Programming Assignment 2
Double Trouble

Time due: 9:00 PM Tuesday, January 31

Homework 1 gave you extensive experience with the Sequence type using both arrays and dynamically-allocated arrays. In this project, you will re-write the implementation of the Sequence type to employ a doubly-linked list rather than an array. You must *not* use arrays. You will also implement a couple of algorithms that operate on sequences.

Implement Sequence yet again

Consider the Sequence interface from problem 2 of Homework 1:

```
typedef TheTypeOfElementGoesHere ItemType;

class Sequence
{
  public:
    Sequence();
    bool empty() const;
    int size() const;
    bool insert(int pos, const ItemType& value);
    int insert(const ItemType& value);
    bool erase(int pos);
    int remove(const ItemType& value);
    bool get(int pos, ItemType& value) const;
    bool set(int pos, const ItemType& value);
    int find(const ItemType& value) const;
    void swap(Sequence& other);
};
```

In problem 3 of Homework 1, you implemented this interface using an array. For this project, implement this Sequence interface using a doubly-linked list. (You must not use the `list` class template from the C++ library.)

For the array implementation of problem 3 of Homework 1, since you declared no destructor, copy constructor, or assignment operator, the compiler wrote them for you, and they did the right thing. For this linked list implementation, if you let the compiler write the destructor, copy constructor, and assignment operator, they will do the wrong thing, so you will have to declare and implement these public member functions as well:

Destructor

When a Sequence is destroyed, the nodes in the linked list must be deallocated.

Copy constructor

When a brand new Sequence is created as a copy of an existing Sequence, enough new nodes must be allocated to hold a duplicate of the original list.

Assignment operator

When an existing Sequence (the left-hand side) is assigned the value of another Sequence (the right-hand side), the result must be that the left-hand side object is a duplicate of the right-hand side object, with no memory leak of list nodes (i.e. no list node from the old value of the left-hand side should be still allocated yet inaccessible).

Notice that there is now no *a priori* limit on the maximum number of items in the Sequence (so the one-argument form of `insert` should never return −1). Also, as in Homework 1, if a Sequence has a size of *n*, then the values of the first parameter to `get` for which that function retrieves an item (that was previously inserted by a call to one of the `insert` member functions) and returns true are *0*, *1*, *2*, ..., *n−1*; for other values, it returns false without setting its second parameter.

Another requirement is that as in Problem 5 of Homework 1, the number of statement executions when swapping two sequences must be the same no matter how many items are in the sequences.

Implement some sequence algorithms

Implement the following two functions. Notice that they are *non-member* functions: They are *not* members of Sequence or any other class, so they must *not* access *private* members of Sequence.

```
int subsequence(const Sequence& seq1, const Sequence& seq2);
```

Consider all the items in `seq2`; let's call them $seq2_0$, $seq2_1$, ..., $seq2_n$. If there exists at least one k such that $seq1_k == seq2_0$ and $seq1_{k+1} == seq2_1$ and ... and $seq1_{k+n} == seq2_n$, and $k+n < seq1.size()$, then this function returns the smallest such k. (In other words, if `seq2` is a consecutive subsequence of `seq1`, the function returns the earliest place in `seq1` where that subsequence starts.) If no such k exists or if `seq2` is empty, the function returns -1. For example, if `seq1` were a sequence of `int`s consisting of

```
30 21 63 42 17 63 17 29 8 32
```

and `seq2` consists of

```
63 17 29
```

then the function returns 5, since the consecutive items 63 17 29 appear in `seq1` starting with the 63 at position 5. (The result is not 2, because while there's a 63 at position 2, followed eventually by a 17 and a 29, those items are not *consecutive* in `seq1`.) With the same `seq1`, if `seq2`consists of

```
17 63 29
```

then the function returns -1.

```
void interleave(const Sequence& seq1, const Sequence& seq2, Sequence&
result);
```

This function produces as a result a sequence that consists of the first item in `seq1`, then the first in `seq2`, then the second in `seq1`, then the second in `seq2`, etc.

More formally: Consider the items in `seq1`; let's call them $seq1_0$, $seq1_1$, …, $seq1_m$, and let's call the items `seq2` contains $seq2_0$, $seq2_1$, …,$seq2_n$. If *m* equals *n*, then when this function returns, `result` must consist of the sequence $seq1_0$, $seq2_0$, $seq1_1$, $seq2_1$, …, $seq1_m$, $seq2_n$, and no other items.

