C++ PROJECT
198:314 Principles of Programming Languages
Spring 1999
due: April 15, 1999, at midnight

Project Description

In this project, you are asked to write a simulator for an airport within a limited airspace. You have to design the class hierarchy for different types of aircraft, and design and implement the airport as a class.

Airspace

The airspace has three dimensions and is modeled as a grid with the dimensions (xdim 0-49, ydim 0-49, zdim 0-1). An aircraft’s Location is a position vector within this space. A location of an aircraft is valid iff

- the location is within the airspace boundaries, and
- no other aircraft is at the same location, and
- the location is not inside the airport area, and
- if the location is within the landing approach zones, it has the low altitude, i.e., zdim = 0.

The exact position of the airport and landing zones is given below. For example, (15,0,1) and (23, 45, 0) are valid locations, but (-1, 3, 1) and (23, 34, 2) are not. Illegal locations will lead to a crash of the aircraft.

Aircraft

There are five types of aircraft in our simulation. Each aircraft has different characteristics and capabilities. The following tables gives a set of capabilities.

<table>
<thead>
<tr>
<th></th>
<th>can takeoff at night</th>
<th>can fly high</th>
<th>can hover</th>
<th>is fast</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blimp</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Helicopter</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Cargo</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Commuter</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Jet</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

You will have to implement methods that return yes/no answers for each of these capabilities. These methods should be placed in your class hierarchy at places where most code sharing is possible.

Each aircraft has also a set of different characteristics.
<table>
<thead>
<tr>
<th>Aircraft</th>
<th>Fuel Capacity</th>
<th>Fuel Consumption Per Time Step</th>
<th>&quot;char&quot; Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blimp</td>
<td>100 units</td>
<td>1 unit</td>
<td>'b' or 'B'</td>
</tr>
<tr>
<td>Helicopter</td>
<td>50 units</td>
<td>1 unit</td>
<td>'h' or 'H'</td>
</tr>
<tr>
<td>Cargo</td>
<td>1000 units</td>
<td>4 units</td>
<td>'c' or 'C'</td>
</tr>
<tr>
<td>Commuter</td>
<td>200 units</td>
<td>2 units</td>
<td>'o' or 'O'</td>
</tr>
<tr>
<td>Jet</td>
<td>1000 units</td>
<td>5 units</td>
<td>'j' or 'J'</td>
</tr>
</tbody>
</table>

Each time an aircraft lands, its tanks will be filled to capacity. During each simulation step, the current fuel level will be reduced by the units consumed during a simulation step. If a plane runs out of fuel, it will crash.

While the aircraft is landed, its "landed" field has to have the value true, otherwise it has to be false. Once the aircraft gets cleared for takeoff by the airport, its field "ready_for_takeoff" has to be set to true. Once the aircraft is airborne again, the flag "ready_for_takeoff" has to be reset to false. These two flags are used by the simulator to determine when to remove the aircraft from the display, and when to insert them back.

If an airborne aircraft occupies a location, its representative character is printed at the corresponding xdim, ydim location. A lower case character represents a low (zdim=0), and an upper case character a high altitude (zdim=1) location.

**make_move method**

The simulator calls the `make_move` method for each aircraft exactly once during each simulation time step. The parameters of `make_move` gives the current location of an aircraft and a pointer to the simulation object. This allows the aircraft to check different properties of the simulation environment such as wind direction, and the locations of other aircraft in its surrounding airspace. This information is useful to determine the next move of an aircraft. A call to `make_move` returns a DirectionVector object. This vector specifies the new location of the aircraft relative to its current location:

\[ \text{new location} = \text{current location} + \text{directionvector}, \]

where "+" is a element-wise vector addition.

