Paper II

Problem 5 [21 points] Classes and Objects

This problem involves 2 classes called '`Dog`' and '`Human`'. Below are the header files of the 2 classes.

```
#ifndef DOG_H  /* Filename: Dog.h  */
#define DOG_H

#include <string>
using namespace std;

class Dog {
  // A line shall be added here to allow Human access any private data members
  // of Dog freely.

  private:
    string name;   // Name of the dog
    int weight;    // Weight of the dog
    int happiness; // Measures how happy the dog is, should be 100 initially

  public:
    // Constructor.
    // Initialize the Dog.
    Dog(string name, int weight);

    // Eat.
    // Increase the weight by 1.
    void eat();

    // Run.
    // Decrease the weight by half, round down to the nearest integer if needed.
    void run();

    // Lick another dog.
    // Increase the happiness of itself (this object) by 10.
    // Increase the happiness of anotherDog by 20.
    void lick(Dog& anotherDog);
};

#endif /* DOG_H */
```

```
#ifndef HUMAN_H  /* Filename: Human.h */
#define HUMAN_H

#include <iostream>
#include "Dog.h"

class Human {
  private:
    Dog** dogs; // It is a dynamic array of Dog pointers which point to Dog objects.
                // It should always be just big enough to hold all the dogs owned
                // by the human. e.g. if there are 5 dogs owned by the human,
                // the size of the dogs array should be exactly 5

    int dogCount; // The number of dogs the human owns,
                  // i.e., the size of the dogs array

  public:
    // Constructor.
    // Set dogs to NULL.
    // Set dogCount to 0.
    Human();

    // Destructor.
    // Perform memmory deallocation.
    // Make sure there is no memory leak.
    ~Human();

    // Adopt a dog.
    // Deep copy the given dog and add it to the end of the dogs array.
    // Make sure the array is just big enough to hold all dogs, as described.
    // Therefore, you need to increase the size of the array by 1.
    // If an existing dog already has the same name as the given dog, do nothing.
    void adoptDog(const Dog& dog);

    // Give a dog to someone.
    // Remove the dog of the given name from the dogs array.
    // Make sure the array is just big enough to hold all dogs, as described.
    // Therefore, you need to reduce the size of the array by 1.
    // Do nothing if no dog has the given name.
    void giveDog(string name);

    // Find the dog with the given name, and return it.
    // Return NULL if no dog has the given name.
    Dog* findDog(string name) const;

    // Pet the dog with the given name.
    // That means happiness of that dog will be increased by 30.
    // Do nothing if no dog has the given name.
    void petDog(string name);
```

```
    void showDogs() const {
      cout << "I love my dogs: ";
      for(int i=0;i<dogCount;i++)
        cout << dogs[i]->name << (i<dogCount-1?",":"\n");
    }
};

#endif /* HUMAN_H */
```

Your task is to implement the missing member functions in the two given classes '`Dog`' and '`Human`'. The following is the test program "`test-human-dog.cpp`" for the 2 classes.

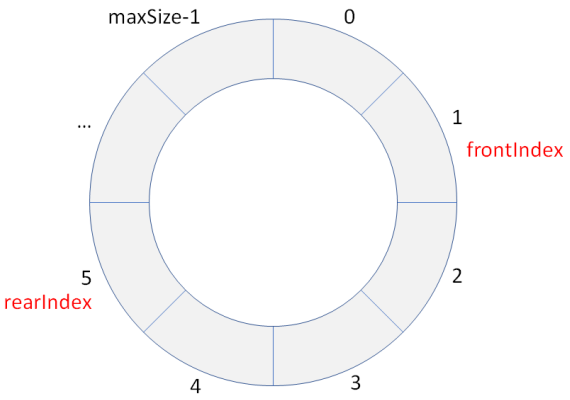
```
#include <iostream>  /* test-human-dog.cpp */
#include "Dog.h"
#include "Human.h"

using namespace std;

int main()
{
  Dog shiba("kuro", 20);
  Dog husky("terminator", 40);
  Dog hkOriginal("dimsum", 30);

  Human tina;

  tina.adoptDog(shiba);
  tina.adoptDog(husky);
  tina.adoptDog(hkOriginal);

  tina.showDogs();

  cout << "Gift terminator to my son!" << endl;
  tina.giveDog("terminator");

  tina.showDogs();

  return 0;
}
```

A sample run of the test program is given as follows:

```
I love my dogs: kuro,terminator,dimsum
Gift terminator to my son!
I love my dogs: kuro,dimsum
```

Problem 6 [20 points] Class Template

This problem involves the implementation of a template class `Queue` using a dynamic circular array. A queue is a data structure in which objects are added to the back (specified by `rearIndex`) of the queue, and removed from the front (specified by `frontIndex`) of the queue, and it enforces First-In-First-Out (FIFO) behaviour, while a circular array is an array with the end of the array wraps around to the start of the array. The following shows a queue implemented using a circular array.