If *m* is less than *n*, then when this function returns, `result` must consist of the sequence $seq1_0$, $seq2_0$, $seq1_1$, $seq2_1$, …, $seq1_m$, $seq2_m$,$seq2_{m+1}$, $seq2_{m+2}$, …, $seq2_n$, and no other items.

If *n* is less than *m*, then when this function returns, `result` must consist of the sequence $seq1_0$, $seq2_0$, $seq1_1$, $seq2_1$, …, $seq1_n$, $seq2_n$, $seq1_{n+1}$, $seq1_{n+2}$, …, $seq1_m$, and no other items.

If either sequence is empty, then when this function returns, `result` must consist of a copy of the other sequence.

When this function returns, `result` must not contain any items that this spec does not require it to contain. (You must *not* assume `result` is empty when it is passed in to this function; it might not be.)

As an example, if `seq1` were a sequence of `int`s consisting of

```
30 21 63 42 17 63
```

and `seq2` consists of

```
42 63 84 19
```

then no matter what value it had before, `result` ends up consisting of:

```
30 42 21 63 63 84 42 19 17 63
```

Be sure this function behaves correctly in the face of *aliasing*: What if `seq1` and `result` refer to the same Sequence, for example?

Other Requirements

Regardless of how much work you put into the assignment, your program will receive a zero for correctness if you violate these requirements:

- Your class definition, declarations for the two required non-member functions, and the implementations of any functions you choose to inline must be in a file named `Sequence.h`, which must have appropriate include guards. The implementations of the functions you declared in `Sequence.h` that you did not inline must be in a file named `Sequence.cpp`. Neither of those files may have a main routine (unless it's commented out). You may use a separate file for the main routine to test your Sequence class; you won't turn in that separate file.
- Except to add a destructor, copy constructor, assignment operator, and `dump` function (described below), you must not add functions to, delete functions from, or change the public interface of the Sequence class. You must not declare any additional struct/class outside the Sequence class, and you must not declare any *public* struct/class inside the Sequence class. You may add whatever private data members and private member functions you like, and you may declare *private* structs/classes inside the Sequence class if you like. The source files you submit for this project must not contain the word `friend` or the character `[` (open square bracket). You must not use any global variables whose values may be changed during execution.

  If you wish, you may add a public member function with the signature `void dump() const`. The intent of this function is that for your own testing purposes, you can call it to print information about the sequence; we will never call it. You do not have to add this function if you don't want to, but if you do add it, it must not make any changes to the sequence; if we were to replace your implementation of this function with one that simply returned immediately, your code must still work correctly. The `dump` function must not write to `cout`, but it's allowed to write to `cerr`.

- `Sequence.cpp` must not contain the word `string`. (`Sequence.h` may contain it only in the typedef statement, and must contain `#include <string>` if the typedef statement contains the word `string`.)
- Your code must build successfully (under both g32 and either Visual C++ or clang++) if linked with a file that contains a main routine.
- You must have an implementation for every member function of Sequence, as well as the non-member functions `subsequence` and `interleave`. Even if you can't get a function implemented correctly, it must have an implementation that at least builds successfully. For example, if you don't have time to correctly implement `Sequence::erase` or `interleave`, say, here are implementations that meet this requirement in that they at least build successfully:

```
bool Sequence::erase(int pos)
{
    return true;  // not correct, but at least this compiles
}

void interleave(const Sequence& seq1, const Sequence& seq2, Sequence&
result)
{
    // does nothing; not correct, but at least this compiles
}
```

