**Homework 1**

**Time due: 9:00 PM Tuesday, April 17**

Here is a C++ class definition for an abstract data type **Set of string**s, representing the concept of an unordered collection of strings, without duplicates. (A web server might record unique visitors to a website in a set of strings, for example.) To make things simpler for you, the case of letters in a string matters, so that the strings Roti and rOtI are *not* considered duplicates.

class Set

{

public:

Set(); // Create an empty set.

bool empty(); // Return true if the set is empty, otherwise false.

int size(); // Return the number of items in the set.

bool insert(const std::string& value);

// Insert value into the set if it is not already present. Return

// true if the value was actually inserted. Return false if the

// value was not inserted (perhaps because it is already in the set

// or because the set has a fixed capacity and is full).

bool erase(const std::string& value);

// Remove the value from the set if present. Return true if the

// value was removed; otherwise, leave the set unchanged and

// return false.

bool contains(const std::string& value);

// Return true if the value is in the set, otherwise false.

bool get(int i, std::string& value);

// If 0 <= i < size(), copy into value an item in the set and

// return true. Otherwise, leave value unchanged and return false.

// (See below for details about this function.)

void swap(Set& other);

// Exchange the contents of this set with the other one.

};

(When we don't want a function to change a key or value parameter, 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.)

The get function enables a client to iterate over all elements of a Set because of this property it must have: If nothing is inserted into or erased from the set in the interim, then calling get repeatedly with the first parameter ranging over all the integers from 0 to size()-1 inclusive will copy into the second parameter every value from the set exactly once. The order in which elements are copied is up to you. In other words, this code fragment

Set ss;

ss.insert("A");

ss.insert("C");

ss.insert("A");

ss.insert("B");

string all;

for (int k = 0; k < ss.size(); k++)

{

string x;

ss.get(k, x);

all += x;

}

must result in the string all having exactly one of the values ABC, ACB, BAC, BCA, CAB, or CBA, and the client can't depend on it being any particular one of those six. If the set is modified between successive calls to get, all bets are off as to whether a particular value is returned exactly once.

If nothing is inserted into or erased from the set in the interim, then calling get repeatedly with the same value for the first parameter each time must copy the same value into the second parameter each time, so that this code is fine:

Set ss;

ss.insert("pita");

ss.insert("roti");

string s1;

assert(ss.get(1,s1) && (s1 == "roti" || s1 == "pita"));

string s2;

assert(ss.get(1,s2) && s2 == s1);

The empty string is just as good a string as any other; it should not be treated in any special way:

Set ss;

ss.insert("focaccia");

assert(!ss.contains(""));

ss.insert("tortilla");

ss.insert("");

ss.insert("lavash");

assert(ss.contains(""));

ss.erase("focaccia");

assert(ss.size() == 3 && ss.contains("lavash") && ss.contains("tortilla") &&

ss.contains(""));

Here's an example of the swap function:

Set ss1;

ss1.insert("injera");

Set ss2;

ss2.insert("matzo");

ss2.insert("dosa");

ss1.swap(ss2);

assert(ss1.size() == 2 && ss1.contains("matzo") && ss1.contains("dosa") &&

ss2.size() == 1 && ss2.contains("injera"));

When comparing items for insert, erase, and contains, just use the == or != operators provided for the string type by the library. These do case-sensitive comparisons, and that's fine.

Here is what you are to do:

1. Determine which member functions of the Set class should be const member functions (because they do not modify the Set), and change the class declaration accordingly.
2. As defined above, the Set class allows the client to use a set that contains only strings. Someone who wanted to modify the class to contain items of another type, such as only ints or only doubles, would have to make changes in many places. Modify the class definition you produced in the previous problem to use a typedef-defined type for all values wherever the original definition used a std::string. Here's an example of a use of typedef:
3. typedef int Number; // define Number as a synonym for int
4. int main()
5. {
6. Number total = 0;
7. Number x;
8. while (cin >> x)
9. total += x;
10. cout << total << endl;
11. }

To modify this code to sum a sequence of longs or of doubles, we need make a change in only one place: the typedef.

