Section Handout: Templates

Multiset by Jerry, partition text by Nihkil Sarin, code and practice by Jerry and Hiroshi.

Templates and Operator Overloading

A `MultiSet` is a collection class which simply keeps track of the elements that have been placed in it. The semantics of all supported operations are identical to those of a traditional `Set` in that the `MultiSet` supports union (via `operator+` and its variants) and intersection (via `operator&`, even though I don't include this one in the interface). The only difference between a `Set` and a `Multiset` is that a `Multiset` allows (and even keeps track of) the number of times a particular element has been inserted.

Your job is to implement a template `MultiSet` class to provide a polymorphic, type-safe container class where the amount of storage devoted to any particular instance is proportional to the number of unique keys it stores. The `.h` file is presented below, and you must provide a complete `.cc` file with all of the implementation code needed to support it. The interface file would normally be much larger, but I've only included six `MultiSet` methods (in `bold`).

```
#ifndef ___multiset___
#define ___multiset___

template <class Key>
class MultiSet {

   public:
      vector<Key> getKeys() const;
      int getMultiplicity(const Key& key) const;
      const MultiSet operator+(const MultiSet<Key>& mset) const;
      const MultiSet& operator+=(const Key& key);
      const MultiSet& operator+=(const MultiSet<Key>& mset);

   private:
      vector<pair<Key, int> > items;
      void operator+=(const pair<Key, int>& key);
};

#include "multiset.cc"
#endif
```

Assume that the `vector` and the templetized `Key` type support all necessery deep-copying semantics, including a default constructor, a copy constructor, `operator=`, and `operator==`. While implementing the `MultiSet`, you must respect the above interface—that is, you may not define any more methods, nor may you add any new data members. Provide a full implementation of the templatized `MultiSet` class. Be careful to properly templatize any class and arguments that need to be tempatized.

Solution

```
template <class Key>
vector<Key> MultiSet<Key>::getKeys() const
{
   vector<Key> keys;
   for (int i = 0; i < items.size(); i++) {
      keys.push_back(items[i].first);
   }

   return keys;
}

template <class Key>
int MultiSet<Key>::getMultiplicity(const Key& key) const
{
   int count = 0;
   for (int i = 0; i < items.size(); i++) {
      if (key == items[i].first)
         return items[i].count;
   }

   return 0;
}

template <class Key>
const MultiSet<Key>& MultiSet<Key>::operator+=(const Key& key)
{
   for (int i = 0; i < items.size(); i++) {
      if (key == items[i].first) {
         items[i].second++;
         return *this;
      }
   }

   items.push_back(make_pair(key, 1));
   return *this;
}

template <class Key>
void MultiSet<Key>::operator+=(const pair<Key, int>& pair)
{
   for (int i = 0; i < items.size(); i++) {
      if (pair.key == items[i].first) {
         items[i].second += pair.second;
         return;
      }
   }

   items.push_back(make_pair(key, 1));
}
```

```
template <class Key>
const MultiSet<Key>&
MultiSet<Key>::operator+=(const MultiSet<Key>& mset)
{
   for (int i = 0; i < mset.items.length(); i++) {
      this->operator+=(mset.items[i]);
   }

   return *this;
}

template <class Key>
const MultiSet<Key>
MultiSet<Key>::operator+(const MultiSet<Key>& mset) const
{
   MultiSet sum;
   sum += *this;
   sum += mset;
   return sum;
}
```

You'll note that I provide absolutely nothing in the interface regarding constructors, destructors, or assignment. Why, then, does the following test progam compile and run as expected?

```
int main()
{
   MultiSet<int> ints;           // calls the default constructor
   for (int i = 0; i < 300; i++)
       for (int j = 0; j < i; j++)
         ints += i;

   MultiSet<int> copy(ints);     // calls the copy constructor
   MultiSet<int> sum;            // calls the default constructor
   sum = ints + copy;            // calls the assignment operator

   vector<int> myIntKeys(sum.getKeys());
   for (int k = 0; k < myIntKeys.length(); k++)
      cout << "Key: " << myIntKeys[k] << " (occurs "
           << sum.getMultiplicity(myIntKeys[k]) << " times). " << endl;
}
```

Solution

Recall that C++ generates a default constructor, a default copy-constructor, a default destructor, and a default assignment (operator=) method. Compiler genarated memory-management methods will recursively call the same methods on all of the direct embedded objects within a class instance. Since the only data member (I was careful to set it up that way and to advertise the interface as something which was non-negotiable) is a direct vector object with a full suite of it's own memory management methods.

