Programming Assignment #2

~~Files - Due Wednesday, February 5, 2014 at 5:00pm~~
Checklist – Due same day at the start of class

26 Jan 2015 12:30pm, before class.
Checklist submit on the same day in class.

This assignment gives you a chance to implement an API for your first container. The container is a modified double-linked list that has an interface that is a small subset of the `std::list` interface. (Everyone has implemented several linked-lists in the past so that aspect of the assignment is trivial.) There are a few enhancements that you will make to the typical linked-list interface. The task is to implement a templated class named *BList* which clients can use as a container of any type. I call it a BList because the nodes in the list are similar to nodes in a BTree, which we will discuss later in the semester. It's also kind of like a hybrid between a linked-list and an array in that it appears to be a linked-list of arrays. The arrays can be sorted or unsorted.

The essential difference from a "standard" linked-list is that each node can contain more than one data item. A standard linked- list has a one-to-one mapping of items to nodes. This means that a BList that has 2 items per node will require fewer nodes than a list with nodes containing only 1 item per node. A BList with 4 items per node will require even less nodes, and so on. This can provide a significant performance increase while only requiring some additional complexity in the code.

The *BList* class will also have the capability to maintain an order (a sort order) when using the *insert* method. The partial class definition is below:

```
template <typename T, unsigned Size>
class BList
{
  public:
    struct BNode
    {
      BNode *next;
      BNode *prev;
      unsigned count; // number of items currently in the node
      T values[Size];
      BNode() : next(0), prev(0), count(0) {} // default counstructor
    };

    BList()                  // default constructor
    BList(const BList &rhs) // copy constructor
    ~BList()                 // destructor

      // arrays will be unsorted
    void push_back(const T& value) throw(BListException);
    void push_front(const T& value) throw(BListException);

      // arrays will be sorted
    void insert(const T& value) throw(BListException);

    void remove(int index) throw(BListException);
    BList& operator=(const BList &rhs) throw(BListException);

    unsigned size() const;
    void clear();
    static unsigned nodesize();

  private:
    Node *head_;
    Node *tail_;
      // Other private fields and methods
};
#include "BList.cpp"
```

Notes (more details will be discussed in class)

1. Several public methods may throw a *BListException* exception. This exception class is already defined and implemented for you. There are only two kinds of exceptions for this assignment: out of memory exceptions (E_NO_MEMORY) and invalid index (E_BAD_INDEX). The E_DATA_ERROR is reserved.
2. If there is no memory to allocate a node, you should throw an exception with the code set to E_NO_MEMORY. You must catch the `std::bad_alloc` thrown by **`new`** and throw a `BListException`. Do not use `std::nothrow`. If you don't know what that is, you are probably not using it.
3. If an invalid index is provided to the *remove* method or the subscript operators, an exception should be thrown with the code set to E_BAD_INDEX.
4. The only class that you can use from the STL is **`std::string`** (or possibly **`std::stringstream`** for formatting error messages.) You cannot use any generic algorithms or functors. You may not even need **`std::string`**.
5. When a node contains 0 items (due to removing items), you must remove the empty node from the list and delete it.
6. Since this is a templated class, you don't know exactly what types are going to be put into the list. During the insert method and the find method you need to compare items. **Be sure to only use the less than operator and the equality operator for comparisons**. All data is guaranteed to support these two operators. There is no guarantee that the data supports any other operators
7. Make sure to test all of your code. If there is a method in the class that the driver does not call, it won't be instantiated and may fail during the grading. Also, pay attention to details like self-assignment; make sure you can deal with those kinds of things.
8. Finally, consult the driver for many more details. Almost every question you may have will be answered by looking at the examples in the driver.

Testing

As always, testing represents the largest portion of work and insufficient testing is a big reason why a program receives a poor grade. (My driver programs takes much longer to create than the implementation of the linked-list itself.) A sample driver program for this assignment is available. You should use the driver program as an example and create additional code to thoroughly test all functionality with a variety of cases. (Don't forget stress testing.)

What to submit

You must submit your program files (header and implementation file) and the index.chm file by the due date and time to the appropriate submission page as described in the syllabus.

Source Files	Description
BList.h	The header files. **No** implementation iis allowed by the student, although the exception class is implemented here. The public interface must be exactly as described above.
BList.cpp	The implementation file. **All implementation goes here**. You must document this file (file header comment) and functions (function header comments) using Doxygen tags as usual. Make sure you document all functions, even if some were provided for you.

Usual stuff

Your code must compile (using the compilers specified in this course) to receive credit. The code must be formatted as per the documentation on the website. Note that you must not submit any other files in the .zip file other than the 3 files specified. Details about what to submit and how are described in the syllabus.

Make sure your name and other info is on all documents.