You've probably met this requirement if the following file compiles and links with your code. (This uses magic beyond the scope of CS 32.)

```
#include "Sequence.h"
#include <type_traits>

#define CHECKTYPE(f, t) { auto p = (t)(f); (void)p; }

      static_assert(std::is_default_constructible<Sequence>::value,
              "Sequence must be default-constructible.");
      static_assert(std::is_copy_constructible<Sequence>::value,
              "Sequence must be copy-constructible.");

void thisFunctionWillNeverBeCalled()
{
    CHECKTYPE(&Sequence::operator=,  Sequence& (Sequence::*)(const
ItemType&));
    CHECKTYPE(&Sequence::empty,      bool (Sequence::*)() const);
    CHECKTYPE(&Sequence::size,       int  (Sequence::*)() const);
    CHECKTYPE(&Sequence::insert,     bool (Sequence::*)(int, const
ItemType&));
    CHECKTYPE(&Sequence::insert,     int (Sequence::*)(const
ItemType&));
    CHECKTYPE(&Sequence::erase,      bool (Sequence::*)(int));
    CHECKTYPE(&Sequence::remove,     int  (Sequence::*)(const
ItemType&));
    CHECKTYPE(&Sequence::get,   bool (Sequence::*)(int, ItemType&)
const);
```

```
      CHECKTYPE(&Sequence::set,   bool (Sequence::*)(int, const
ItemType&));
      CHECKTYPE(&Sequence::find,       int  (Sequence::*)(const
ItemType&) const);
      CHECKTYPE(&Sequence::swap,       void (Sequence::*)(Sequence&));
      CHECKTYPE(subsequence, int  (*)(const Sequence&, const
Sequence&));
      CHECKTYPE(interleave,  void (*)(const Sequence&, const Sequence&,
Sequence&));
  }

  int main()
  {}
```

- If you add `#include <string>` to `Sequence.h`, have Sequence's typedef specify `ItemType` as `std::string`, and link your code to a file containing

```
#include "Sequence.h"
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

void test()
{
    Sequence s;
    assert(s.insert(0, "lavash"));
    assert(s.insert(0, "tortilla"));
    assert(s.size() == 2);
    ItemType x = "injera";
    assert(s.get(0, x)  &&  x == "tortilla");
    assert(s.get(1, x)  &&  x == "lavash");
}

int main()
{
    test();
    cout << "Passed all tests" << endl;
}
```

  the linking must succeed. When the resulting executable is run, it must write `Passed all tests` to `cout` and nothing else to `cout`.

- If we successfully do the above, then make no changes to `Sequence.h` other than to change the typedef for Sequence so that `ItemType` specifies `unsigned long`, recompile `Sequence.cpp`, and link it to a file containing

```
#include "Sequence.h"
#include <iostream>
#include <cassert>
using namespace std;
```

```
void test()
{
    Sequence s;
    assert(s.insert(0, 10));
    assert(s.insert(0, 20));
    assert(s.size() == 2);
    ItemType x = 999;
    assert(s.get(0, x)  &&  x == 20);
    assert(s.get(1, x)  &&  x == 10);
}

int main()
{
    test();
    cout << "Passed all tests" << endl;
}
```

the linking must succeed. When the resulting executable is run, it must write `Passed all tests` to `cout` and nothing else to `cout`.

- During execution, if a client performs actions whose behavior is defined by this spec, your program must not perform any undefined actions, such as dereferencing a null or uninitialized pointer.
- Your code in `Sequence.h` and `Sequence.cpp` must not read anything from `cin` and must not write anything whatsoever to `cout`. If you want to print things out for debugging purposes, write to `cerr` instead of `cout`. `cerr` is the standard error destination; items written to it by default go to the screen. When we test your program, we will cause everything written to `cerr` to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

Turn it in

By Monday, January 30, there will be a link on the class webpage that will enable you to turn in your source files and report. You will turn in a zip file containing these three files:

- `Sequence.h`. When you turn in this file, the typedef must specify `std::string` as the `ItemType`.
- `Sequence.cpp`. Function implementations should be appropriately commented to guide a reader of the code.
- `report.doc` or `report.docx` (in Microsoft Word format) or `report.txt` (an ordinary text file) that contains:

- a description of the design of your doubly-linked list implementation. (A couple of sentences will probably suffice, perhaps with a picture of a typical Sequence and an empty Sequence. Is the list circular? Does it have a dummy node? What's in your list nodes?)
- pseudocode for non-trivial algorithms (e.g., `Sequence::remove` and `interleave`).
- a list of test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. For example, here's the beginning of a presentation in the form of code:

  The tests were performed on a sequence of strings (i.e., the `ItemType` typedef specified `std::string`).

```
  // default constructor
Sequence s;
  // For an empty sequence:
assert(s.size() == 0);              // test size
assert(s.empty());                  // test empty
assert(s.remove("paratha") == 0);  // nothing to remove
```

  Even if you do not correctly implement all the functions, you must still list test cases that would test them. Don't lose points by thinking "Well, I didn't implement this function, so I won't bother saying how I would have tested it if I *had* implemented it."