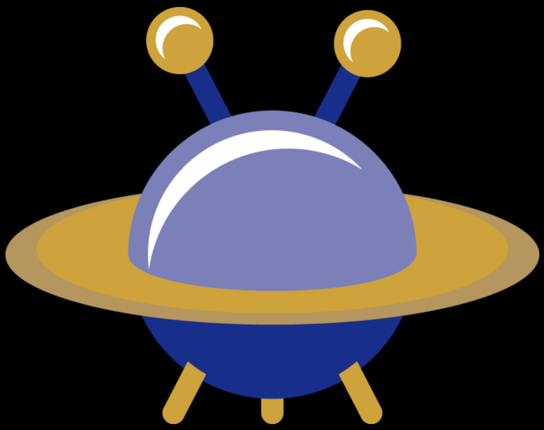
CS302

Assignment 8: NASA SpaceX



Description

As we know, a certain someone (the guy is super rich, named his car company after the scientist who discovered the 3 phase AC electric motor) is trying to discover life on other planets in the universe. They have recently discovered a planet and they have a set of space ships (some automated and some with people inside), that are trying to land on this planet. As they are trying to land, they might need to turn their thrusters on/off at certain times to conserve their fuel and to maintain their velocity since we do not want the lander to impact the surface of the planet at a high speed. Each lander does not control its own thrusters, it's controlled by one person on earth (due to budget cuts due to a pandemic of some sort), thus each lander with the highest priority needs to be processed (based on how close the lander is to the surface and also using a round robin type mechanism so all the landers are timeshared somewhat evenly). Your job is to implement this interface that allows this person to control this lander simulator using predefined throttle data to be used on each lander.

Lander Class

```
class lander
{
public:
  lander(double = 0, double = 0, double = 0, double = 0, double = 0, int = 0);
  double get_velocity() const;
  double get_altitude() const;
  double get_fuel_amount() const;
  int get_id() const;
  double get_mass() const;
  char get_status() const;
  bool change_flow_rate(double);
  void simulate();
  bool operator?(const lander&) const;
private:
  double flow_rate;
  double velocity;
  double altitude;
  double fuel_amount;
  bool stillLanding;
  int id;
```

```
    int timesSimulated;
    double LANDER_MASS;
    double MAX_FUEL_CONSUMPTION_RATE;
    double MAX_THRUST;
};
```

- `double flow_rate` - throttle percentage for the thrusters
- `double velocity` - velocity of the lander, if it's positive then the lander moving away from the surface, if it's negative then the lander is moving towards the surface
- `double altitude` - altitude (height) from the surface, if it's 0 then it made contact with the surface, well some sort of contact with the surface
- `double fuel_amount` - amount of fuel in the tank for the lander
- `bool stillLanding` - contains `true` if it is still airborne and `false` if no longer airborne
- `int id` - lander id
- `int timesSimulated` - a counter that denotes the amount of times the lander has been processed by the central tower (on earth)
- `double LANDER_MASS` - mass of the lander
- `double MAX_FUEL_CONSUMPTION_RATE` - max amount of fuel that can burned for each time the thrusters are activated for the lander
- `double MAX_THRUST` - the max amount of thrust the thrusters can do at maximum throttle amount
- `lander::lander(double mass, double max_thrust, double max_fuel, double alt, double fuel, int id)` - constructor that sets the appropriate fields of the lander object
- `double lander::get_velocity() const` - returns the velocity field
- `double lander::get_altitude() const` - returns the altitude field
- `double lander::get_mass() const` - returns the mass field
- `double lander::get_fuel_amount() const` - returns the fuel amount field
- `int lander::get_id() const` - returns the id field
- `char lander::get_status() const` - returns 'a' if still airborne, returns 'c' if not airborn and the velocity is less than or equal to -2 (lander crashed), and returns 'l' if not airborne and velocity is greater than -2 (lander has landed)
- `bool lander::change_flow_rate(double r)` - if r is between 0 and 1 (inclusive), then set flow rate with r if fuel amount is larger than 0 and the return `true`, return `false` if r is not between 0 and 1 (inclusive) and do not set flow_rate
- `void lander::simulate()` - if the lander is airborne
  1. Calculate the instantaneous velocity (TIME is a constant that contains 1)

     $$v = \text{TIME} \times \frac{\text{flow\_rate} \times \text{MAX\_THRUST}}{\text{LANDER\_MASS} + \text{fuel\_amount}} - 1.62$$

  2. Increment/update velocity field by v
  3. Increment/update altitude field by TIME $\times$ velocity field
  4. Update stillLanding field if necessary
  5. Decrement/update fuel_amount by TIME $\times$ MAX_FUEL_RATE $\times$ $|v|$

6. Set fuel_amount to 0 if the amount in the above step results to negative for fuel_amount
7. Increment/update timesSimulated by 1

