CS253 HW6: Smaph

Description

For this assignment, you will implement a templated STL-like container. It's kind of like a **s**et, kind of like a **m**ap, and kind of like a gr**aph**, so it's a **Smaph**.

You will turn in a single file, `Smaph.h`. You may use C++11 features.

Template Parameters

A `Smaph` takes three template parameters:

- The first of the two types to be stored.
- The second of the two types to be stored.
- Comparison functor (defaults to `std::less<value_type>`). It compares two instances of `std::pair`, and returns true if the first should be before the second in the collection. `std::less` with a `std::pair` is a two-tier comparison: it compares the first values, then break ties by comparing the second values. However, a user-supplied functor may compare in any way it wants to.

Examples

Here are several example programs:

```
#include "Smaph.h"
#include <iostream>

using namespace std;

int main() {
    Smaph<double, int> s(6.6, 3);
    const pair<double,int> data[] = {{4.4, 9}, {2.2, 1}};
    s.insert(data, data+2);
    s.insert(2.2,1);
    s.insert(make_pair(4.4,3));
    cout << "There are " << s.size() << " elements.\n";
    for (Smaph<double,int>::iterator it=s.begin(); it!=s.end();
++it)
        cout << (*it).first << ',' << it->second << '\n';
}

There are 4 elements.
2.2,1
4.4,3
4.4,9
```

6.6,3

```
#include "Smaph.h"
#include <iostream>
#include <string>

using namespace std;

template<typename T>
ostream &operator<=(ostream &os, const T &rhs) {
    os << rhs.size() << " items:";
    for (typename T::iterator it=rhs.begin(); it!=rhs.end(); ++it)
        os << ' ' << (*it).first << ',' << it->second;
    return os << '\n';
}

int main() {
    Smaph<string, unsigned short> cat, tab;
    cat.insert("c",3);
    cat.insert("a",1);
    cat.insert("t",20);
    tab.insert("t",20);
    tab.insert("a",1);
    tab.insert("b",2);
    cout <= cat <= tab <= (cat|tab) <= (cat&tab);
}

3 items: a,1 c,3 t,20
3 items: a,1 b,2 t,20
4 items: a,1 b,2 c,3 t,20
2 items: a,1 t,20
```

```
// A C++11 test program that uses a comparison functor

#include "Smaph.h"
#include <iostream>
#include <utility>          // for pair
#include <cstdlib>          // for abs

using namespace std;

// Order two pairs by “width” (the difference between the elements
// of the pair).  A pair with small width (elements close
together)
```

```
    // should come before a pair with large width (elements far
apart).
    //
    // If the widths of the pairs are identical, sort the pairs
themselves.

    struct Width {
        bool operator()(pair<int,int> a, pair<int,int> b) const {
            int width_a = abs(a.first-a.second); // width of first
pair
            int width_b = abs(b.first-b.second); // width of second
pair
            if (width_a < width_b) return true;
            if (width_a > width_b) return false;
            return a < b;                  // tie, resort to pair
compare
        }
    };

    int main() {
        pair<int,int> data[] = {{100, 104}, {1,2}, {9,6}, {9,5},
{10,19}};
        Smaph<int, int> s(data, data+5);

        cout << "With default comparison:\n";
        for (auto p : s)
            cout << p.first << ',' << p.second << '\n';

        Smaph<int, int, Width> t(data, data+5);

        cout << "\nWith Width comparison functor:\n";
        for (auto p : t)
            cout << p.first << ',' << p.second << '\n';

    }

    With default comparison:
    1,2
    9,5
    9,6
    10,19
    100,104

    With Width comparison functor:
    1,2
```

```
    9,6
    9,5
    100,104
    10,19
```

```
    // A test program with incomparable types.

    #include "Smaph.h"
    #include <iostream>
    #include <utility>
    #include <string>

    using namespace std;

    // A wrapper class around a string.  Names can't be compared with
< or ==.
    class Name {
      public:
        Name() : name("Unknown") { }
        Name(const string &n) : name(n) { }
        string name;                    // oddly public
    };

    // Order pairs of <Name,int> by the id number
    struct Order {
        bool operator()(const pair<Name,int> &a, const pair<Name,int>
&b) const {
            return a.second < b.second;
        }
    };

    int main() {
        Smaph<Name, int, Order> s;

        s.insert(Name("Dora"), 800000002);
        s.insert(Name("Jack"), 800000001);
        s.insert(Name("Fred"), 800000009);
        s.insert(Name("Xena"), 800000001);          // oops--duplicate
ID!

        for (Smaph<Name, int, Order>::iterator it=s.begin();
it!=s.end(); it++)
            cout << it->first.name << ' ' << it->second << '\n';
    }
```

```
Jack 800000001
Dora 800000002
Fred 800000009
```

Requirements for stored type

The type stored by this templated class has the following requirements:

- Default constructor
- Copy constructor
- Assignment operator
- It is *not* required that ==, <, etc., be valid for the stored types. Your code must *not* use such operators—it must use the comparison functor, which should default to the `less<T>` functor.

