Spring 2018 CS 32

Homework 1

Time due: 11:00 PM Tuesday, April 17

Here is a C++ class definition for an abstract data type **Sequence of `string`s**, representing the concept of, well, a sequence of strings. When we talk about the position of an item in the sequence, we start at position zero. For example, in the three-item sequence `"lavash" "tortilla" "injera"`, the string at position 1 is `"tortilla"`.

```
class Sequence
{
  public:
    Sequence();    // Create an empty sequence (i.e., one with no items)
    bool empty();  // Return true if the sequence is empty, otherwise false.
    int size();    // Return the number of items in the sequence.
    bool insert(int pos, const std::string& value);
      // Insert value into the sequence so that it becomes the item at
      // position pos.  The original item at position pos and those that
      // follow it end up at positions one higher than they were at before.
      // Return true if 0 <= pos <= size() and the value could be
      // inserted.  (It might not be, if the sequence has a fixed capacity,
      // e.g., because it's implemented using a fixed-size array.)  Otherwise,
      // leave the sequence unchanged and return false.  Notice that
      // if pos is equal to size(), the value is inserted at the end.

    int insert(const std::string& value);
      // Let p be the smallest integer such that value <= the item at
      // position p in the sequence; if no such item exists (i.e.,
      // value > all items in the sequence), let p be size().  Insert
      // value into the sequence so that it becomes the item at position
      // p.  The original item at position p and those that follow it end
      // up at positions one higher than before.  Return p if the value
      // was actually inserted.  Return -1 if the value was not inserted
      // (perhaps because the sequence has a fixed capacity and is full).

    bool erase(int pos);
      // If 0 <= pos < size(), remove the item at position pos from
      // the sequence (so that all items that followed that item end up at
      // positions one lower than they were at before), and return true.
      // Otherwise, leave the sequence unchanged and return false.

    int remove(const std::string& value);
      // Erase all items from the sequence that == value.  Return the
      // number of items removed (which will be 0 if no item == value).

    bool get(int pos, std::string& value);
      // If 0 <= pos < size(), copy into value the item at position pos
      // in the sequence and return true.  Otherwise, leave value unchanged
      // and return false.

    bool set(int pos, const std::string& value);
      // If 0 <= pos < size(), replace the item at position pos in the
      // sequence with value and return true.  Otherwise, leave the sequence
      // unchanged and return false.

    int find(const std::string& value);
      // Let p be the smallest integer such that value == the item at
      // position p in the sequence; if no such item exists, let p be -1.
```

```
      // Return p.

    void swap(Sequence& other);
      // Exchange the contents of this sequence with the other one.
};
```

(When we don't want a function to change a parameter representing a value of the type stored in the sequence, we pass that parameter by constant reference. Passing it by value would have been perfectly fine for this problem, but we chose the const reference alternative because that will be more suitable after we make some generalizations in a later problem.)

Here's an example of the `remove` function for a Sequence of strings:

```
Sequence s;
s.insert(0, "a");
s.insert(1, "b");
s.insert(2, "c");
s.insert(3, "b");
s.insert(4, "e");
assert(s.remove("b") == 2);
assert(s.size() == 3);
string x;
assert(s.get(0, x)  &&  x == "a");
assert(s.get(1, x)  &&  x == "c");
assert(s.get(2, x)  &&  x == "e");
```

Here's an example of the `swap` function:

```
Sequence s1;
s1.insert(0, "paratha");
s1.insert(0, "focaccia");
Sequence s2;
s2.insert(0, "roti");
s1.swap(s2);
assert(s1.size() == 1  &&  s1.find("roti") == 0  &&  s2.size() == 2  &&
            s2.find("focaccia") == 0  &&  s2.find("paratha") == 1);
```

Notice that the empty string is just as good a string as any other; you should not treat it in any special way:

```
Sequence s;
s.insert(0, "dosa");
s.insert(1, "pita");
s.insert(2, "");
s.insert(3, "matzo");
assert(s.find("") == 2);
s.remove("dosa");
assert(s.size() == 3  &&  s.find("pita") == 0  &&  s.find("") == 1  &&
            s.find("matzo") == 2);
```

When comparing items for `remove`, `find`, and the one-parameter `insert`, just use the comparison operators provided for the string type by the library: <, <=, ==, etc. These do case-sensitive comparisons, and that's fine.