**class DirectionVector {**

```
public:
    int xdir;
    int ydir;
    int zdir; // 0, 1, -1 only possible values!
    DirectionVector(int x, int y, int z);
};
```

Each entry of a direction vector may be either -2, -1, 0, 1, or 2 for Cargo and Jet (with the exception of `zdir`). All other aircraft are slower, i.e., may not have -2 or 2 as a direction vector entry. For an aircraft that can hover, the returned direction vector may be (0,0,0), i.e., xdir = 0, ydir = 0, and zdir = 0. For an aircraft that does not have the hover capability, such a direction vector would lead to a crash and therefore is illegal. Legal direction vectors that need to move in order to stay airborne have to respect the following constraint on the generated sequence of direction vectors:

*The same entry in a direction vector cannot change its sign ("+" or "-") from one time step to the next. A "0" entry has no sign.*
For example, generating the sequence (1,0,0), (0,1,0), (1,0,1) is legal, but (1,0,0), (-1,0,1) is not.

**Airport**

The airport area has a fixed set of locations within the airspace: \(22 \leq x_{dim} \leq 26\) and \(17 \leq y_{dim} \leq 32\). Aircraft cannot fly over the airport area, i.e., the airport locations are illegal. The airport has two landing approach zones.

1. **WEST zone** with locations \(22 \leq x_{dim} \leq 26\) and \(14 \leq y_{dim} \leq 16\)
2. **EAST zone** with locations \(22 \leq x_{dim} \leq 26\) and \(33 \leq y_{dim} \leq 35\)

At most one aircraft can be in the **WEST zone**. The same holds for the **EAST zone**. No aircraft can be in a landing zone at the high altitude \((x_{dim} = 1)\). An aircraft has to land to refuel before it runs out of fuel. An aircraft can only request to land if

1. it lands against the current wind direction, i.e., has to use the **WEST zone** in case the wind direction is from the east, and the **EAST zone** if the wind direction is from the west.
2. it is within the predefined landing zone at each end of the runway (east end or west end, depending on wind direction), and
3. it has no more than half of its fuel left.

Even if all these conditions are satisfied, an aircraft may not be given landing permission if the airport is full.

**ASSIGNMENT**

You should modify the given structure of the programming project as little as possible. Only a few changes to the files simulation.h and simulation.C should be required. The goal of the assignment is to come up with a clean object oriented design and implementation. You don’t necessarily get a top grade if you manage to avoid crashes, i.e., make your aircrafts behave very “smart”. However, your aircrafts should have some form of intelligence (just flying against the airspace boundaries or the airport is not acceptable!). Classes are defined in “.h” files, and their methods are implemented in the corresponding “.C” files.

1. Design and implement the class hierarchy, including the data fields and methods for the different types of aircraft. This will involve the implementation of all virtual functions defined in class Aircraft. The classes that implement the different aircrafts (blimp, helicopter, ...) are all derived from class Aircraft, i.e., Aircraft is the root of the class hierarchy.

2. Design and implement the class Airport. This involves the design of an interface between the airport and the simulation, and between the airport and aircrafts.

Your implementation has to handle landing requests from aircrafts. The airport should maintain a queue data structure (queue ADT) that contains all landed aircraft. At each time step, at most one aircraft is allowed to take off, assuming that the takeoff zone does not contain another aircraft. The takeoff zone is always opposite to the landing zone, i.e., **WEST zone** if the wind direction is from the west, and **EAST zone** if the wind direction is from the east.

Your airport has a predefined capacity, i.e., your airport may be full at some point during the simulation, requiring you to reject any landing requests for aircraft. The airport capacity is
defined in simulation.h as the value of AIRPORTCAPACITY. You may want to play around with different capacity values to test your airport simulation.

**Handout**

The directory $\sim$uli/cs314/project2 on the undergraduate UNIX cluster contains the initial code for your simulator. We are using the gnu C++ compiler (g++) for this project. The code does not compile as-is, since there are virtual abstract methods in base class Aircraft. Say make at the UNIX shell prompt to see what happens. Once you extended the code to define all virtual functions, an executable airport will be generated. airport has two parameters: the first allows you to regulate the speed of the simulation (smaller integer values lead to a faster simulation), while the second allows the specification of the number of aircraft in the simulation. The parameters are optional. Example: airport 1 100 – fastest simulation for 100 aircraft.

Initial aircraft types and positions are generated based on a random procedure. **NOTE:** Start early on the project since parts of this project may not be completely defined, and making reasonable design decisions is part of the problem. In other words, this project is not merely a “coding exercise”!

Good luck.