Flights

Deadline: July 5th 11:59:59 PM

Goals

This project aims to get you more familiar with C, especially C memory management.

Getting Started

Please accept the github classroom assignment by clicking this link. Once you've created your repository, you'll need to clone it to your instructional account. You'll also need to add the starter code

```
$ git remote add staff https://github.com/61c-teach/su19-proj1-starter
$ git pull staff master
```

If we publish changes to the starter code, you may get them by running `git pull staff master` Please read the entire spec before beginning the project.

For this project we recommend you work entirely on the hive machines. While your code should work on a machine that is not a Linux machine, you will be graded on Linux machines and some of the debugging tools in this class are only available for Linux.

All the tools you will need for this class are already installed on the hive. If you want to connnect to the hive you can do so via `ssh`, while connected to airbears2 if you are on campus and through nearly all wireless networks off campus. The only real restriction is that CalVistor **DOES NOT** allow you to connect via ssh.

C and Memory Allocation

You will be completing the implementation of `flights_structs.h` and `flights.c`, a flight system that keeps track of the flights between a series of airports. The flight system, represented by the struct `flightSys_t`, will hold all the airports in this system. Each airport, represented by the struct `airport_t`, will hold both its name (as a string) and a schedule of all the flights departing from it. Each entry in the schedule should contain:

- a pointer to the destination airport,
- time of departure,
- time of arrival,
- and the cost of the flight.

These will be the contents of your `flight_t` struct. We have provided a program, `RouteTime.c`, which will both provide the data to your flight system and use the data you store to figure out the cost of flying via a certain route. You do not need to know how `RouteTime.c` works to complete the assignment. We have also provided you a struct, `timeHM_t`, defined in `timeHM.h` that is used to represent time in hours and minutes. It also contains several useful functions.

For this assignment you should only modify `flights.c` and `flight_structs.h`. A skeleton has been provided for you, but you will need to define the structs `flightSys_t`, `flight_t`, and `airport_t` and implement the following functions:

- `flightSys_t* createSystem (void)`
- `flight_t* createFlight(airport_t* destination, timeHM_t* departure, timeHM_t* arrival, int cost)`
- `void deleteSystem(flightSys_t* system)`
- `void deleteFlight(flight_t* flight)`
- `void addAirport(flightSys_t* system, char* name)`
- `airport_t* getAirport(flightSys_t* system, char* name)`
- `void printAirports(flightSys_t* system)`

- `void addFlight(airport_t* source, airport_t* destination, timeHM_t* departure, timeHM_t* arrival, int cost)`
- `void printSchedule(airport_t* airport)`
- `bool getNextFlight(airport_t* source, airport_t* destination, timeHM_t* now, timeHM_t* departure, timeHM_t* arrival, int* cost)`
- `int validateFlightPath(flight_t** flight_list, char** airport_name_list, int size)`

Descriptions for each function can be found in `flights.c`.

You have the freedom to design the structs and the way you store the information however you like (so have fun!). However, your code must be able to handle an arbitrary number of airports and schedule entries; in other words your data structures must be able to grow dynamically. **You are not allowed to use a single fixed sized array!** One possibility is linked list structures. Another is to use arrays but create larger ones to replace ones that are out of space (in which case you may find `realloc` useful). If you are out of memory (e.g. `malloc` fails), call `allocation_failed()` (given in `flights.c`).

Input Validation

A portion of your grade will also be based upon making sure your functions are robust. This means your functions should be able to handle incorrect/invalid inputs without crashing. While this is not possible in a completely general sense (for example there is no way to check an address is valid), you should be checking for NULL, that all flights depart before they arrive (there are no overnight flights), and that all costs are non-negative. We will provide autograder tests for all of these, which will be made available to you soon.

Testing

A `Makefile` is provided for you. To run the provide tests you can run

```
// creates executable RouteTime
$ make
// run a program on the provided files from the config file
// Config file has 3 args on each line -- (1) list of airports, (2) schedules,
(3) routes
$ ./RouteTime integration_config
```

Alternatively you can just run:

```
// Runs both of the above commands
$ make run
```

The program takes in a single argument `integration_config`. This file is a text file with 3 file names on each line:

1. **list of airports** – airport names, one on each line (no guarantees about name length!)
2. **schedules** – listed by airport and separated by blank lines. `AIRPORT: source_airport_name` starts a schedule, followed by lines of the format `destination_airport departure_time arrival_time $cost_of_flight`.
3. **routes** – each route begins with `ROUTE: route_name start_airport time_now` and subsequent lines are airport names along the route.

To add additional tests that you create you simply need to add a new line to `integration_config`, containing the names of the 3 files you want to use (you can reuse or swap some or all of the files for you new tests).

You should test each part individually as you go along. Once you have completed `printAirports()`, RouteTime should echo the airports listed in `airports.txt` correctly. Similarly, once you have completed `printSchedule()`, the schedules in `schedules.txt` should be echoed. With `getNextFlight()` completed, the program should be able to give the correct times for when routes (given in routes.txt) should finish, or say that it's not possible. The correct output is given in `flights.out`.

Having just the correct output will not be enough, though! We'll check that your code doesn't access memory it shouldn't be and that it doesn't leak any memory. We can test for this with `Valgrind's memcheck tool`. This is a tool that can help you catch memory problems too:

```
// runs Valgrind on RouteTime with default arguments
$  make flights-memcheck
```

The tool will tell you where you are making invalid reads or writes and what memory is leaked (memory that you've lost all pointers to). The program should not have any of these errors nor leaked memory (don't worry about Valgrind's report on "still reachable" or "suppressed").

