DataBufferIterator and IterableDataBuffer Assignment

The world of data structures is about different ways of organizing and accessing data. There are many different kinds of data structures with different strengths and weaknesses, depending on how you need to use the data. Ideally, we would like our code to be able to use different kinds of data structures, without having to rewrite the code. You just use a different data structure. Part of what makes that possible is an abstraction for iterating through data, called an Iterator. C++ and Java each have their own version of the Iterator. The original Iterator was a pattern in the famous Design Patterns book by Gamma, Helm, Johnson, & Vlissides.

In this assignment, you will use the original Design Patterns version of the Iterator class, and create a subclass of that class called DataBufferIterator. The purpose of the DataBufferIterator is to iterate through objects of the DataBuffer class in a way that is abstract (the code that uses it doesn't have to know anything about the data structure that holds the data), and independent (there can be more than one iterator in use at the same time without interfering with each other).

For this assignment, you are to create another subclass of the DataBuffer class, called IterableDataBuffer. The IterableDataBuffer supports one new method, called createIterator. The createIterator method takes no arguments and returns a pointer to an object of the Iterator class. In reality, the Iterator class is an abstract class, meaning that the class itself has no implementation. The implementation is provided by a subclass of the Iterator class that is specific to the data structure over which it can iterate. For this assignment, you must therefore create a subclass of the Iterator class called DataBufferIterator, that implements all of the methods of the abstract Iterator class shown below.

```
class Iterator {
public:
    virtual void first() = 0;
    virtual void next() = 0;
    virtual int currentItem() = 0;
    virtual bool isDone() = 0;
};
```

In C++ the work virtual means that calls to that member function should use the most derived version of that function if the object is an object of a subclass, even if the pointer type is the base class. The assignment to 0 at the end of the declaration indicates that no implementation will be given in this class. Thus only subclasses that provide the implementations can be used. In other words, you cannot create an object of the Iterator class, itself.

In C++, without the virtual key word, the default behavior is to use the function of the class that matches the pointer type used in the call. In other words, if you have and add() function in your DataBuffer class, and an add() function in the OrderedDataBuffer class, then the add function that gets called depends on the type of pointer you use. If the pointer is OrderedDataBuffer*, then a call to the add function calls the implementation in the OrderedDataBuffer class. However, if the pointer is for DataBuffer, even if the object was created as an OrderedDataBuffer

```
DataBuffer* buf = new OrderedDataBuffer();
```

the function that is called is the one implemented in the DataBuffer class, unless the function was declared as virtual in its base class. The default behavior in Java is to treat all methods as virtual.

C++ doesn't do this by default because calls to virtual functions require an extra level of "indirection", which involves an extra memory reference to a table of function pointers for the address of derived functions, called the VTable. Memory references are inefficient and slow program execution.

Read the List explanation below to understand the iterator concept and how to implement an Iterator for a class called List. To help you get started, we are also giving you the code for the Iterator.h and DataBufferIterator.h files, and your main project file. Those two pieces appear at the end. This assignment again subclasses the original DataBuffer from assignment 6, and does not use the OrderedDataBuffer from assignment 7.

Description of the Iterator for a List Class

When you have a list of things, it is common to want to go through the list one element after the other. You might be looking for something in the list, or you might simply want to show everything in the list. This is easy to do with an array. You just write a "for" loop that uses the loop counter as an index to access every element of the array in order.

```
for (int i = 0; i < length; i++)
    cout << arr[i];
```

But not every list is an array. You will learn in Data Structures class that there are many different kinds of "lists", each with different advantages depending on how much data you have and the kinds of things you want to do. So we need a way of going through a list that works for any kind of list. Some list data structures do not have an index at all. For example, linked list and tree data structures have an order, but no index. The generic interface for going through a list is called an Iterator. The original Iterator Pattern, found in the Design Patterns book, has four methods in its interface which work as follows:

```
for (iter.first(); !iter.isDone(); iter.next())
    cout << iter.currentItem();
```

When part of a program wants to access elements in a list data structure, it uses an Iterator interface. Notice that there is nothing that corresponds to the index in the first example. The iterator itself keeps track of the current position in the list. First() sets the iterator to the first element in the list, while `next()` advances to the next element. Inside the iterator, it sets an index or a pointer to the current element. That way, after doing `next()`, the `isDone()` method can determine if it has run off the end of the list, and the `currentItem()` method can return the value of that element (or the element itself). The internal pointer or index of an iterator, the thing that keeps track of where in the list it is, is called a "cursor", just like the cursor in a text editor.

The obvious place to put the cursor is in the same class that holds the list. But the program may have several clients needing access to the data at the same time. If there is only one cursor, one client of the data may expect the cursor to be in one position, while another part of the program moves it to another location. Think of a situation in a document where you want to mark two different places in the text (for example to mark the beginning and end of a selection). The solution is make each iterator a separate object with its own cursor. The list has a method called `createIterator`() that returns an Iterator object that can be used to iterate over the list. Each time it is called it returns a new object.

```
Iterator iter1 = mylist.createIterator();
Iterator iter2 = mylist.createIterator();
```

The Iterator object must have a pointer to the object with list over which it can iterate and a cursor to keep track of its current position. But to work correctly, the Iterator class must access the internal (private) implementation of the list object where it uses its cursor to access elements of the list. In object oriented languages a class that has access to the private details of another class is called a "friend" class. We do this by putting a statement inside the List class, that identifies the Iterator class as its friend.

```
class List {
    friend class ListIterator;
    …
```