- `bool lander::operator?(const lander& rhs) const` - you need to implement one of the following operators: `<, <=, >, >=` to be used in the priority queue class (min or max heap), you compare the priority of two `lander` objects (the `*this` object and the `rhs` object), the following tier system is how the priorities between the two objects can be determined:
  1. The object that has been simulated fewer times has the higher priority (this implements the round robin part of the priority)
  2. The object with the smallest altitude has the higher priority
  3. The object with less fuel has the higher priority
  4. The object with the larger mass has the higher priority
  5. The object with the smaller id number has the higher priority

Priority Queue Class

```
class priorityQ
{
public:
  priorityQ(int = 10);
  priorityQ(const priorityQ<Type>&);
  ~priorityQ();
  const priorityQ<Type>& operator=(const priorityQ<Type>&);
  void insert(const Type&);
  void deleteHighestPriority();
  Type getHighestPriority() const;
  bool isEmpty() const;
  void bubbleUp(int);
  void bubbleDown(int);
  int getSize() const;
private:
  int capacity;
  int items;
  Type * heapArray;
};
```

- `Type * heapArray` - array that contains the elements for priority queue
- `int capacity` - the length of the heapArray
- `int items` - the amount of items currently stored in the heap
- `priorityQ<Type>::priorityQ(int capacity)` - sets `this->capacity` with `capacity`, allocates a dynamic array to heapArray, and sets items to 1 or 0
- `priorityQ<Type>::priorityQ(const priorityQ<Type>& copy)` - copy constructor, performs a deep copy of the `copy` object to the `*this` object
- `priorityQ<Type>::~priorityQ()` - destructor
- `const priorityQ<Type>& priorityQ<Type>::operator=(const priorityQ<Type>& rhs)` - assignment operator, performs a deep copy of `rhs` object into `*this` object, remember to check for a self assignment and deallocate `this->heapArray` first before performing the actual deep copy

- `void priorityQ<Type>::insert(const Type& element)` - inserts element to the end of the heap and bubbles the element up, resizes if needed, also increments items counter by 1
- `void priorityQ<Type>::deleteHighestPriority()` - removes the root element by assigning the root with the end element in the heap and bubbles the element down, also decrements items by 1 (does nothing if the heap is empty)
- `Type priorityQ<Type>::getHighestPriority() const` - returns the highest priority item, the item at index 1 of the heapArray
- `bool priorityQ<Type>::isEmpty() const` - returns `true` if there are no items in the heap and `false` otherwise
- `void priorityQ<Type>::bubbleUp(int index)` - bubbles up an element in heapArray at index "index", and keeps bubbling up as needed, remember an the parent of element at index x in the heapArray is at index x / 2
- `void priorityQ<Type>::bubbleDown(int index)` - bubbles down an element from index "index", and keeps bubbling down as needed, remember the left child and right child of element x is at location 2 * x and 2 * x + 1 respectively, you always bubble down with the highest priority child, also remember if 2 * x > items then x is a leaf node and no bubbling down is needed
- `int priorityQ<Type>::getSize() const` - returns the amount of elements stored in the heap

Contents of main

In main, you first prompt for the input file for the lander data, each line the attributes for a single `lander` in the following order

```
mass max_thrust max_fuel alt fuel id
```

You would read these and store them into into a `lander` object using the constructor and then insert the `lander` object into the `priorityQ` of type `lander`. You perform this for each line in the input file. You would then close the input filestream.

Now you need to prompt for the input file that contains lander simulation info. Each line contains a potential value that can be used for a `lander` object's `change_flow_rate(double)` function. However each line could contain a number less than 0, larger than 1, or an invalid type (a string for example), thus you need to read until valid input is read.

Once a valid number is read, you need to get the highest priority lander object output its stats (refer to sample output), update its flow rate, simulate 1 unit of time (by calling its appropriate functions), if it landed or crashed report that, otherwise place it back into the priority queue, you perform this until either the priority queue is empty (which happens once all the landers crash/land) or until the end of file is reached. If end of file is reached before the priority queue is empty, then you need to output the landers that are still in the simulation.

Specifications

- Comment your code and your functions
- Do not add extra class members or remove class members and do not modify the member functions of the class
- No global variables (global constants are ok)
- Make sure your program is memory leak free

- Make sure `priorityQ` is truly a templated class, thus you don't access fields on the lander class, so `heapArray[x].get_altitude()` would not be a good idea since `Type` can be anything, you want to make this class work potentially with any data type as long as it has a comparison operator overloaded

Sample Run

```
Refer to the sample output text files
```

Submission

Compress your source files and upload to class website by deadline

References

- Link to the top image can be found at *https : //www.pinclipart.com/maxpin/iTwoTmT/*