Write a template class `Queue` that stores a list of objects using a dynamic circular array and provides the following data members and member functions:

- Data members
  - `maxSize`: a constant variable that represents the maximum size of the dynamic array.
  - `data`: a pointer that points at a dynamic array of type `T`, where `T` is a type name.
  - `frontIndex`: an `int` that represents the index of the front object in the queue.
  - `rearIndex`: an `int` that represents the index of the last object in the queue.
- Member functions
  - a constructor with an `int` parameter, `maxSize`, and it dynamically allocates an array of objects in type `T` according to the parameter `maxSize`. As the queue is initially empty, both `frontIndex` and `rearIndex` should be initialized to -1.
  - a copy constructor that performs deep copy.
  - a destructor that de-allocates the dynamic array.
  - a constant member function `isEmpty` that returns a `bool` value indicating whether the queue is empty.
  - a constant member function `isFull` that returns a `bool` value indicating whether the queue is full.

- `front()` that returns a reference to the front object in the queue, without taking the front object out of the queue.
- `enqueue()` that accepts an object of type `T` using a constant reference parameter, and it inserts the object to the back of queue. It returns true if the object is successfully inserted to the queue. Otherwise, it returns false.
- `dequeue()` that removes the object at the front of the queue. It returns true if the object is successfully removed from the queue. Otherwise, it returns false.

For simplicity, implement all the member functions **INSIDE** the class template definition in a single file called "`Queue.h`". Note that your solution should work with the testing program "`test-queue.cpp`" and produce the following outputs:

```
Enqueue 20 and 30
Front element: 20
Dequeue: Success
Front element: 30
Dequeue: Success
The queue is empty
```

```
#include <iostream>   /* Filename: test-queue.cpp */
#include "Queue.h"
using namespace std;

int main() {
  Queue<int> intQueue(10);  // A queue of size 10

  cout << "Enqueue 20 and 30" << endl;
  // Insert data to the back of the queue
  intQueue.enqueue(20);
  intQueue.enqueue(30);

  // Print data at the front of the queue
  if(!intQueue.isEmpty())
    cout << "Front element: " << intQueue.front() << endl;

  // Remove data from the front of the queue
  cout << "Dequeue: " << ((intQueue.dequeue()) ? "Success" : "Fail") << endl;
  // Print data at the front of the queue
  if(!intQueue.isEmpty())
    cout << "Front element: " << intQueue.front() << endl;

  // Remove data from the front of the queue
  cout << "Dequeue: " << ((intQueue.dequeue()) ? "Success" : "Fail") << endl;
  if(!intQueue.isEmpty())
    cout << "Front element: " << intQueue.front() << endl;
  else
    cout << "The queue is empty" << endl;
}
```

Problem 7 [23 points] Operator Overloading

The following shows a mini application that involves 4 classes called '`Staff`', '`Customer`', '`CarPickup`', and '`Shop`'. The class '`Shop`' consists of two queue containers that are instantiated using the template class you have just defined in Question 6 to store `Staff` and `Customer` objects. Given the following complete definition and implementation of the first 2 classes, '`Staff`' and '`Customer`',

```
#include <iostream>  /* Filename: Staff.h */
#include <string>
using namespace std;

class Staff {
  private:
    string name; // Name of staff

  public:
    Staff(string n = "") { name = n; } // Default / Conversion constructor

    // Friend declaration
    friend class Shop;
    friend ostream& operator<<(ostream& os, const Staff& s) {
      os << s.name;
      return os;
    }
};
```

```
#include <iostream>  /* File: Customer.h */
#include <string>
using namespace std;

class Customer {
  private:
    string name; // Name of customer

  public:
    Customer(string n = "") { name = n; } // Default / Conversion constructor

    // Friend declaration
    friend class Shop;
    friend ostream& operator<<(ostream& os, const Customer& c) {
      os << c.name;
      return os;
    }
};
```

your task in this question is to complete all the missing parts in `CarPickup` and `Shop` classes so that the given testing program "`test-shop.cpp`" will compile, run, and produce the expected output.

```cpp
#include <iostream>  /* Filename: test-shop.cpp */
#include "Shop.h"
using namespace std;

int main() {
  // Instantiate a shop object
  Shop shop("HKUST Bakery", 50);

  // Instantiate two staff objects
  Staff wallace("Wallace");
  Staff luke("Luke");

  // Instantiate two carpickup objects
  CarPickup car1(wallace);
  CarPickup car2(luke);
  shop << car1 << car2;   // Add two staff to the staffList of the shop

  // Instantiate two customer objects
  Customer john("John");
  Customer peter("Peter");
  shop << john << peter;  // Add two customers to the customerList of the shop

  cout << shop;           // Print the shop

  Staff staff;            // Instantiate a staff object

  // Instantiate a carpickup object
  CarPickup car3(staff);  // Instantiate a carpickup object
  shop >> car3;           // Pickup a staff who will be off-duty
  cout << "Off duty: " << staff << endl;  // Print the staff

  CarPickup car4(staff);  // Instantiate a carpickup object
  shop >> car4;           // Pickup a staff who will be off-duty
  cout << "Off duty: " << staff << endl;  // Print the staff

  Customer customer;      // Instantiate a customer object
  shop >> customer;       // Serve a customer, remove the customer from the queue
  cout << "Served: " << customer << endl; // Print the details of the served customer

  return 0;
}
```

Expected output of the testing program:

```
Name: HKUST Bakery
Size (in square feet): 50
Staff: Wallace Luke
Customers: John Peter
Off duty: Wallace
Off duty: Luke
Served: John
```

/* Rough work */

/* Rough work */

/* Rough work */