Programming Assignment 2

Double Feature

Time due: 11:00 PM Tuesday, April 20

Homework 1 gave you extensive experience with the Set type using both arrays and dynamically-allocated arrays. In this project, you will re-write the implementation of the Set 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 sets.

Implement Set yet again

Consider the Set interface from problem 2 of Homework 1:

```
using ItemType = TheTypeOfElementGoesHere;

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

In problem 3 of Homework 1, you implemented this interface using an array. For this project, implement this Set 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 Set is destroyed, the nodes in the linked list must be deallocated.
Copy constructor
When a brand new Set is created as a copy of an existing Set, enough new nodes must be allocated to hold a duplicate of the original list.
Assignment operator
When an existing Set (the left-hand side) is assigned the value of another Set (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 Set. Notice also that, as in Homework 1, if a Set 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 insert) and returns true are *0*, *1*, *2*, …, *n−1*; for other values, it returns false without setting its second parameter. For example:

```
Set ss;  // ItemType is std::string
 ss.insert("ccc");
 ss.insert("aaa");
 ss.insert("bbb");
 ItemType x = "xxx";
 assert(!ss.get(3, x)  &&  x == "xxx");  // x is unchanged
 assert(ss.get(1, x)  &&  x == "bbb");   // "bbb" is less than
                                         // exactly 1 item
```

This is the same visible behavior as in Homework 1.

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

Implement some set algorithms

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

```
void unite(const Set& s1, const Set& s2, Set& result);
```

When this function returns, result must contain one copy of each of the values that appear in s1 or s2 or both, and must not contain any other values. (You must *not* assume result is empty when it is passed in to this function; it might not be.) Since result is a Set, it must, of course, not have any duplicates. For example, if s1 were a set of ints into which the values

```
2 8 3 9 5
```

have been inserted, and s2 had the values

    6 3 8 5 10

inserted, then no matter what value it had before, result must end up as a set containing these values and no others (not necessarily in this order):

    9 3 6 5 10 2 8

void difference(const Set& s1, const Set& s2, Set& result);

When this function returns, result must contain one copy of each of the values that appear in s1 or s2 **but not both**, and must not contain any other values. (You must *not* assume result is empty when it is passed in to this function; it might not be.) For example, if s1 and s2 were as in the example above, result must end up as a set containing these values and no others (not necessarily in this order):

    9 6 2 10

(Here's a use for difference: Given a set of id numbers of students who are enrolled in this class and a set of id numbers of students who turned in Homework 1, how would you get a set containing both enrolled students who didn't turn in the homework and students who turned in the homework but are not enrolled, perhaps because they dropped the class after turning in the homework?)

Be sure that in the face of *aliasing*, these functions behave as this spec requires: Does your implementation work correctly if s1 and result refer to the same Set, 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 Set.h, which must have appropriate include guards. The implementations of the functions you declared in Set.h that you did not inline must be in a file named Set.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 Set 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 Set class. You must not declare any additional struct/class outside

the Set class, and you must not declare any *public* struct/class inside the Set class. You may add whatever private data members and private member functions you like, and you may declare *private* structs/classes inside the Set class if 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 set; 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 set; 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.

- The source files you submit for this project must not contain the word friend or pragma or vector or the character [(open square bracket). You must not use any global variables whose values may be changed during execution. (Global *constants* are fine.)
- Set.cpp must not contain the word string. (Set.h may contain it only in a using statement introducing a type alias, and must contain #include <string> if a using statement introducing a type alias 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 Set, as well as the non-member functions unite and difference. 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 Set::erase or difference, say, here are implementations that meet this requirement in that they at least build successfully:

```
bool Set::erase(const ItemType& value)
{
    return false;  // not correct, but at least this compiles
}

void difference(const Set& s1, const Set& s2, Set& 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 "Set.h"
#include <type_traits>

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

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

void thisFunctionWillNeverBeCalled()
{
    CHECKTYPE(&Set::empty,     bool (Set::*)() const);
    CHECKTYPE(&Set::size,      int  (Set::*)() const);
    CHECKTYPE(&Set::insert,    bool (Set::*)(const ItemType&));
    CHECKTYPE(&Set::erase,     bool (Set::*)(const ItemType&));
    CHECKTYPE(&Set::contains,  bool (Set::*)(const ItemType&) const);
    CHECKTYPE(&Set::get,       bool (Set::*)(int, ItemType&) const);
    CHECKTYPE(&Set::swap,      void (Set::*)(Set&));
    CHECKTYPE(unite,      void (*)(const Set&, const Set&, Set&));
    CHECKTYPE(difference, void (*)(const Set&, const Set&, Set&));
}

int main()
{}
```

-

If you add #include <string> to Set.h and have the type alias for ItemType specify std::string, then if we make no change to your Set.cpp, compile it, and link it to a file containing

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

void test()
{
    Set ss;
    assert(ss.insert("pita"));
    assert(ss.insert("roti"));
    assert(ss.size() == 2);
    assert(ss.contains("roti"));
    ItemType x = "laobing";
    assert(ss.get(0, x)  &&  x == "roti");
    assert(ss.get(1, x)  &&  x == "pita");
}

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, and terminate normally.

If we successfully do the above, then make no changes to Set.h other than to change the type alias so that ItemType specifies unsigned long, recompile Set.cpp, and link it to a file containing

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

void test()
{
    Set uls;
    assert(uls.insert(10));
    assert(uls.insert(20));
    assert(uls.size() == 2);
    assert(uls.contains(20));
    ItemType x = 30;
    assert(uls.get(0, x)  &&  x == 20);
    assert(uls.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, and terminate normally.
- 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 Set.h and Set.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, April 19, 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:

- Set.h. When you turn in this file, the using statement must specify ItemType as a type alias for std::string.
- Set.cpp. Function implementations should be appropriately commented to guide a reader of the code.
- report.docx or report.doc (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 Set and an empty Set. Is the list circular? Does it have a dummy node? What's in your list nodes? Are they in any particular order?)
    - pseudocode for non-trivial algorithms (e.g., difference).

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 set of strings (i.e., ItemType was a type alias for std::string).

```
  // default constructor
Set ss;
  // For an empty set:
assert(ss.size() == 0);       // test size
assert(ss.empty());           // test empty
assert(!ss.erase("roti"));    // 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."