IMPORTANT: We will grade your solution using different input files so you should test your code carefully! In particular, you will notice that the input files we have provided do not thoroughly test your code. These input files should be used as a sanity check only. In addition, we will make no guarantees on the length of airport names, schedules, or routes or the size of the airport system. Finally, any changes you make to files other than `flights.c` and `flight_structs.h` will NOT be used in grading.

Debugging

When you encounter bugs in this assignment you should use either `cgdb` or `valgrind`. To use `cgdb` you can follow the following steps

IMPORTANT: Make sure you are either debugging on the Hive computers (sshing into your instructional account) or you have installed a version of cgdb and Valgrind on your local computer, otherwise these commands will not work.

```
// Make sure your executable is up to date
$ make
// run cgdb
$ cgdb RouteTime
// Set any break points, refer to lab01 for information on how to do this
...
// Run the program
$ run integration_config
```

At this point you can now begin stepping through your program. Since `integration_config` runs all tests if you are checking your own tests you may wanna set your breakpoints accordingly or provide a version of `integration_config` with just the tests you seek to run.

Additionally as previously mentioned you can run valgrind with `make flights-memcheck`. While this is useful for finding memory leaks it can also be useful for determining where illegal reads/writes have occurred and when a program segfaults will provide you a stack trace. To teach you how to read this output here are a couple of examples from valgrind on different programs.

```
==28913== Invalid read of size 4
==28913==    at 0x10872F: main (cleared_ex.c:13)
==28913==  Address 0x522d040 is 0 bytes inside a block of size 4 free'd
==28913==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==28913==    by 0x1086FB: f (cleared_ex.c:6)
==28913==    by 0x10872A: main (cleared_ex.c:12)
==28913==  Block was alloc'd at
==28913==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==28913==    by 0x108710: main (cleared_ex.c:10)
==28913==
==28913== Invalid write of size 4
==28913==    at 0x108738: main (cleared_ex.c:13)
==28913==  Address 0x522d040 is 0 bytes inside a block of size 4 free'd
==28913==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==28913==    by 0x1086FB: f (cleared_ex.c:6)
==28913==    by 0x10872A: main (cleared_ex.c:12)
==28913==  Block was alloc'd at
==28913==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==28913==    by 0x108710: main (cleared_ex.c:10)
==28913==
==28913== Invalid read of size 4
==28913==    at 0x10873E: main (cleared_ex.c:14)
==28913==  Address 0x522d040 is 0 bytes inside a block of size 4 free'd
==28913==    at 0x4C30D3B: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==28913==    by 0x1086FB: f (cleared_ex.c:6)
==28913==    by 0x10872A: main (cleared_ex.c:12)
==28913==  Block was alloc'd at
==28913==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==28913==    by 0x108710: main (cleared_ex.c:10)
==28913==
```

In this first image there is the line that states `Invalid read of size 4` on `cleared_ex.c:13`. This indicates that we are accessing memory that we don't have allocated on line 13 and suggests that when debug in `cgdb` this is where we should begin our debugging from. Unfortunately not all valgrind outputs are easy to read. Consider this second example:

```
==28772==
==28772== Conditional jump or move depends on uninitialised value(s)
==28772==    at 0x4E988DA: vfprintf (vfprintf.c:1642)
==28772==    by 0x4EA0F25: printf (printf.c:33)
==28772==    by 0x108721: main (no_segfault_ex.c:8)
==28772==
==28772== Use of uninitialised value of size 8
==28772==    at 0x4E9486B: _itoa_word (_itoa.c:179)
==28772==    by 0x4E97F0D: vfprintf (vfprintf.c:1642)
==28772==    by 0x4EA0F25: printf (printf.c:33)
==28772==    by 0x108721: main (no_segfault_ex.c:8)
==28772==
==28772== Conditional jump or move depends on uninitialised value(s)
==28772==    at 0x4E94875: _itoa_word (_itoa.c:179)
==28772==    by 0x4E97F0D: vfprintf (vfprintf.c:1642)
==28772==    by 0x4EA0F25: printf (printf.c:33)
==28772==    by 0x108721: main (no_segfault_ex.c:8)
==28772==
==28772== Conditional jump or move depends on uninitialised value(s)
==28772==    at 0x4E98014: vfprintf (vfprintf.c:1642)
==28772==    by 0x4EA0F25: printf (printf.c:33)
==28772==    by 0x108721: main (no_segfault_ex.c:8)
==28772==
==28772== Conditional jump or move depends on uninitialised value(s)
==28772==    at 0x4E98B4C: vfprintf (vfprintf.c:1642)
==28772==    by 0x4EA0F25: printf (printf.c:33)
==28772==    by 0x108721: main (no_segfault_ex.c:8)
==28772==
```

Here the trace seems to indicate that we need to step into the library for print. But what this actually means is that one of the arguments for print is an uninitialized value. Again this tells us where in main to look, even if it appears the actual error occurs inside the standard library.

For more insight into how valgrind can help debug code, refer to the bonus exercise in lab02.

Autograder

We will be releasing a partial autograder for this project on gradescope. This autograder will constitute 50% of your grade. It will consist of all the input validation tests and the tests originally distributed to you in the starter code.

Submission

Project submission is done through gradescope. We will only be accepting submissions on gradescope through Github. When you try and submit the first time, click a button that says `Connect to Github`. This will sync your github account to gradescope. Then you will be allowed to select your project repository from a list of your repositories.

You final grade will be released in the same assignment that you submit to the autograder for but with an updated autograder. Note that your final grade will **NOT** be your most recent autograder score after the deadline. We will run hidden tests and make a piazza announcement when final grades are available. All completely graded submissions are final and there will be no regrades. So make sure you do your own testing on your code in addition to the default autograder to be sure that you'll pass the hidden tests.