CMSC 202 — Fall 2015 — Prof. Marron

Project 2

Due: Thursday, October 15, 2015 9:00PM

Objective
The objectives of this project are 1) to practice using C++ class syntax, 2) to practice using C++ `vector` and `string` classes.

Background
For this project, you will implement member functions of a C++ class that stores sets of strings. You will use the C++ `vector` class to store the set. (More on this below.) This class will represent not just finite sets of strings, but also some infinite sets of strings. In particular, objects in this class can represent *cofinite* sets. A set is cofinite if its complement is finite. For example, the subset of natural numbers { 1, 5, 7, 8, 9, 10, 11, ...} is cofinite because it is the complement of the set { 0, 2, 3, 4, 6}. In C++, we can represent a cofinite set by storing its complement and an additional boolean value to indicate that we really want the complement of the stored values.

It turns out that if two sets *A* and *B* are either finite or cofinite, then their union, intersection and complements must also be finite or cofinite. For this project, you will implement member functions that will create the union, intersection and complements of finite and cofinite sets of *strings*. You will also implement functions to compute the membership and subset relations.

Assignment
Your assignment is to implement the member functions of the class `SoS` (short for "Set of Strings") defined below. Each `SoS` object has two data members: `m_cofinite`, `m_vos`. The value of `m_cofinite` is `true` if the object represents a cofinite set. The `vector` object `m_vos` has the strings in the representation. These are either the strings that are in a finite set or the strings that are not in a cofinite set. (For now you can think of a `vector` as an array. They behave very much like lists in Python or ArrayLists in Java. Section 7.3 of the textbook has an introduction to vectors.) These should be the only data members you need. The requirements for each member function is described below.

```
class SoS {

   public:

   // Do not change the member function prototypes for
   // any public member function.

   // See documentation in Project 2 description.

   SoS() ;
```

```
    void insert(string str) ;
    void print() const ;

    bool isMember(string str) const ;
    bool isSubset(const SoS& B) const ;
    bool isFinite() const ;

    SoS makeComplement() const ;
    SoS makeUnion(const SoS& B) const ;
    SoS makeIntersection(const SoS& B) const ;


    // The function dump() is used for grading.  Do not modify!
    //
    vector<string>  dump() const { return m_vos; }


    private:

    // Do not change the types of these private data members

    bool m_cofinite ;
    vector<string> m_vos ;    // vos = vector of strings


    // Declarations for Additional private member functions
    // may be added below. Fully document these.



} ;
```

Here are the requirements for each member function.

- `SoS() ;`
  The default constructor should set up the host object as an empty set.
- `void insert(string str) ;`
  This function should add the string `str` to the set represented by the host object. You must not have duplicate values stored in `m_vos`. This is after all a *set*. Also, you may be inserting `str` into a cofinite set. In this case, you would want to remove `str` from `m_vos` if it is there. Although the `vector` class does support an `erase()` method, it is simpler to construct a new vector of strings that omits `str` and assign the new vector to `m_vos`.
- `void print() const ;`
  For finite sets, print out the strings in the set, one per line. For cofinite sets, print out "COMPLEMENT OF:" followed by the strings in `m_vos`, one per line.
- `bool isMember(string str) const ;`
  Return `true` if `str` is a member of the set represented by the host object. Return `false` otherwise. Do take care of the case where the host object is representing a cofinite set.
- `bool isSubset(const SoS& B) const ;`
  Return `true` if the host object represents a set that is a subset of the set represented by `B`. (From now on, we will just say "host object" and "`B`" instead of "the set represented by the host object" and "the set represented by `B`".) Note that `B` is passed as a `const` reference. You will have to take care of the four cases where the host object and `B` are finite and cofinite:

- If both sets are finite, then you can just check whether each member of the host object is a member of `B`.
- If the host object is cofinite and `B` is finite, then the answer is always `false` because an infinite set is never the subset of a finite set.
- If the host object is finite and `B` is cofinite, then you can still check whether each member of the host object is a member of `B`.
- Finally, if both sets are cofinite, then you have to consider the contrapositive of the subset relation: check that every non-member of `B` is a non-member of the host object. If this is true for every non-member of `B`, then you cannot have a member of the host object that is not in `B`. Thus, the host object is a subset of `B`. Conversely, if you find a non-member of `B` that *is* a member of the host object, then clearly the host object is not a subset of `B`. Fortunately, there are only a finite number of non-members of `B` and they are all stored in `m_vos`.

- `bool isFinite() const ;`
  Return `true` if the host object is a finite set and `false` otherwise.