One caveat: I needed to create a default constructor for the inner Pair class; otherwise the vector<Pair> would have a hard time allocating space to store the Pairs.

The following program does not compile, because of **one** problem with the MultiSet. What isn't the MultiSet doing?

```
int main()
{
    MultiSet<int> ints;
    MultiSet<int> copy(ints);
    MultiSet<int> sum = ints + ints;

    MultiSet<MultiSet<int> > mySets;
    mySets += sum;
    mySets += copy;
    mySets += ints;
}
```

Solution

The MultiSet isn't exporting an operator== method. As a result, code for the MultiSet<MultiSet<int>>::operator+= can't be compiled, because it is expanded to something like the following:

```
const MultiSet<MultiSet<int>>&
MultiSet<MultiSet<int>>::operator+=(const MultiSet<int>& key)
{
    for (int i = 0; i < items.length(); i++) {
        if (key == items[i].key) {
            items[i].count++;
            return *this;
        }
    }

    items.append(Pair(key));
    return *this;
}
```

I've circled the operator==-based comparison of one MultiSet<int > with another. Since operator== isn't part of the MultiSet abstraction (and you can't compare structs or classes using == unless you provide one, i.e. there is no default), you run into a problem.

Note that the declaration of the mySets variable isn't necessarily a problem, since most compilers will only generate individual methods of a template on an as-needed

basis. Only the constructor is needed, and the constructor doesn't depend on any operator== functionality of the type in any way.

Templates and the STL

Shown on the following page is a function template `partition`. `partition` takes a range of elements (with beginning `start` and end `end`), and reorders the range such that all elements that satisfy the given `predicate` (the 'predicate' is a function that when passed an element, returns either `true` or `false`. Here, a pointer to this function is passed to the template `partition`) are collated at the front of range, and the elements that don't satisfy it are collected at the end of it. `partition` returns a pointer to that element in the range that is the first element (counting from the beginning of the range) of those that failed the `predicate`.

```
template <class ForwardIterator, class Predicate>
ForwardIterator partition(ForwardIterator start,
                          ForwardIterator end,
                          Predicate pred)
{
   ForwardIterator next = start;
   while (start != end)
   {
      if (pred(*start))
      { // start belongs in front half
         swap(*next, *start);
         ++next;
      }
      ++start;
   }

   return next;
}
```

Note that we use the ubiquitous `swap` template as well.

Here's how we would use our `partition` template:

```
inline bool isEven(int a)
{
   return (a%2 == 0);
}

int range[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int *mid = partition(range, range + 10, isEven);
```

The result after `partition` is:

```
{ 2, 4, 6, 8, 10, 3, 7, 1, 9, 5 };
```

and `mid` points to the entry in the `range` array that contains 3.

What constraints does `partition` impose on its parameter types? Not too many; in fact the only constraints on `ForwardIterator` are imposed by `swap`, namely that whatever we get by dereferencing a `ForwardIterator` have a copy constructor and an assignment operator defined. What is more interesting is the need for the increment operator to work correctly on `ForwardIterators`, i.e. incrementing should take us to the next item in the range. Since we have used an array in our example, this works just fine.
However, if our iterator type had been set to address an element of a linked list, such as

```
struct cell{
   int value;
   cell *next;
}
```

and the range was specified by two pointers into a linked list, then `partition` would not do such a good job. In this case, we would need to overload the increment operator for pointers to `cell` structs. The increment operator would cause a pointer to take on the value of the `next` field of the structure it points to.

