The University of Chicago
Department of Computer Science

CMSC 15200 – Introduction to Computer Science 2
Summer Quarter 2007
Homework #6 (08/10/2007)
Due: 08/15/2007 @ 1:30pm

Name:

Student ID:

Instructor: Borja Sotomayor

Do not write in this area

1	2	3	4	5	**TOTAL**

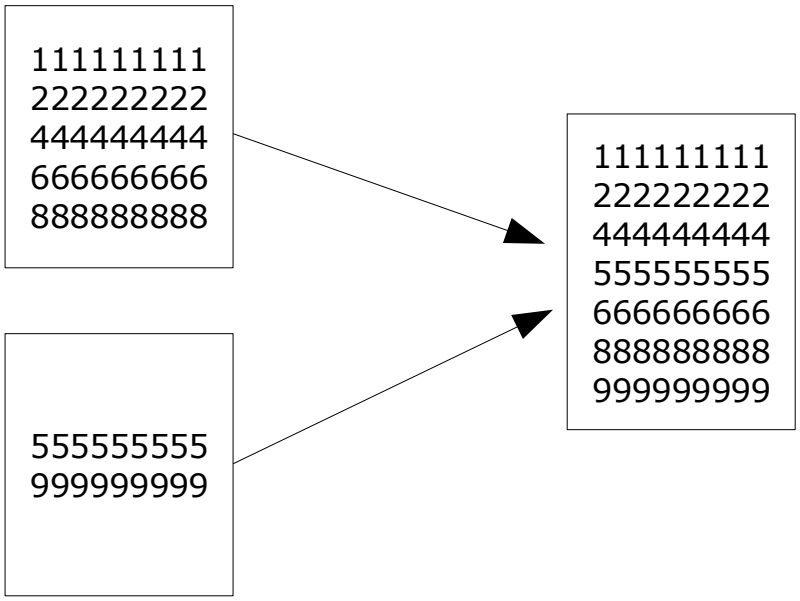
Maximum possible points: 65 + 20

In this homework, your exercises must be compiled using **`make`**. In particular, the instructor must be able to compile your exercises simply by running **`make exN`** (where N is the exercise number; e.g., **`make ex3`** to compile exercise 3).

Exercise 1 <<20 points>>

PhoneCorp and PhoneTech, the two biggest phone companies in the US, have just completed a corporate merger. They are now faced with the daunting task of *merging* their client data files into a single file. In particular, each company has a text file with the social security numbers of their clients (one 9-digit number in each line) in increasing order. Your task is to create a program that takes those two files and creates a new file with the numbers from both files, in increasing order. You can assume that the two file have no numbers in common (i.e. the two sets of clients are disjoint)

For example:



Your program must be run like this:

`ex1 <clientfile1> <clientfile2> <result>`

A skeleton **`ex1.cpp`** file is provided showing how to retrieve the command-line parameters.

For full credit (10 points otherwise), your file must perform the merge doing a *single pass* through each of the files, without loading them into memory.

Note: Two example files (`clients1` and `clients2`) are provided in the homework files.

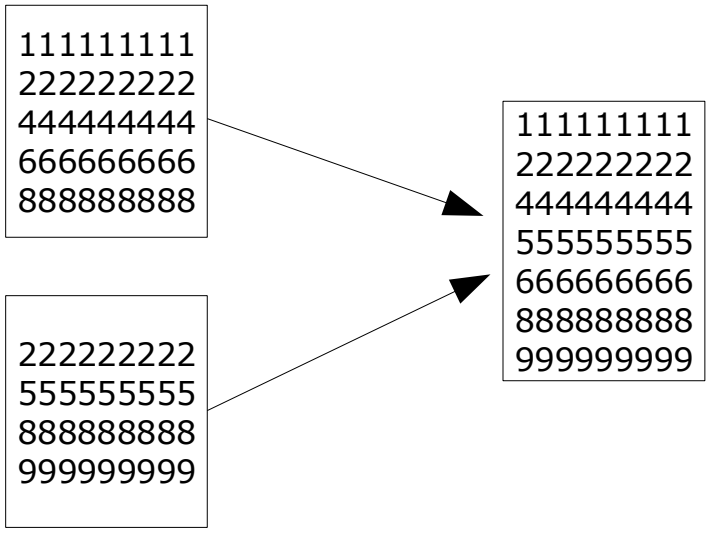
The result of correctly merging these two files is the following:

```
113624982
135648623
154564564
168943568
193215689
236584654
276546846
279461322
283216569
295464654
316243213
333218954
342654658
345654858
393546562
395221354
399321545
412135468
422354943
462213589
493215812
539565482
539635482
783213542
896546543
973213215
993219953
```

Hint 2: This is a good example of an exercise you should try to first solve with a more reduced problem set, before approaching the complete problem. For example, to familiarize yourself with the merging algorithm (without dealing with all the I/O messiness), try merging two 5-position arrays (preloaded with any integers you want, as long as they are in increasing order) into a 10-position array. When you do start to add the I/O code, first give your algorithm a try with smaller files than the ones provided (with single-digit integers, for example, which are easier to check than 9-digit numbers).