There is one last detail in C++ that has to be addressed. In C++, the one pass compiler means that the compiler has to already know what it needs to know when it encounters a line of code. In this case, it needs to know something about the names Iterator and ListIterator when it encounters a line with those names. It might seem simple to include the `ListIterator.h` header file for the Iterator class in the List.h header file for the List class. But, at the same time, you need to include the `List.h` header file in the `ListIterator.h` header file, since the ListIterator class needs a pointer to a List object. Unfortunately, that creates a circularity (like a dog chasing its tail). We resolve the problem of two classes referring to each other, by observing that the friend statement only needs to know that ListIterator is a class. It does not need to know anything more about it. So we simply include a prototype declaration in `List.h` declaring the Iterator and ListIterator names refer to a class. We include the `List.h` header file in `Iterator.h`, but not the opposite.

```
class Iterator;
class ListIterator;

class List {
    friend class ListIterator;
    Iterator* createIterator();
    …
```

Implementations of Iterator.h, DataBufferIterator.h, and the Main Assignment .cpp File

Implementations of the Iterator.h, DataBufferIterator.h file and your main assignment file, including two test functions appear below. Note that the original DataBuffer class must still work as before (except that things that were "private" must now be declared as "protected"). Follow the example modifications to the List.h file shown above. You must do the same thing in order to tell your IterableDataBuffer subclass that the DataBufferIterator is its friend.

(In case you are wondering, the Design Patterns book is often referred to as GoF, which is short for Gang of Four, in reference to the fact that the Design Patterns book has four authors.)

If you found the discussion of the Iterator above confusing, read the book or search online for tutorial discussions of friend classes, and virtual functions. Note that if you look for a tutorial discussion of the iterator, you will discover that both C++ and Java have their own versions of the iterator which are different than the GoF iterator. Our next assignment will be to implement the C++ style of Iterator, which involves the use of templates and operator overloading. The next

buffer assignment will introduce more complicated concepts, so make sure you understand and become comfortable with the concepts presented here, to prepare yourself for what comes next.

```
//
// File:    Iterator.h
// Project: GoF Iterator in C++
// Author:  Michael Van Hilst
// Version: 1.0 February 5, 2015
//
// Copyright (c) Michael Van Hilst 2015 All rights reserved.
//
class Iterator {
public:
    virtual void first() = 0;
    virtual void next() = 0;
    virtual int currentItem() = 0;
    virtual bool isDone() = 0;
};
```

```
//
// File:    DataBufferIterator.h
// Project: GoF Iterator in C++
// Author:  Michael Van Hilst
// Version: 1.0 February 5, 2015
//
// Copyright (c) Michael Van Hilst 2015 All rights reserved.
//
#pragma once

#include "Iterator.h"
#include "IterableDataBuffer.h"

class DataBufferIterator : public Iterator {
protected:
    IterableDataBuffer* collection;
    int cursor;
public:
    DataBufferIterator(IterableDataBuffer* collection);
    void first();
    void next();
    bool isDone();
    int currentItem();
};
```

```
//
// File:    Assignment06.cpp
```

```
// Project: GoF Iterator in C++
// Author:  Michael Van Hilst
// Version: 1.0 February 5, 2015
//
// Copyright (c) Michael Van Hilst 2015 All rights reserved.
//
#include <iostream>
#include "IterableDataBuffer.h"
#include "Iterator.h"
using namespace std;

void testDataBuffer(int arr[], int length);
void testIterableDataBuffer(int arr[], int length);

int main() {
    const int ARR0_LEN = 2;
    int arr0[ARR0_LEN] = { -2, -1 };
    const int ARR1_LEN = 10;
    int arr1[ARR1_LEN] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    const int ARR2_LEN = 25;
    int arr2[ARR2_LEN] = { 2, 4, 6, 8, 10, 12, 14, 16, 7, 6, 22, 8,
        9, 16, 5, 2, 7, 8, 12, 2, 0, 14, 17, 19, 22 };
    testDataBuffer(arr0, ARR0_LEN);
    testDataBuffer(arr1, ARR1_LEN);
    testDataBuffer(arr2, ARR2_LEN);
    testIterableDataBuffer(arr1, ARR1_LEN);
    testIterableDataBuffer(arr2, ARR2_LEN);
    return 0;
}

void testDataBuffer(int arr[], int length) {
    IterableDataBuffer data;
    data.copyData(arr, length);
    cout << endl; // blank line
    data.print();
    cout << "Sum " << data.sum() << endl;
    cout << "Min " << data.min() << endl;
    cout << "Max " << data.max() << endl;
    cout << "Range " << data.range() << endl;
    cout << "Mean " << data.mean() << endl;
}

void testIterableDataBuffer(int arr[], int length) {
    IterableDataBuffer data;
    data.copyData(arr, length);
    cout << endl; // blank line
    data.print();
    Iterator* iter1 = data.createIterator();
    iter1->next();
    cout << "The first iterator is on " << iter1->currentItem()
```

```
         << endl;
    Iterator* iter2 = data.createIterator();
    cout << "Using the second iterator: " << endl;
    for (iter2->first(); !iter2->isDone(); iter2->next())
        cout << iter2->currentItem() << " ";
    cout << endl;
    cout << "The first iterator is still on " << iter1->currentItem()
         << endl;
    delete iter1;
    delete iter2;
}
```