Here is what you are to do:

1. Determine which member functions of the Sequence class should be const member functions (because they do not modify the Sequence), and change the class declaration

accordingly.

2. As defined above, the Sequence class allows the client to use a sequence that contains only `std::string`s. Someone who wanted to modify the class to contain items of another type, such as only `int`s or only `double`s, would have to make changes in many places. Modify the class definition you produced in the previous problem to use a type alias for all values wherever the original definition used a `std::string`. A *type alias* is a name that is a synonym for some type; here is an example:

```
  // The following line introduces the type alias Number as a synonym
  // for the type int; anywhere the code uses the name Number, it means
  // the type int.

using Number = int;

int main()
{
    Number total = 0;
    Number x;
    while (cin >> x)
        total += x;
    cout << total << endl;
}
```

   The advantage of using the type alias Number is that if we later wish to modify this code to sum a sequence of `long`s or of `double`s, we need make a change in only one place: the using statement introducing the type alias Number.

   (Aside: Prior to C++11 (and still usable now), the only way to introduce a type alias was to use a `typedef` statement, e.g.`typedef int Number;`. Appendix A.1.8 of the textbook describes `typedef`.)

   To make the grader's life easier, we'll require that everyone use the same synonym for their type alias: You must use the name `ItemType`, with exactly that spelling and case.

3. Now that you have defined an interface for a sequence class where the item type can be easily changed, implement the class and all its member functions in such a way that the items in a sequence are contained in a data member that is an array. (Notice we said an array, not a pointer. It's not until problem 5 of this homework that you'll deal with a dynamically allocated array.) A sequence must be able to hold a maximum of `DEFAULT_MAX_ITEMS` items, where

```
const int DEFAULT_MAX_ITEMS = 200;
```

   Test your class for a Sequence of `unsigned long`s. Place your class definition, non-inline function protoypes, and inline function definitions (if any) in a file named `Sequence.h`, and your non-inline function implementations (if any) in a file named`Sequence.cpp`. (If we haven't yet discussed inline, then if you haven't encountered the topic yourself, all your functions will be non-inline, which is fine.)

Except to add a `dump` function (described below), you must not add public data or function members to, delete functions from, or change the public interface of the Sequence class. You may add whatever private data members and private member functions you like.

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`.

Your implementation of the Sequence class must be such that the compiler-generated destructor, copy constructor, and assignment operator do the right things. Write a test program named `testSequence.cpp` to make sure your Sequence class implementation works properly. Here is one possible (incomplete) test program:

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

int main()
{
    Sequence s;
    assert(s.empty());
    assert(s.find(42) == -1);
    assert(s.insert(42) == 0);
    assert(s.size() == 1  &&  s.find(42) == 0);
    cout << "Passed all tests" << endl;
}
```

Now change (only) the type alias in `Sequence.h` so that the Sequence will now contain `std::string`s. Except to add`#include <string>` if necessary, make no other changes to `Sequence.h`, and make no changes to `Sequence.cpp`. Verify that your implementation builds correctly and works properly with this alternative main routine (which again, is not a complete test of correctness):

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

int main()
{
    Sequence s;
    assert(s.empty());
    assert(s.find("laobing") == -1);
    assert(s.insert("laobing") == 0);
    assert(s.size() == 1  &&  s.find("laobing") == 0);
    cout << "Passed all tests" << endl;
}
```

You may need to flip back and forth a few times to fix your `Sequence.h` and `Sequence.cpp` code so that the *only* change to those files you'd need to

make to change a sequence's item type is to the type alias in `Sequence.h`. (When you turn in the project, have the type alias in `Sequence.h` specify the item type to be `unsigned long`.)

Except in a using statement in `Sequence.h`, the words `unsigned` and `long` must not appear in `Sequence.h` or `Sequence.cpp`. Except in a using statement introducing a type alias and in the context of `#include <string>` in `Sequence.h`, the word `string`must not appear in `Sequence.h` or`Sequence.cpp`.

(Implementation note: The `swap` function is easily implementable without creating any additional array or additional Sequence.)

