



**The University of
Chicago**
Department of
Computer Science

CMSC 15200 – Introduction to Computer Science 2
Summer Quarter 2006
Homework #8 (08/18/2006)
Due: 08/23/2006

Name:

Student ID:

Instructor:

Borja Sotomayor

Do not write in this area							
1	2	3	4	5	6	7	TOTAL

Maximum possible points: 75

This homework assignment is divided into two parts: one related to the fundamental OO features of C++ (the "Dynamic Array" part) and one related to modeling OO systems (the "Nimbus Airlines" part)



Dynamic Array

You will implement a DynamicArray ADT (Abstract Data Type), a C++ class that will allow us to use a dynamic integer array without worrying about all the memory management issues. To accomplish this, DynamicArray will have two private member variables: a pointer to int (which will point to an array allocated in the heap) and the size of the array. It is the responsibility of the DynamicArray class to correctly initialize these private member variables and to make sure that they are always in a consistent state.

This dynamic array has the following characteristics:

- Its positions are numbered starting with 0, like C/C++ arrays.
- The array must *grow* to meet the user's needs. For example, if the user creates an array of size 5, and the user then wants to assign a value to position 9, then DynamicArray must automatically grow its size to 10.
- The values in the array can only be non-negative integers.
- Because the array can dynamically grow, there might be "empty positions". For example, if the user creates an empty array, assigns a value in position 0 and then a value in position 4, the array will have size 5, but positions 1, 2, and 3 will be "empty". Empty positions will have value -1.



The class declaration is the following (DynamicArray.h in the homework files):

```
class DynamicArray
{
    private:
        int *a;
        int s;

        int grow(int s2); // Optional

    public:
        // Constructors
        DynamicArray();
        DynamicArray(int s);
        DynamicArray(int *a2, int s2);
        DynamicArray(const DynamicArray &da);
        ~DynamicArray();

        // Member functions
        int getSize(); // Already implemented
        int setValue(int pos, int v);
        int getValue(int pos, int &v) const;

        // Operator overload
        DynamicArray& operator=(const DynamicArray &da2);
        bool operator==(const DynamicArray &da2) const;
        friend ostream& operator<<(ostream &os, DynamicArray &da);
        int& operator[](int pos);
        int operator[](int pos) const;
};
```

(Note: The header file also includes some constant member variables not shown above)

To test your list implementation, a main.cpp is provided in the homework files. Running this program with a correct DynamicArray implementation should yield the following:

```
FIRST ASSIGNMENT
Had to grow the array.

SECOND ASSIGNMENT
Had to grow the array.

THIRD ASSIGNMENT
Did not have to grow the array.

ARRAY A1: 37 -1 7 -1 23
ARRAY A2: 42 42 42 42 42
ARRAY A3: 42 42 42 42 42
ARRAY A4: 42 42 42 42 42

ACCESSING VALUES
Position 2: 7
Position out of range.

COMPARISONS
```



a1 and a2 don't have the same contents.
a1 and a3 don't have the same contents.
a2 and a3 have the same contents.

ASSIGNMENT

ARRAY A1: 37 -1 7 -1 23
ARRAY A2: 42 42 42 42 42
ARRAY A3: 42 42 42 42 42
ARRAY A4: 37 -1 7 -1 23

BRACKET OPERATOR

a1[2] = 7
a1[4] = 23
a1[10] = 100
ARRAY A1: 37 -1 7 -1 23 -1 -1 -1 -1 -1 100

Exercise 1 <<5 points>>

Implement the constructors and destructor:

```
/* Creates an empty array (size 0) */  
DynamicArray();  
  
/* Creates an array of size "s" */  
DynamicArray(int s);  
  
/* Initializes the array with the values of another array. The  
 * constructor receives "a2" (the 'source array') and its size.*/  
DynamicArray(int *a2, int s2);  
  
/* Copy constructor. */  
DynamicArray(const DynamicArray &da);  
  
/* Destructor. */  
~DynamicArray();
```

Exercise 2 <<10 points>>

Implement the setValue operation:

```
/* Modifies the values in position "pos" with new value "v".  
 * Must return the following:  
 * 0: Success. Did not need to grow the array.  
 * 1: Success. Had to grow the array.  
 * 2: Error. Specified position is < 0  
 * 3: Error. Specified value is < 0  
 */  
int setValue(int pos, int v);
```

Take into account that setValue must *grow* the array if the user specifies a position that is larger than the size of the array. For example, if the current size of the array is 5 and the user specifies a value for position 14, then the array must grow to a size of 15. To make your setValue implementation simpler, it might be useful to think of *growing* as a



separate operation:

```
/* Grows the array to size "s2". This means copying the current array
 * to a new array of size "s2", and filling all the new positions with
 * value -1 (empty positions).
 */
int grow(int s2);
```

Implementing the *grow* function is *optional* (you can just include all the growing code inside the *setValue* function).