- `SoS makeComplement() const ;`
  Return an `SoS` object that is the complement of the host object. Note that you cannot just invert `m_cofinite` because that does not return an object. Also, `makeComplement()` is a `const` member function, so you cannot modify `m_cofinite` anyway. So, you must make a copy of the host with `m_cofinite` flipped and return that. (Yes, it is OK to return a local object because it gets copied during the return.)

- `SoS makeUnion(const SoS& B) const ;`
  Return an `SoS` object that is the union of the host object and `B`. Again, you must create a new object. As with `isSubset()`, you need to consider the four cases where the host object and `B` are finite and cofinite. Draw lots of Venn diagrams. You should not have any duplicate strings stored in `m_vos` member of the new `SoS` object.

- `SoS makeIntersection(const SoS& B) const ;`
  Same as `makeUnion()` except this time you want the intersection. Again, draw lots of Venn diagrams. You should not have any duplicate strings stored in `m_vos` member of the new `SoS` object.

- `vector dump() const ;`
  The `dump()` member function is already implemented. So, you don't have to do anything. It returns `m_vos`, so we can check that your implementation is correct in the main program. Normally, classes should not have such a member function except for debugging purposes.

Implementation Issues

Here are some issues to consider and pitfalls to avoid. (You should read these before coding!)

- **Reminder:** do not develop your programs in the submission directories. Develop your programs in some other location in GL or on your own computer. The files that you have in the submission directory should not be your only copy of your code.

- You should not dynamically allocate any objects in your program. That is, if you are using `new` or `delete`, you are thinking about this wrong. The `vector` and `string` classes are well behaved, so we also do not need a destructor (the default destructor works fine) or a copy constructor or an overloaded assignment operator.

- You might find it useful to use `isMember()` in the implementation of the other member functions. However, remember that the return value of `isMember()` is the opposite of what is stored in `m_vos` when the set is cofinite. You might want to implement a private member function that searches `m_vos` without regard to `m_cofinite`. Either that or make very sure when you want `isMember()` and when you want `!isMember()`.

- On some compilers, to use the `string` class you must have:

```
#include <string>
```

- You can compare C++ strings with `==`. You don't need to use `strncmp()` because the `string` class has overloaded comparison operators.

- Empty strings are strings.

- To use the `vector` class, you must

```
#include <vector>
```

- Very quick tutorial on `vector`. (You should read Section 7.3 of the textbook!) The `vector` class lets you create automatically expanding arrays of any type. The type is enclosed in < and >.

```
vector<int> A ;       // "array" of ints
vector<float> B ;     // "array" of floats
vector<string> C ;    // "array" of strings
```

  After that you can use `A[i]`, `B[i]` and `C[i]` to index an item in the vector. Indexing starts at zero, just like arrays. You can use `C.size()` to retrieve the number of items in the vector. Note that the last item in the vector `C` has index `C.size() - 1` .

  Initially, a vector is empty. Indexing into a non-existent position in a vector can cause a segmentation fault.

  To add an item to a `vector`, use the `push_back()` method. Don't forget the underscore character in `push_back`. (This is analogous to `append()` for lists in Python.) For example,

```
A.push_back(5) ;
B.push_back(3.14) ;
C.push_back("Hello World") ;
```

- The header file `sos.h` along with some test main programs are available on GL here:

```
/afs/umbc.edu/users/c/h/chang/pub/cs202stuff/proj2
```

  You should put your implementation of the `SoS` member functions in a file called `sos.cpp`. You should not have a `main()` function in `sos.cpp`. All `main()` functions should be in a separate file. The test programs are not exhaustive, you should make your own tests.

- On GL, you can compile the test program with your implementation with one command, like this:

```
linux3% g++ -Wall -g test1.cpp sos.cpp
```

  or separately, like this:

```
linux3% g++ -Wall -g -c sos.cpp
linux3% g++ -Wall -g -c test1.cpp
linux3% g++ -g sos.o test1.o
```

  You only need to recompile `sos.cpp` and `test1.cpp` if you change it.

Submitting your program

You should submit these files to the `proj2` subdirectory:

- `sos.h` (The only change should be the addition of private member function prototypes, if any.)
- `sos.cpp` This should contain your implementations of all of the `SoS` member functions. This file **should not** have a `main()` function.
- `mytest.cpp` A main program that exercises the *working* parts of your submission. This is where you tell us what works. If you could not get some member function to work, don't test it here. We will test all functions with other programs.

If you followed the directions in setting up shared directories, then you can copy your code to the submission directory with:

```
cp sos.h sos.cpp mytest.cpp ~/cs202proj/proj2/
```

You can check that your program compiles and runs in the `proj2` directory, but please clean up any `.o` and executable files. Again, do not develop your code in this directory and you should not have the only copy of your program here.