4. Now that you've implemented the class, write some client code that uses it. We might want a class that records a Sequence of CS 32 project scores. Implement the following class that uses a Sequence of `unsigned long`s:

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

const unsigned long NO_SCORE = std::numeric_limits<unsigned long>::max();

class ScoreList
{
  public:
    ScoreList();       // Create an empty score list.

    bool add(unsigned long score);
      // If the score is valid (a value from 0 to 100) and the score list
      // has room for it, add it to the score list and return true.
      // Otherwise, leave the score list unchanged and return false.

    bool remove(unsigned long score);
      // Remove one instance of the specified score from the score list.
      // Return true if a score was removed; otherwise false.

    int size() const;  // Return the number of scores in the list.

    unsigned long minimum() const;
      // Return the lowest score in the score list.  If the list is
      // empty, return NO_SCORE.

    unsigned long maximum() const;
      // Return the highest score in the score list.  If the list is
      // empty, return NO_SCORE.

  private:
    // Some of your code goes here.
};
```

Your ScoreList implementation must employ a data member of type Sequence that uses the type alias `ItemType` as a synonym for `unsigned long`. (Notice we said a member of type *Sequence*, not of type *pointer to Sequence*.) Except to change one line (the type alias in Sequence.h), you must not make any changes to the `Sequence.h` and `Sequence.cpp` files you produced for Problem 3, so you must not add any member functions to the Sequence class. Each of the member functions `add`, `remove`, `size`, `minimum`, and `maximum` must delegate as much of the work that they need to do as they can to Sequence member functions. (In other words, they must not do work themselves that they can ask Sequence member

functions to do.) If the compiler-generated destructor, copy constructor, and assignment operator for ScoreList don't do the right thing, declare and implement them. Write a program to test your ScoreList class. Name your files `ScoreList.h`, `ScoreList.cpp`, and `testScoreList.cpp`.

The words `for` and `while` must not appear in `ScoreList.h` or `ScoreList.cpp`, except in the implementations of `ScoreList::minimum` and `ScoreList::maximum` if you wish. The characters `[` (open square bracket) and `*` must not appear in `ScoreList.h` or `ScoreList.cpp` outside of comments. You do not have to change `unsigned long` to `ItemType` in `ScoreList.h` and `ScoreList.cpp` if you don't want to (since unlike Sequence, which is designed for a wide variety of item types, ScoreList is specifically designed to work with unsigned longs). In the code you turn in, `ScoreList`'s member functions must not call `Sequence::dump`.

5. Now that you've created a sequence type based on arrays whose size is fixed at compile time, let's change the implementation to use a *dynamically allocated* array of objects. Copy the three files you produced for problem 3, naming the new files `newSequence.h`, `newSequence.cpp`, and `testnewSequence.cpp`. Update those files by either adding another constructor or modifying your existing constructor so that a client can do the following:

```
Sequence a(1000);   // a can hold at most 1000 items
Sequence b(5);      // b can hold at most 5 items
Sequence c;         // c can hold at most DEFAULT_MAX_ITEMS items
ItemType v = some value of the appropriate type;

  // No failures inserting 5 items into b
for (int k = 0; k < 5; k++)
    assert(b.insert(v) != -1);

  // Failure if we try to insert a sixth item into b
assert(b.insert(v) == -1);