Also, notice that *setValue* must return a status code. These status codes are already defined as constant member variables in class *DynamicArray*:

```
static const int SET_OK_NOGROW = 0;
static const int SET_OK_GROW = 1;
static const int SET_ERROR_NEGPOS = 2;
static const int SET_ERROR_NEGVALUE = 3;
```

Exercise 3 <<5 points>>

Implement the *getValue* function:

```
/* Returns the value in position "pos" using parameter "v".
 * The function must return the following:
 * 0: Success.
 * 1: Error. A bad position was specified.
 */
int getValue(int pos, int &v) const;
```

Exercise 4 <<10 points>>

Overload the assignment and equality operators:

```
/* Assignment operator overload */
DynamicArray& operator=(const DynamicArray &da2);

/* Equality operator overload.
 * Returns true if both DynamicArrays are the same. We assume
 * that two DynamicArrays are the same if (a) they have the same
 * size and (b) they have the same value in every position. */
bool operator==(const DynamicArray &da2) const;
```



Exercise 5 <<5 points>>

Overload the shifting operator to print a DynamicArray to an output stream:

```
friend ostream& operator<<(ostream &os, DynamicArray &da);
```

Exercise 6 <<10 points>>

Overload the bracket operator:

```
int& operator[](int pos);  
int operator[](int pos) const;
```

The bracket operator overload was not discussed in class, so you will have to read about it on your own. You must overload the bracket operator in such a way that the user can access a position in the dynamic array by using the bracket operator, instead of the `getValue` function. For example:

```
int a[5] = {42,42,42,42,42};  
DynamicArray al(a,5);  
  
cout << al[3]; // Prints out 42
```

Note that the bracket operator in this case is not as safe as the `getValue` function, as there is no way of returning a status code telling if the specified position is valid or not. So, if the user specifies a non-existing position, the bracket operator must return it, regardless of whether it exists or not (this mimics the behaviour of C/C++ arrays).

Finally, notice how there are *two* bracket operator overloads. One is used when the `DynamicArray` is used as an l-value, and the other one when it is used as an r-value (you will have to figure out which is which!). The r-value version must simply return the value in the specified position, while the l-value version might have to grow the array so the position will be valid.

```
int a[5] = {42,42,42,42,42};  
DynamicArray al(a,5);  
  
cout << al[3]; // R-Value version, prints out 42  
a[1] = 5;      // L-Value version, doesn't grow the array  
cout << al[3]; // R-Value version, prints out 5  
a[10] = 37;    // L-Value version, has to grow the array
```



Nimbus Airlines

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.

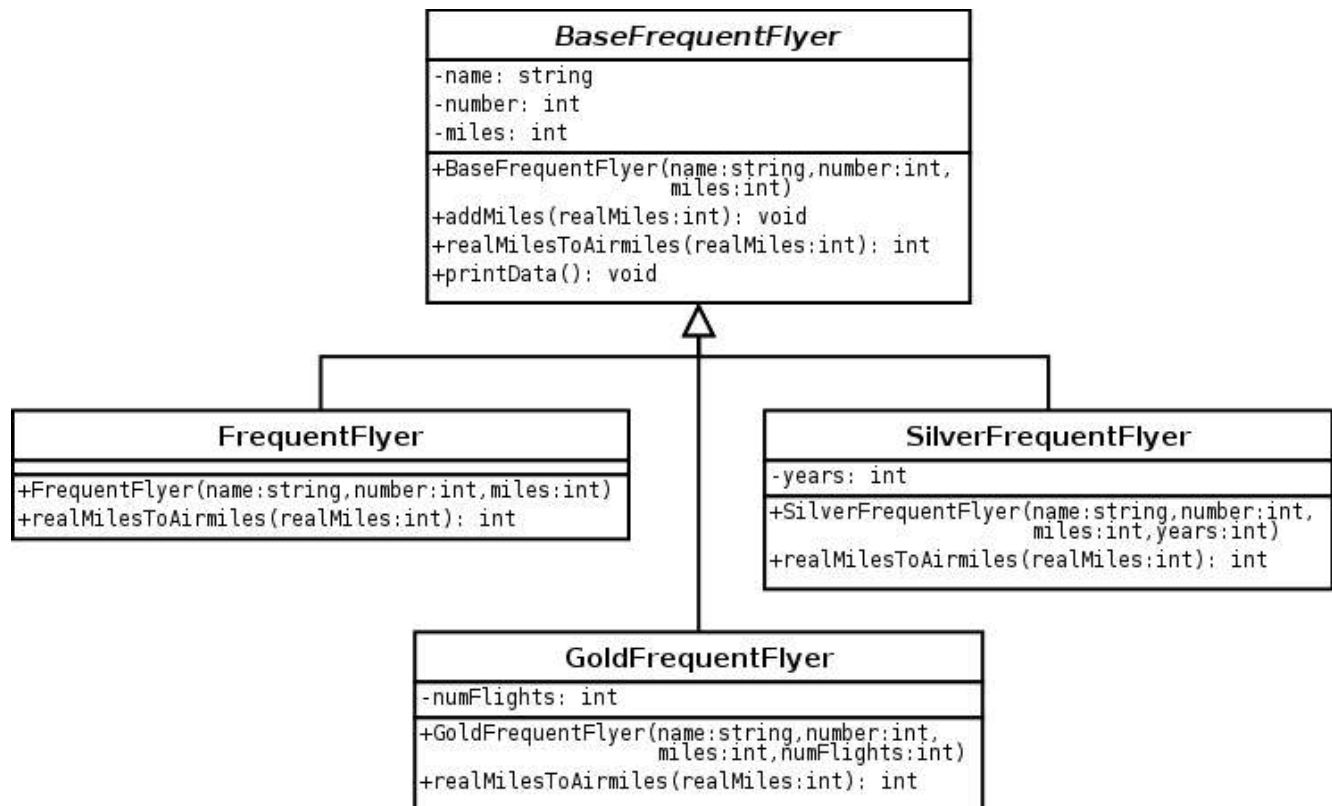
However, the number of miles accumulated in a client's FF account does not necessarily correspond with the actual number of miles traveled by the client. Depending on the client's category within the FF program, he/she might accumulate more miles than actually traveled. We usually use the terms *real 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.



Exercise 7 <<30 points>>

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.