Required public types of `Smaph`

The following types have similar meanings to those of a `map`:

- `size_type` (must be an unsigned integral type)
- `key_type` (the first template argument)
- `mapped_type` (the second template argument)
- `value_type` (`std::pair<key_type,mapped_type>`)
- `iterator`

Required public methods of `Smaph`

- default ctor
- copy ctor
- ctor that takes a half-open range of two iterators
- These do *not* have to be iterators from a `Smaph`.
- ctor that takes two arguments: `key_type`, `mapped_type`
- ctor that takes a `value_type`
- assignment operator
- destructor
- `iterator begin() const`
- `iterator end() const`
- `bool empty() const`

- `size_type size() const`
- Number of data items currently stored
- `size_type max_size() const`
- Maximum possible number of elements allowed by this design
- `iterator find(const key_type &, const mapped_type &) const`
  `iterator find(const value_type &) const`
- Look for the instance of the given value. Return end() upon failure.
- `size_type count(const key_type &, const mapped_type &) const`
  `size_type count(const value_type &) const`
- Return how many times the given value occurs in the container.
- `pair<iterator,bool> insert(const key_type &, const mapped_type &)`
  `pair<iterator,bool> insert(const value_type &)`
- Insert this value, in order
- Duplicates are not permitted
- Returns an `iterator` pointing to the value in the container. An interator pointing to the value is *always* returned, whether or not anything got inserted.
- Returns a boolean that is `true` if the value was just inserted, as opposed to being there already.
- `void insert(iterator, iterator)`
- These are not necessarily `Smaph` iterators.
- Insert the values in this half-open range into the collection.
- The iterators can be of any type, referring to any kind of collection.
- The iterators may point to duplicates; or values out of order.
- However, the resulting `Smaph` must not contain any duplicates, and must be in order.
- `bool erase(const key_type &, const mapped_type &)`
  `bool erase(const value_type &)`
- Erase the value from the collection.
- Return true if it was there.
- `void erase(iterator)`
- This is an `iterator` from the `Smaph`.
- Erase the value at that location.
- `void erase(iterator, iterator)`
- These are `iterators` from the `Smaph`.
- Erase the values in that half-open range.
- `void clear()`
- Make it have no values.

Required operators

- `Smaph | Smaph`
- `Smaph | value_type`
- `value_type | Smaph`
- Returns the union of the two arguments.
- `Smaph & Smaph`
- `Smaph & value_type`
- `value_type & Smaph`
- Returns the intersection of the two arguments.
- `Smaph |= Smaph`
- `Smaph |= value_type`
- Same as `Smaph = Smaph |` *right-hand-side*
- `Smaph &= Smaph`
- `Smaph &= value_type`

- Same as `Smaph = Smaph &` *right-hand-side*

Required operations on `Smaph::iterator`:

- default constructor
- copy constructor
- assignment operator
- destructor
- `==`
- `!=`
- indirection
- `->`
- pre-/post-increment/decrement

The value exposed by the iterator must be read-only:

- `*iter=...`; must not compile
- `iter->first=...`; must not compile
- `iter->second=...`; must not compile

Requirements

- We will compile your program with: `g++ -std=c++11`
- Your header file must have `#include` guards.
- Your header file must not have any `using` declarations, not even selective ones like `using std::pair`.
- You may not use any C++ containers in your implementation. No `list`, no `vector`, no `set`, no `string`, etc.
- You may use C++ algorithms, functors, `pair`, etc.
- You may not use any external programs, e.g., via `system`, `popen`, `fork`, `exec`, etc.
- No memory leaks.
- It must be possible for a program have several instances of `Smaph` active at the same time, and they must not interfere with each other.
- It is acceptable for `insert`, `erase`, and `clear` to invalidate all existing iterators. It is also acceptable for them to not invalidate any iterators. It's up to you.
- Pay attention to `const`-correctness. We may try to assign a `const Smaph` to a non-`const Smaph`, take the `size()` of a `const Smaph`, etc.
- Brush your teeth after every meal.
- You may add other methods and operators, as needed.
- You may implement the methods inside or outside of the class declaration.