  // When two Sequences' contents are swapped, their capacities are
  // swapped as well:
a.swap(b);
assert(a.insert(v) == -1  &&  b.insert(v) != -1);
```

Since the compiler-generated destructor, copy constructor, and assignment operator no longer do the right thing, declare them (as public members) and implement them. Make no other changes to the public interface of your class. (You are free to make changes to the private members and to the implementations of the member functions, and you may add or remove private members.) Change the implementation of the `swap` function so that the number of statement executions when swapping two sequences is the same no matter how many items are in the sequences. (You would not satisfy this requirement if, for example, your swap function caused a loop to visit each item in the sequences, since the number of statements executed by all the iterations of the loop would depend on the number of items in the sequences.)

The character `[` (open square bracket) must not appear in `newSequence.h` (but is fine in `newSequence.cpp`).

Test your new implementation of the Sequence class. (Notice that even though the file is named `newSequence.h`, the name of the class defined therein must still be `Sequence`.)

Verify that your ScoreList class still works properly with this new version of Sequence. You should not need to change your ScoreList class or its implementation in any way, other than to #include "newSequence.h" instead of "Sequence.h". (For this test, be sure you link with `newSequence.h`, not `Sequence.h`.) (Before you turn in `ScoreList.h` and `ScoreList.cpp`, be sure restore any #includes to "Sequence.h" instead of "newSequence.h".)

Turn it in

By Monday, April 16, there will be a link on the class webpage that will enable you to turn in this homework. Turn in one zip file that contains your solutions to the homework problems. (Since problem 3 builds on problems 1 and 2, you will not turn in separate code for problems 1 and 2.) If you solve every problem, the zip file you turn in will have nine files (three for each of problems 3, 4, and 5). The files *must* meet these requirements, or your score on this homework will be severely reduced:

- Each of the header files `Sequence.h`, `ScoreList.h`, and `newSequence.h` must have an appropriate include guard. In the files you turn in, the using statement in `Sequence.h` and `newSequence.h` must introduce `ItemType` as a type alias for `unsigned long`.
- If we create a project consisting of `Sequence.h`, `Sequence.cpp`, and `testSequence.cpp`, it must build successfully under both g32 and either Visual C++ or clang++. (Note: To build an executable using g32 from some, but not all, of the .cpp files in a directory, you list the .cpp files to use in the command. To build an executable named `req1` for this requirement, for example, you'd say `g32 -o req1 Sequence.cpp testSequence.cpp`.)
- If we create a project consisting of `Sequence.h`, `Sequence.cpp`, `ScoreList.h`,`ScoreList.cpp`, and `testScoreList.cpp`, it must build successfully under both g32 and either Visual C++ or clang++.
- If we create a project consisting of `newSequence.h`, `newSequence.cpp`, and `testnewSequence.cpp`, it must build successfully under both g32 and either Visual C++ or clang++.
- If we create a project consisting of `newSequence.h`, `newSequence.cpp`, and `testSequence.cpp`, where in `testSequence.cpp` we change only the `#include "Sequence.h"` to `#include "newSequence.h"`, the project must build successfully under both g32 and either Visual C++ or clang++. (If you try this, be sure to change the `#include` back to "Sequence.h" before you turn in `testSequence.cpp`.)
- The source files you submit for this homework must not contain the word `friend` or `vector`, and must not contain any global variables whose values may be changed during execution. (Global *constants* are fine.)

- No files other than those whose names begin with `test` may contain code that reads anything from `cin` or writes anything to `cout`, except that for problem 5, the implementation of the constructor that takes an integer parameter may write a message and exit the program if the integer is negative. Any file may write to`cerr` (perhaps for debugging purposes); we will ignore any output written to `cerr`.

- You must have an implementation for every member function of Sequence and ScoreList. If you can't get a function implemented correctly, its implementation must at least build successfully. For example, if you don't have time to correctly implement `Sequence::erase` or`Sequence::swap`, say, here are implementations that meet this requirement in that they at least allow programs to build successfully even though they might execute incorrectly:

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

void Sequence::swap(Sequence& other)
{
    // does nothing; not correct, but at least this compiles
}
```

- If we add `#include <string>` to your `Sequence.h`, change the type alias for the Sequence's item type to specify`std::string` as the item type, make no change to your `Sequence.cpp`, compile your `Sequence.cpp`, and link it 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` and nothing more to `cout`and terminate normally.

- If we successfully do the above, then change the type alias for the Sequence's item type to specify `unsigned long` as the item type without making any other changes, 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` and nothing more to `cout`and terminate normally.

- If we add `#include <string>` to your `newSequence.h`, change the type alias for the Sequence's item type to specify`std::string` as the item type, make no change to your `newSequence.cpp`, compile your `newSequence.cpp`, and link it to a file containing

```
#include "newSequence.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` and nothing more to `cout`and terminate normally.

- If we successfully do the above, then change the type alias for the Sequence's item type to specify `unsigned long` as the item type without making any other changes, recompile `newSequence.cpp`, and link it to a file containing

```
#include "newSequence.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` and nothing more to `cout`and terminate normally.

- During execution, your program must not perform any undefined actions, such as accessing an array element out of bounds, or dereferencing a null or uninitialized pointer.

Notice that we are not requiring any particular content in `testSequence.cpp`, `testScoreList.cpp`, and `testnewSequence.cpp`, as long as they meet the requirements above. Of course, the intention is that you'd use those files for the test code that you'd write to convince yourself that your implementations are correct. Although we will throughly evaluate your implementations for correctness, for homeworks, unlike for projects, we will not grade the thoroughness of your test cases. Incidentally, for homeworks, unlike for projects, we will also not grade your program commenting.