Exercise 2 <<5 points>>

Make a simple modification to Exercise 4 so that you program will be able to handle files with common numbers (i.e. the two companies share some clients in common, so the two sets of clients are *not* disjoint). For example:



Note: Two example files (`clients1_rep` and `clients2_rep`) are provided in the homework files. The result of correctly merging these two files is the following:

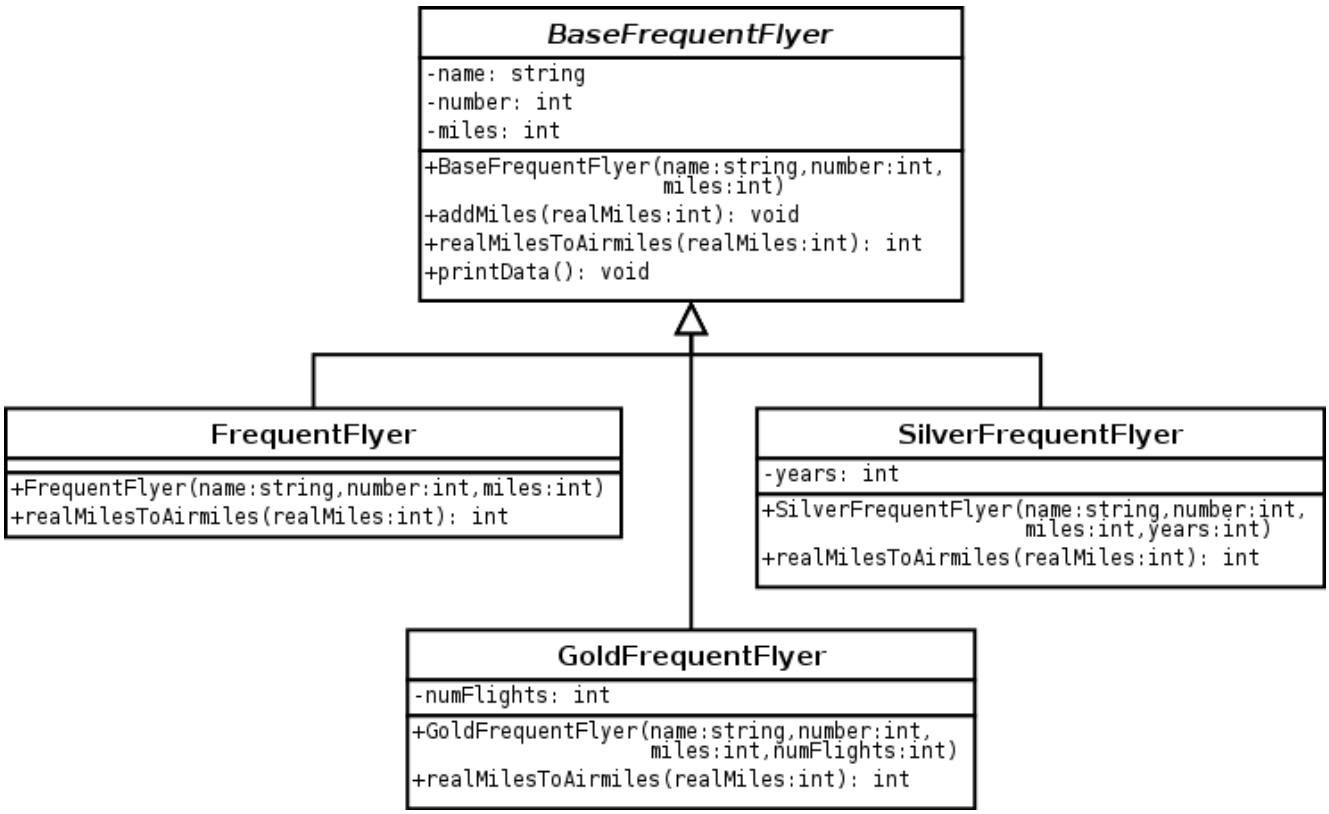
```
113624982
135648623
168943568
236584654
256684623
276546846
279461322
295464654
316243213
333218954
342654658
345654858
356698823
393546562
395221354
399321545
399645652
412135468
419654332
422354943
462213589
539565482
539635482
712315465
896546543
936532132
946546523
982132132
983213223
```

