help solve, but, in general, the best abstractions are those that maintain independence from others.
- Use references to pass parameters who need to be modified in the calling function, as opposed to the C way of passing by explicit address/pointer. By convention, all objects are passed by reference (or `const` reference to achieve the performance benefit while indicating the parameter will not be changed). In fact, our `SymbolTable` and `FlightList` (as well as the `iostream` classes) cannot be passed by value, and thus must always be passed by reference.
- Use `const` instead of `#define` for defining constants. Consider defining the constant in the smallest necessary scope (i.e. if only used in one function, you can limit the scope of the constant to just that one function). Get used to using `const` to mark parameters and variables that will not be modified.
- Use the C++ `new` and `delete` memory management operators instead of C's `malloc` and `free` for all dynamic storage needs.
- Use C++ I/O facilities (`iostreams`) instead of C's `printf` and `scanf` functions. We will touch on streams in lecture and section, but mostly you will be responsible for reading up on this (in the Deitel text or Eckel or wherever) to get all the myriad details. File input and output is one of the less interesting language features— every language does I/O, each is different in annoying ways, and there are always millions of little details to absorb. I, for one, favor the approach of looking up the details on a need-to-know basis.
- The standard C `assert` macro can be used to detect and report exception conditions (out of memory, file not found, etc.) This macro takes one argument, an expression which is evaluated and if `true` (non-zero) execution continues on, but if `false` (zero) results, it will print an error and terminate the program. The idea is to `assert` what must be true before carrying out an operation that depends on those assumptions. Getting in the habit of programming defensively from the beginning is an excellent idea. The few seconds it takes to put in an assert statement can save you hours of debugging time. Further on in the quarter, we will learn about C++ exceptions, an even better facility for handling errors.

- You'll note that object decomposition leads to a different sort of code structuring than you're used to. For example, in C, you would likely group all the file-reading functions into one unit. In C++, each object takes responsibility for reading its own data, which will have the effect of distributing the code for file reading around various classes. This can be a little disconcerting. Although object decomposition is an effective tool for managing complex projects, this sort of consequence is one of the downsides to it.
- There will be some code that doesn't fit into an object (most notably the main code that kicks off the file-reading or handles the interaction with the user). Don't be concerned about this, one of the benefits of C++ is that you can use straight C where appropriate. Just write normal C functions in a good readable style and organize them sensibly.
- Oddly enough, there is no standard on the extension used to identify C++ files. The original convention was to use `.C` (as opposed to `.c` for standard C files), but that fails miserably on systems that aren't case sensitive. Since then, .cxx, .cpp, .cp, .cc and others have been proposed, with no clear victor standing out. We're going to use .cc, since that's what the `make` utility seems to prefer.
- Beware of fussy compilers. The C++ compiler is much pickier about code than an ordinary C compiler. Be prepared for it to get indignant about things that used to be considered harmless—slightly mismatched types, functions used without declarations, conversions between `enums` and `ints`, use of `const`, etc. We expect your code to compile cleanly, without warnings.

Getting started

Create a local working directory in your leland space where you'd like to consolidate all of your Assignment 1 files. When ready, type the following at the command prompt:

```
elaine20> cp -r /usr/class/cs193d/assignments/hw1 .      // note the dot!
```

and all of the assignment 1 starter files will be copied verbatim the current directory. In particular, you will see a `Makefile`, several header files, and several source files—all of which will contribute to your program development efforts.

If you choose to do your development elsewhere, that's totally fine with me, but understand that it's your responsibility to move the starter files to your own environment and then move them back to your leland space when you're done. No matter where you do your own development work, when done, you must be sure your project will compile and run on the `leland` workstations. All assignments will be electronically submitted there and that is where we will compile and test your project. This means you will need to deal with the `make` files that you normally wouldn't need to touch. While C++ has been normalized via an ANSI specification, different compilers interpret the specificaion differently, so what compiles cleanly on

one platform may compile with marning or even errors on another. Be sure to allot time for the porting process; porting issues are not a legitimate gripe for getting an extension or a pardon from the use of a late day.

Electronic submission

Our goal is an entirely paperless class, so you will not submit printouts, but instead use a `submit` script to electronically deliver your entire project directory to us. Before submitting it, make sure your project directory contains all your source files, the `Makefile`, and a short `README` file. Here are the steps to follow:

1. Log into your leland account using one of the elaines.

2. Remove any unnecessary files from your project directory such as object files, binaries, etc. If you forget, the submission script will not let you submit. Include only source code, `Makefile`, `README`, and any other files which the assignment specifically needs.

3. Make very sure that the project you submit compiles properly by just issuing the command make. If it doesn't compile, we won't be able to grade your submission. We'll also frown on code that does not compile cleanly (i.e. produces warnings == bad) — your project should make with no complaints.

4. Put a `README` file in your project directory, containing your real name, your leland username, and an e-mail address where we can contact you if we have any problems. If you have any special comments you want the grader to know about your submission, please add them here.

5. Type the command: `/usr/class/cs193d/bin/submit`.

6. The script is going to ask you for your full name, which assignment you are submitting, and the path to the directory containing your project. Once the script verifies that the project is properly cleaned, has a `Makefile` and a `README`, it submits your project for grading. The easiest way to use the submit script is to first cd to your project directory and then specify the path as just .(dot) which indicates the project is in the current directory.

7. If all goes well, you get the happy "SUBMIT SUCCESS" message. If you get a message that says something else, follow its directions. If you get stuck, send mail to cs193d@cs.

8. If you need to resubmit something, just redo these directions. The script will notice if you are attempting to re-submit and allow you to overwrite. Only your most recent submission will be graded.

The assignment is due Monday evening and the absolute cut-off is midnight. The electronic script has no mercy for things submitted even just 10 seconds past midnight, so be sure to give yourself enough leeway to work through the e-submit process. Cutting it too close may lose you an entire late day!