You may find the example using typedef starting at the bottom of p. 156 of the textbook useful.

To make the grader's life easier, we'll require that everyone use the same synonym as their typedef-defined name: You must use the name ItemType, with exactly that spelling and case.

1. Now that you have defined an interface for a set 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 set 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 set must be able to hold a maximum of DEFAULT\_MAX\_ITEMS items, where
2. const int DEFAULT\_MAX\_ITEMS = 200;

Test your class for a Set of std::strings. Place your class definition and inline function definitions (if any) in a file named Set.h, and your non-inline function definitions (if any) in a file named Set.cpp. You may add any private data members or private member functions that you like, but you must not add anything to or delete anything from the public interface you defined in the previous problem, nor may you change the function signatures. Your implementation must be such that the compiler-generated destructor, copy constructor, and assignment operator do the right things. Write a test program named testSet.cpp to make sure your Set class implementation works properly. Here is one possible (incomplete) test program:

#include "Set.h"

#include <iostream>

#include <string>

#include <cassert>

using namespace std;

int main()

{

Set s;

assert(s.empty());

string x = "arepa";

assert( !s.get(42,x) && x == "arepa"); // x unchanged by get failure

s.insert("chapati");

assert(s.size() == 1);

assert(s.get(0,x) && x == "chapati");

cout << "Passed all tests" << endl;

}

Now change (only) the typedef in Set.h so that the Set will contain unsigned longs. Make no other changes to Set.h, and make no changes to Set.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 "Set.h"

#include <iostream>

#include <cassert>

using namespace std;

int main()

{

Set s;

assert(s.empty());

unsigned long x = 9876543;

assert( !s.get(42,x) && x == 9876543); // x unchanged by get failure

s.insert(123456789);

assert(s.size() == 1);

assert(s.get(0,x) && x == 123456789);

cout << "Passed all tests" << endl;

}

You may need to flip back and forth a few times to fix your Set.h and Set.cpp code so that the *only* change to those files you'd need to make to change a set's item type is to the typedef in Set.h. (When you turn in the project, have it so that the item type is unsigned long.)

Except in the typedef statement in Set.h, the words unsigned and long must not appear in Set.h or Set.cpp. Except in the context of #include <string>, the word string must not appear in Set.h or Set.cpp.

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

1. Now that you've implemented the class, write some client code that uses it. We might want a class that records all students who submit a project. Students may make more than one submission, but we'll include them only once. Implement the following class that uses a Set of unsigned longs:
2. #include "Set.h"
3. class StudentSet
4. {
5. public:
6. StudentSet(); // Create an empty student set.
7. bool add(unsigned long id);
8. // Add a student id to the StudentSet. Return true if and only if the
9. // id was actually added.
10. int size() const; // Return the number of ids in the StudentSet.
11. void print() const;
12. // Print every student id in the StudentSet exactly once, one per line.
13. private:
14. // Some of your code goes here.
15. };

Your StudentSet implementation must employ a data member of type Set that uses the typedef ItemType as a synonym for unsigned long. (Notice we said a member of type *Set*, not of type *pointer to Set*.) Except for the typedef, you must not make any changes to the Set.h and Set.cpp files you produced for Problem 3, so you must not add any member functions to the Set class. Each of the member functions add, size, and print must delegate as much of the work that they need to do as they can to Set member functions. (In other words, they must not do work that they can ask Set member functions to do instead.) If the compiler-generated destructor, copy constructor, and assignment operator for StudentSet don't do the right thing, declare and implement them. Write a program to test your StudentSet class. Name your files StudentSet.h, StudentSet.cpp, and testStudentSet.cpp.

Except for the implementation of StudentSet::print, the words for and while must not appear in StudentSet.h or StudentSet.cpp. The word friend must not appear in any of the files you submit for this homework.