Exercise 3 <<30 points>>

We have been asked to develop a software solution for Nimbus Airlines' Frequent Flyer department. Nimbus Airlines has provided the following explanation of how its Frequent Flyer program works:

> In the Frequent Flyer (FF) department, we need to keep track of all the clients who have enrolled in the FF program. When a client signs up, he/she must provide his/her *name*, and is assigned an 8-digit *number*. The client's record also reflects the number of *miles* accumulated, which can later on be used to obtain free flights or upgrades.
>
> However, the number of miles accumulated in a client's FF account does not necessarily correspond with the actual number of miles traveled by the client. Depending on the client's category within the FF program, he/she might accumulate more miles than actually traveled. We usually use the terms *real miles* and *air miles* to distinguish both types of miles (the former being the real number of miles traveled, and the latter being the number of miles accumulated in the client's account).
>
> Our FF program has three types of frequent flyers:
>
> - The ordinary *frequent flyer* is the lowest category. Clients in this category receive as many air miles as real miles.
> - The *silver frequent flyer* rewards clients that have been with us for a long time. When a silver frequent flyer adds miles to his/her account, the number of air miles is equal to the number of real miles times the number of years, plus one, that client has been enrolled in the program. For example, if a silver frequent flyer takes an 800 mile flight, and he/she has been enrolled for three years, the number of air miles is 800 * (3+1) = 3200.
> - The *gold frequent flyer* rewards clients that fly with us frequently. When a gold frequent flyer adds miles to his/her account, the number of air miles is equal to the number of real miles times one tenth (rounded down) of the number of times he/she has flied with us (plus one). For example, if a gold frequent flyer takes an 800 mile flight, and he/she has previously been on 97 flights, the number of air miles is 800 * (97/10 + 1) = 800 * (9 + 1) = 8000.

Our software design department has come up with the following object-oriented model for the stated problem:



The design department has also provided the following comments to the diagram:

- printData() prints the client's name, number, and miles.
- realMilesToAirmiles() converts a number of real miles to air miles, taking into account the client's category.
- addMiles() adds a number of real miles to a user's account. Internally, this function must use the realMilesToAirmiles() function to determine what amount of airmiles must be added to the *miles* member variable.
- The diagram does not reflect what functions are virtual.

Implement the four classes described in the UML class diagram shown above. The points in this exercise are divided as follows:

- 15 points for implementing the four classes, correctly using the inheritance features in C++.
- 15 points for correctly implementing addMiles() and realMilesToAirmiles() using the polymorphism features in C++.

To test your implementation, a main.cpp file is provided in the homework files that should yield the following output:

```
Cornelius Doe        #23166841       200mi
   Lucius Doe        #94565432       700mi
    Rufus Doe        #32155994      1200mi
```

No header files are provided, so the main.cpp file will only work if you use exactly the same class and function names shown in the UML diagram.

Exercise 4 <<10 points>>

You will write a simple program that will ask the user for an undefined amount of numbers (i.e. the user just keeps on inserting numbers until he decides to stop). You will use an STL vector to store these numbers. Once the user has finished typing in numbers, you will print out the contents of the vector (a) using the vector class's bracket operator, and (b) using iterators. Then you will sort the vector using one of the STL algorithm functions, and will print the contents of the vector again.

```
Type in number #1: 56
Another number? (y/n): y
Type in number #2: 12
Another number? (y/n): y
Type in number #3: 78
Another number? (y/n): y
Type in number #4: 99
Another number? (y/n): y
Type in number #5: 34
Another number? (y/n): y
Type in number #6: 2
Another number? (y/n): y
Type in number #7: 89
Another number? (y/n): n

Numbers: 56 12 78 99 34 2 89
Numbers (with iterators): 56 12 78 99 34 2 89

Sorting...

Numbers: 2 12 34 56 78 89 99
Numbers (with iterators): 2 12 34 56 78 89 99
```

Exercise 5 <<Bonus points: 20 points>>

You are provided with the same Tree implementation provided in the previous homework. Implement the `breadth` function:

```
void breadth(Tree &t);
```

This functions does a breadth-first traversal of the tree. This algorithm has not been explained in class, so you will have to look it up elsewhere. Note that most sources show the algorithm for traversing cyclic graphs. The algorithm for breadth-first traversal of a tree is slightly simpler. Hint: You will not need to make any modifications to the TreeNode struct. Hint #2: You will need to use a queue for this traversal. An implementation of a Queue ADT is provided for you.

A main.cpp file is provided for you to test your implementation. If correctly implemented, the program should print out the following:

```
BREADTH-FIRST TRAVERSAL
-----------------------
15 10 23 4 13 17 29 12 25
```