

## CSC 112 Homework 5

Name \_\_\_\_\_

Due Date: Start of class, Wednesday 11/18/2009

You have been asked to help write an app for the iPhone. A co-worker has developed a PhoneBook class with the following definition:

```
class PhoneBook
{
    public:
        // default constructor - create an empty phone book
        PhoneBook();

        // copy constructor
        PhoneBook(const PhoneBook & inputPhoneBook);

        // destructor
        ~PhoneBook();

        // appends a new phone number to the phone book
        void append(const string & name, const string &
phoneNumber);

        // tests to see if two phone books are the same
        bool operator == (const PhoneBook & inputPhoneBook);

    private:
        // number of entries in this phone book object
        int numberOfEntries;

        // array of names for this phone book object
        string* entryNames;

        // array of phone numbers for this phone book object
        string* entryNumbers;

        // helper function which searches for a name and
returns the position of that name if it appears in the
phone book name array already, -1 if it is not in the phone
book already
        int locationOfName(const string & name);

        //helper function which searches for a phone number and
returns the position of that number if the number appears
in the phone book number array, -1 if it is not already
in the phone book
        int locationOfNumber(const string & number);
}
```

1. Write a **correct copy constructor implementation** for the *PhoneBook* class.

// the key thing is to make sure you dynamically allocate new arrays in the PhoneBook object being constructed

```
PhoneBook::PhoneBook(const PhoneBook & inputPhoneBook)
{
    int i;
    numberOfEntries = inputPhoneBook.numberOfEntries;
    entryNames = new string[numberOfEntries];
    for (i = 0; i < numberOfEntries; i++)
    {
        entryNames[i] = inputPhoneBook.entryNames[i];
    }

    entryNumbers = new string[numberOfEntries];
    for (i = 0; i < numberOfEntries; i++)
    {
        entryNumbers[i] = inputPhoneBook.entryNumbers[i];
    }
}
```

2. Write a correct **implementation of the == operator for the PhoneBook class**, where == is defined as **1) both phone books have the same number of entries, and 2) every name and number associated in the current phone book are present in the input phone book, also associated with each other.** Note that while the name and phone number have to be associated with each other in each phone book (ie William Turkett = 758-4427 in both), they don't have to be in the same position in the phone books. Assume that a name is only present in a PhoneBook once and that a number is only present in a PhoneBook once.

```
bool PhoneBook::operator==(const PhoneBook & inputPhoneBook)
{
    if (numberOfEntries != inputPhoneBook.numberOfEntries)
        return false;

    int j;
    int nameLocation, numberLocation;
    for (j = 0; j < numberOfEntries; j++)
    {
        nameLocation =
inputPhoneBook.locationOfName(entryNames[j]);
        numberLocation =
inputPhoneBook.locationOfNumber(entryNumbers[j]);

        if (nameLocation != numberLocation) return false;
        else if (nameLocation == -1) return false;
    }
    return true;
}
```

3. A strong argument can be made that “as little code as possible” should be written in solving a problem. In the context of operators and classes, this suggests that, as much as possible, one should make use of the correct implementation of one operator in implementing other operators. In doing this, testing for correctness can be primarily focused on those methods with complex implementations, while those methods that just use/lightly manipulate the other operators do not have to undergo significant testing.

Given the following set of operators to be overloaded, suggest and justify three pairs which, under reasonable semantics, should be able to be defined such that one of the pair of operators can be defined by primarily passing off the work to the other operator in the pair.

==  
!=  
=  
+=  
-=  
+  
-  
<  
>  
<=  
>=

The pairs (==, !=), (<, >=), and (>, <=) are all natural inverses of each other so that the answer to one is just the logical negation of the other. If dealing with numerical values, - could be implemented by addition (with the help of multiplying by -1). += and -= could employ the + and - operators respectively.

4. We have seen cases where we define an operator for a class such that the parameter passed to the operator is another variable of the same type (*Vector::operator\*(const Vector & v) for dotProduct*) and we have also seen the parameter as being of a primitive type (*Vector::operator\*(const double & v) for scaling*). I argue that we may also want to put a variable of some other class type as the parameter. Come up with an example of two classes where this makes sense. Feel free to choose any two classes and any operator where it makes sense, as long as you can justify your answer.

Assume we have a `LinkedList` container. We may want to employ the + operator so that it implements append for `LinkedLists` (add a new item to the back of the list). Then an appropriate type would be the data type being held in each node of the list. As an example, one might see:

```
void LinkedList<string>::operator+(const string & data)  
[noting that string is a class type itself]
```

5. In discussing the interplay between the `Node` class and the `LinkedList` class, we discussed that we would either need to make the `LinkedList` class be a friend class of the `Node` class (so that it could manipulate the `Node` instance variables, such as the pointer) or we would need to define `getData()/setData()/getNext()/setNext()` methods for the `Node` class. Is one of these approaches more restrictive in terms of who can modify the `Node` data than the other? If so, which one and why? If not, why not?

To me, the friend-based approach is much more restrictive. It only allows the declared friend, `LinkedList`, to be able to read and modify the `Node` internal variables. Using public `get/set` methods allows any other class or program to read and modify the `Node` internal variables. On the other hand, using the friend method, the friend has direct access to manipulate the variables. Using `get/set` methods, one can control how manipulation is done by code in the methods (such as validating the value being set for the pointer).

6. Given the definition of a `LinkedList` class we started today,

```
class LinkedList
{
    public:
        ...
    private:
        Node* head;
        int numberOfNodes;
};
```

suggest two ways to write an `isEmpty()` method for the `LinkedList` class.

```
bool LinkedList::isEmpty()
{
    return (head == NULL);
}
```

OR

```
bool LinkedList::isEmpty()
{
    return (numberOfNodes == 0);
}
```