It turns out that `partition` is already a part of the C++ Standard Library. In addition, the SGI version of the STL provides a singly-linked list container class called `slist`. Associated with `slist` is an iterator type which meets the requirements of a `ForwardIterator`. So you could do something like:

```
slist<int> sl;
for(int i=0; i<10; ++i)
  sl.push_front(i);

slist<int>::iterator ip =
   partition(sl.begin(), sl.end(), isEven);
```

The STL groups sets of requirements on iterators into a hierarchy of iterator concepts. `ForwardIterator` is one such concept. `InputIterator` is a set of requirements which cover iterators on read-only containers (e.g. and incoming network stream). There is a similar concept called `OutputIterator`. The requirements which form `Forward Iterator` contains all the requirements in `InputIterator` and `OutputIterator`, and also adds a few requirements of its own. It is said that `ForwardIterator` is a *refinement* of `InputIterator` and `Output Iterator`. The refinement relationship between concepts is similar to an inheritance relationship between classes. Therefore, just as we can have a hierarchy of classes, we can have a hierarchy of concepts.
When a class satisfies the requirements of an iterator concept, then that class is said to be a *model* of that concept. For example, `slist<int>::iterator` is a model of `ForwardIterator`. If a class is a model of a concept which is a refinement of another concept, then that class is also a model of that other concept. For example, `slist<int>::iterator` is also a model of both `InputIterator` and `OutputIterator`.

The other two important iterator concepts are `BidirectionalIterator`, which is a refinement of `ForwardIterator`, and `RandomAccessIterator`, which is a refinement of `BidirectionalIterator`. Every pointer is a model of `RandomAccessIterator`. An iterator associated with a doubly-linked list would be a model of `BidirectionalIterator`.

Templatized Algorithms Practice

a.) The templatized `remove_copy_if` algorithm copies elements from the range `[first, last)` to a range beginning at `result`, except that elements for which `pred` is `true` are not copied. The return value is the end of the resulting range. This operation is stable, meaning that the relative order of the copied elements is the same as in the range `[first, last)`.

```
/*
 * Templatized Function: remove_copy_if
 * ------------------------------------
 * Copies elements from the range [first, last) to the range beginning
 * at result, except for those elements for which the specified
 * predicate function is true.  The return value is the end of
 * the resulting range.  Note that remove_copy_if is already a part of
 * the STL.
 */

inline template <class InputIterator, class OutputIterator,
                 class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result,
                              Predicate pred)
{
```

b.) List the assumptions about the `InputIterator`, `OutputIterator`, and the base class referenced by the `Input/OutputIterator` that must be made in order for this algorithm to compile when expanded.

c.) Write a function `removeNegativeFractions`, which takes an array of `Fractions` and filters out those that are negative. The original array should be used, and the effective size of the array after filtering should be returned. You should iterate over the array only once, using the `remove_copy_if` algorithm to do so. You will need to write a helper predicate function.

```
/*
 * Function: removeNegativeFractions
 * ---------------------------------
 * Examines the specified array of Fraction objects and removes
 * those that are negative while preserving the order of those
 * that remain.  The filtering is performed in place, and the
 * effective size of the filtered array is returned.
 */

int removeNegativeFractions(Fraction array[], int n)
{
```

Solution

a.)

```
/*
 * Templatized Function: remove_copy_if
 * ------------------------------------
 * Copies elements from the range [first, last) to the range beginning
 * at result, except for those elements for which the specified
 * predicate function is true.  The return value is the end of
 * the resulting range.
 */

inline template <class InputIterator, class OutputIterator,
                 class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result,
                              Predicate pred)
{
   while (first != last) {
      if (!pred(*first)) {
         *result = *first;
         ++result;
      }
      ++first;
   }

   return result;
}
```

b.)

Whatever true-type gets bound to the templatized type `InputIterator` must respond to `operator!=()`, `operator*()`, and `operator++()`. `OutputIterator` must respond to `operator*()` and `operator++()`. The type returned by `OutputIterator`'s

`operator*()` must be assignable to whatever type is returned by `InputIterator`'s `operator*()`, and finally, `pred` may either be a function pointer or a function object that can be called to return something convertible to a `bool`. The type returned by `InputIterator`'s `operator*()` must be convertible to the argument of the predicate.

c.)

```
static inline bool belowZero(const Fraction& f)
{
   return f < 0;
}

int removeNegativeFractions(Fraction array[], int n)
{
   Fraction* end = remove_copy_if(array, array + n, array, belowZero);
   return end - array;
}
```