Spring 2017 CS 32

Homework 1

Time due: 11:00 PM Tuesday, April 18

Here is a C++ class definition for an abstract data type **Set of `string`**s, representing the concept of a 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.  Leave the set
unchanged
      // and 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 the item in the set that
is
      // greater than exactly i items in the set and return true.
Otherwise,
      // leave value unchanged and return false.

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

(When we don't want a function to change a parameter representing a value of the type stored in the set, 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.)

Notice that the comment for the `get` function implies that for a 5-item set ss, `ss.get(0, x)` will copy the smallest item in the set into x (because the smallest item is greater than 0 items in the set), and `ss.get(4, x)` will copy the largest item (because the largest item is greater than 4 items in the set):

```
    Set ss;
    ss.insert("lavash");
    ss.insert("roti");
    ss.insert("chapati");
    ss.insert("injera");
    ss.insert("roti");
    ss.insert("matzo");
    ss.insert("injera");
    assert(ss.size() == 5);  // duplicate "roti" and "injera" were not
added
    string x;
    ss.get(0, x);
    assert(x == "chapati");  // "chapati" is greater than exactly 0
items in ss
    ss.get(4, x);
    assert(x == "roti");  // "roti" is greater than exactly 4 items in
ss
    ss.get(2, x);
    assert(x == "lavash");  // "lavash" is greater than exactly 2 items
in ss
```

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

```
    Set ss;
    ss.insert("dosa");
    assert(!ss.contains(""));
    ss.insert("tortilla");
    ss.insert("");
    ss.insert("focaccia");
    assert(ss.contains(""));
    ss.erase("dosa");
    assert(ss.size() == 3  &&  ss.contains("focaccia")  &&
ss.contains("tortilla")  &&
                ss.contains(""));
    string v;
    assert(ss.get(1, v)  &&  v == "focaccia");
    assert(ss.get(0, v)  &&  v == "");
```

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

```
    Set ss1;
    ss1.insert("bing");
    Set ss2;
    ss2.insert("matzo");
    ss2.insert("pita");
    ss1.swap(ss2);
    assert(ss1.size() == 2  &&  ss1.contains("matzo")  &&
ss1.contains("pita")  &&
           ss2.size() == 1  &&  ss2.contains("bing"));
```

When comparing items for `insert`, `erase`, and `contains`, just use the `==`, `!=`, `>`, etc., 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 `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 `typedef`-defined type for all values wherever the original definition used a `std::string`. Here's an example of a use of `typedef`:

```
    typedef int Number;  // define Number as a synonym for int

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

   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 typedef.

   You may find the example using `typedef` in Appendix A.1.8 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.

13. 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

14.

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

Test your class for a Set of `std::string`s. Place your class definition and inline function implementations (if any) in a file named `Set.h`, and your non-inline function implementations (if any) in a file named `Set.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.)

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. There is one exception to this: 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`.

Your implementation of the Set class 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());
```

```
        ItemType 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 long`s. 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());
        ItemType 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 the typedef in `Set.h` specify the item type to be `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.)

15. Now that you've implemented the class, write some client code that uses it. We might want a class that records the social security numbers of all

people for whom our company has issued a required tax document. We might issue someone more than one tax document (e.g., one for each account they have), but we'll include their social security number (SSN) only once. Implement the following class that uses a Set of `unsigned long`s:

```
#include "Set.h"

class SSNSet
{
  public:
    SSNSet();          // Create an empty SSN set.

    bool add(unsigned long ssn);
      // Add an SSN to the SSNSet.  Return true if and only if the SSN
      // was actually added.

    int size() const;  // Return the number of SSNs in the SSNSet.

    void print() const;
      // Write every SSN in the SSNSet to cout exactly once, one per
      // line.  Write no other text.

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

Your SSNSet 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 to change one line (the typedef in `Set.h`), 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 or data members 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 themselves that they can ask Set member functions to do instead.) If the compiler-generated destructor, copy constructor, and assignment operator for SSNSet don't do the right thing, declare and implement them. Write a program to test your SSNSet class. Name your files `SSNSet.h`, `SSNSet.cpp`, and `testSSNSet.cpp`.

The words `for` and `while` must not appear in `SSNSet.h` or `SSNSet.cpp`, except in the implementation of `SSNSet::print` if you wish. The characters `[` (open square bracket) and `*` must not appear in `SSNSet.h` or `SSNSet.cpp`, except in comments if you wish. You do not have to change `unsigned long` to `ItemType` in `SSNSet.h` and `SSNSet.cpp` if you don't want to. In the code you turn in, `SSNSet`'s member functions must not call `Set::dump`.

36. 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:

```
    Set a(1000);   // a can hold at most 1000 distinct items
    Set b(5);      // b can hold at most 5 distinct items
    Set c;         // c can hold at most DEFAULT_MAX_ITEMS distinct items
    ItemType v[6] = { six distinct values of the appropriate type };

      // No failures inserting 5 distinct items into b
    for (int k = 0; k < 5; k++)
        assert(b.insert(v[k]));

      // Failure if we try to insert a sixth distinct item into b
    assert(!b.insert(v[5]));

      // When two Sets' contents are swapped, their capacities are swapped
      // as well:
    a.swap(b);
    assert(!a.insert(v[5])  &&  b.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, 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 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 caused a loop to visit each item in the set, since the number of statements executed by all the iterations of the loop would depend on the number of items in the set.)

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

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 SSNSet class still works properly with this new version of Set. You should not need to change your SSNSet class or implementation in any way, other than to include `"newSet.h"` instead of `"Set.h"`. (For this test, be sure to link with `newSet.cpp`, not `Set.cpp`.) (Before you turn in `SSNSet.h`, be sure to restore the include to `"Set.h"` instead of `"newSet.h"`.)

Turn it in

By Monday, April 17, 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`, `SSNSet.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 under both g32 and either Visual C++ or clang++.
- If we create a project consisting of `Set.h`, `Set.cpp`, `SSNSet.h`, `SSNSet.cpp`, and `testSSNSet.cpp`, it must build successfully under both g32 and either Visual C++ or clang++.
- If we create a project consisting of `newSet.h`, `newSet.cpp`, and `testnewSet.cpp`, it must build successfully under both g32 and either Visual C++ or clang++.
- 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 under both g32 and either Visual C++ or clang++. (If you try this, be sure to change the `#include` back to `"Set.h"` before you turn in `testSet.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 4, `SSNSet::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 SSNSet. 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
        }
```

- Given `Set.h` with the typedef for the Set's item type specifying `std::string`, if we make no change to your `Set.cpp`, then if we 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 == "pita");
        assert(ss.get(1, x)  &&  x == "roti");
    }

    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 in `Set.h` 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(20));
        assert(uls.insert(10));
        assert(uls.size() == 2);
        assert(uls.contains(10));
        ItemType x = 30;
        assert(uls.get(0, x)  &&  x == 10);
        assert(uls.get(1, x)  &&  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.

- Given `newSet.h` with the typedef for the Set's item type specifying `std::string`, if we make no change to your `newSet.cpp`, then if we 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 == "pita");
    assert(ss.get(1, x)  &&  x == "roti");
}

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 in `newSet.h` 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(20));
    assert(uls.insert(10));
    assert(uls.size() == 2);
    assert(uls.contains(10));
    ItemType x = 30;
    assert(uls.get(0, x)  &&  x == 10);
    assert(uls.get(1, x)  &&  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.

Notice that we are not requiring any particular content in `testSet.cpp`, `testSSNSet.cpp`, and `testnewSet.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.