1. You're almost done. Now that you've created a set 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 newSet.h, newSet.cpp, and testnewSet.cpp. Update those files by either adding another constructor or modifying your existing constructor so that a client can do the following:
2. Set a(1000); // a can hold at most 1000 distinct items
3. Set b(5); // b can hold at most 5 distinct items
4. Set c; // c can hold at most DEFAULT\_MAX\_ITEMS distinct items
5. ItemType v[6] = { *six distinct values of the appropriate type* };
6. // No failures inserting 5 distinct items into b
7. for (int k = 0; k < 5; k++)
8. assert(b.insert(v[k]));
9. // Failure if we try to insert a sixth distinct item into b
10. assert(!b.insert(v[5]));
11. // When two Sets' contents are swapped, their capacities are swapped
12. // as well:
13. a.swap(b);
14. assert(!a.insert(v[5]));

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.) Change the implementation of the swap function so that the number of statement executions when swapping two sets is the same no matter how many items are in the sets. (You would not satisfy this requirement if, for example, your swap function looped over each item in a set, since the number of iterations of the loop would depend on the number of items in the set.)

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

Verify that your StudentSet class still works properly with this new version of Set. You should not need to change your StudentSet class or its implementation in any way, other than to #include "newSet.h" instead of "Set.h". (Before you turn in StudentSet.h, be sure to restore the #include to "Set.h" instead of "newSet.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 Set.h, StudentSet.h, and newSet.h must have an appropriate include guard. In the files you turn in, the typedefs in Set.h and newSet.h must define ItemType to be a synonym for unsigned long.
* If we create a project consisting of Set.h, Set.cpp, and testSet.cpp, it must build successfully.
* If we create a project consisting of Set.h, Set.cpp, StudentSet.h, StudentSet.cpp, and testStudentSet.cpp, it must build successfully.
* If we create a project consisting of newSet.h, newSet.cpp, and testnewSet.cpp, it must build successfully.
* If we create a project consisting of newSet.h, newSet.cpp, and testSet.cpp, where in testSet.cpp we change only the #include "Set.h" to #include "newSet.h", the project must build successfully. (If you try this, be sure to change the #include back to "Set.h" before you turn in testSet.cpp.)
* 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 4, StudentSet::print must write to cout, and 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 Set and StudentSet. 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 Set::erase or Set::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 Set::erase(const ItemType& value)
* {
* return true; // not correct, but at least this compiles
* }
* void Set::swap(Set& other)
* {
* // does nothing; not correct, but at least this compiles
* }
* If we add #include <string> to your Set.h, change the typedef for the Set's item type to specify std::string as the item type, make no change to your Set.cpp, compile your Set.cpp, and link it to a file containing
* #include "Set.h"
* #include <string>
* #include <iostream>
* #include <cassert>
* using namespace std;
* void test()
* {
* Set ss;
* assert(ss.insert("roti"));
* assert(ss.insert("pita"));
* assert(ss.size() == 2);
* assert(ss.contains("pita"));
* ItemType x = "bing";
* assert(ss.get(0, x) && (x == "roti" || 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 and nothing more to cout and terminate normally.

* If we successfully do the above, then change the typedef for the Set's item type to specify unsigned long as the item type without making any other changes, 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 == 10 || x == 20));
* }
* 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 newSet.h, change the typedef for the Set's item type to specify std::string as the item type, make no change to your newSet.cpp, compile your newSet.cpp, and link it to a file containing
* #include "newSet.h"
* #include <string>
* #include <iostream>
* #include <cassert>
* using namespace std;
* void test()
* {
* Set ss;
* assert(ss.insert("roti"));
* assert(ss.insert("pita"));
* assert(ss.size() == 2);
* assert(ss.contains("pita"));
* ItemType x = "bing";
* assert(ss.get(0, x) && (x == "roti" || 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 and nothing more to cout and terminate normally.

* If we successfully do the above, then change the typedef for the Set's item type to specify unsigned long as the item type without making any other changes, recompile newSet.cpp, and link it to a file containing
* #include "newSet.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 == 10 || x == 20));
* }
* 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.