CS107L Handout 04
Autumn 2007 October 19, 2007

Custom STL-Like Containers and Iterators

This handout is designed to provide a better understanding of how one should write template code and architect iterators to traverse containers. The entire discussion centers on the design and implementation of a singly linked list—something that is officially included in the STL. However, we rewrite a subset of that functionality here, because the linked list is the least complicated class of all classes requiring custom iterator implementations. Let's get to it.

Design of **`mylist`** to look and feel like the STL **`list`** class

If we're going to have a linked list, then there's no denying the need for a link list node.

```
template <typename T> class mylist;                 // forward declare
template <typename T> class mylist_iterator;        // forward declare

template <typename T>
class mylist_node {

   friend class mylist<T>;
   friend class mylist_iterator<T>;

 private:
   mylist_node(const T& t, mylist_node<T> *next) : elem(t), next(next) {}
   ~mylist_node() { delete next; }
   T elem;
   mylist_node<T> *next;
};
```

If, for just a moment, you rid of the constructor, you reduce the **`mylist_node`** to nothing more than a normal **`struct`** that just happens to mark everything as **`private`**. Naturally, to mark everything as **`private`** is to block everyone out, and for the most part, that's what we want here. However, **`mylist`**, **`mylist_node`**, and **`mylist_iterator`** are clearly intertwined, so the **`mylist_node<T>`** class allows any instance of **`mylist<T>`** and **`mylist_iterator<T>`** to examine the **`private`** data of a **`mylist_node<T>`** instance. This permission comes in the form of an explicit **`friend`**ship statement placed at the front of the **`mylist_node`** definition.

Some key (or at least interesting) observations:

1. The **public** and **private** keywords have absolutely no influence over friendship. The very fact that the **mylist_node** granted **friend**ship is enough to give all **mylist** and **mylist_iterator** full access to everything **mylist_node**-related. Note that the friendship is being offered to the classes themselves. That means that all instance and class methods have full access to any instance field of the **mylist_node**—specifically, **mylist** and **mylist_iterator**s can directly access any and all **private** data members and, had there been any, call any **private** methods as well.

2. Understand that the friendship is being granted to classes embracing the same exact type. **mylist_node<string>** grants friendship to **mylist<string>**, **mylist_node<film *>** grants friendship to **mylist<film *>**, and so forth. Technically, **mylist_node<string>** couldn't give a flying feather about **mylist<film *>**, so it sees no reason to grant friendship to all things **mylist**—just **mylist<string>**. This should be clear, because the lines granting **friend**ship include the **<T>**.

```
friend class mylist<T>;
friend class mylist_iterator<T>;
```

3. I chose to inline the implementation of the constructor and destructor, but I'm only getting away with it here because there are so short and so relatively obvious. Had I preferred to place the implementations in the corresponding .cc file instead, I'd have to have written them as follows:

```
template <typename T>
mylist_node<T>::mylist_node(const T& t, mylist_node<T> *next) : elem(t), next(next)
{
   // nothing needed, as everything is taken care of by the initialization list
}

template <typename T>
mylist_node<T>::~mylist_node() { delete next; }
}
```

4. Remember that the compiler automatically generates a copy constructor and an **operator=** method for us if we don't explicitly mention it in the class definition. Since we **don't** mention either here, we get those compiler-synthesized versions (and they're automatically **public**).

Who would have thought that the **mylist_node** template class could be so very interesting? Let's start small and pretend that anything and everything relying on the existence of a true iterator type isn't included.

```
template <typename T>
class mylist {

 public:
  mylist() : head(NULL), tail(NULL) {}
  ~mylist() { delete head; }

  bool empty() const { return head == NULL; }
  void push_back(const T& elem);

 private:
  mylist_node<T> *head, *tail;
};
```

This doesn't even come close to the real linked list, but my ultimate goal here isn't to remind you what a linked list should do and how it works, but rather to motivate and implement an **iterator**. The mechanics of threading together a linked list is either old hat by now, or if not, can be reviewed in a matter of 15 minutes. The only difference between our linked list and the ones you've dealt with in previous courses is the language we're building them in. The logistics of **next** pointers and **NULL** checks and whatnot are exactly the same. Notice I've inlined the implementation of the constructor, the destructor, and the **empty** method. Here's what the **mylist.cc** file would look like if this were all **mylist** was defined to be.

```
template <typename T>
void mylist<T>::push_back(const T& elem)
{
   mylist_node<T> *newnode = new mylist_node<T>(elem, NULL);
   if (head == NULL) {
      head = newNode;
   } else {
      tail->next = newNode;
   }
   tail = newNode;
}
```

The details of linked list insertion and deletion are always tricky, no matter what language they're in. If you doubt the implementation, then you should trace each method to ensure that all cases (empty list, list of length one, and all other lists) are properly handled. The most interesting element of the implementation—at least from a C++ standpoint—is the call to the **mylist_node<T>** constructor—a call that's permitted only because the **mylist** class was granted that **friend**ship mentioned earlier.

Now that we have the basics of the linked list in place, it's high time we introduce the iterator so that built-in STL algorithms like **for_each** and **find** can traverse the elements of our **mylist** from front to back and making it appear as if the elements inside the **mylist** are all laid out sequentially in memory. A terribly bad design would extend our current definition of the **mylist** class to include very buggy implementations of **begin** and **end** methods. Buggy, buggy, buggy, buggy!

```
template <typename T>
class mylist {

   public:
      typedef T *iterator;

   public:
      mylist() : head(NULL), tail(NULL) {}
      ~mylist();

      bool empty() const { return (head == NULL); }
      void push_back(const T& elem);

      iterator begin() { return (head == NULL ? NULL : &head->elem); }
      iterator end() { return NULL; }

   private:
      mylist_node<T> *head, *tail;
};
```

Such an implementation would assume that we've absolutely no other choice for the **typedef**. If **begin** needs to return the 'address' of the first element **and** this 'address' should respond to * and **++** as a true pointer would, one might think there's really nothing else we can do.

Perhaps it's the best that can be done; but if so, then this whole iterator thing would be pretty lame—particularly lame here, since there's no way the individual elements of a list could possibly be accessed by an iterator that assumes all elements are organized side by side in memory. To drive this point home, consider the following:

```
mylist<string> ivies;
ivies.push_back("Harvard");
ivies.push_back("Yale");
ivies.push_back("Princeton");
ivies.push_back("Penn");

mylist<string>::iterator begin = ivies.begin();
mylist<string>::iterator end = ivies.end();

while (begin != end) {
   if (*begin == "Stanford") {
      cout << "Stanford's an Ivy." << endl;
      return;
   }
   ++begin;
}

cout << "They're all snobs anyway." << endl;
```

Algorithmically, the code snippet is sound, and yet it doesn't work—in fact, it seg faults. The blame can't be pinned on the code snippet itself, but rather the current definition of the **`mylist<string>::iterator`**. How can a local **`string *`**—that's all the **`iterator`** is in this example—behave any differently than a regular pointer? All those **`++begin`** instructions are going to advance the pointer through memory as if there's an array of **`string`**s there, and that's just not the case. The **`iterator`** doesn't know how to advance to the next element in the sequence, because the instructions it follows to advance—**`operator++`** being that instruction—just tell it to march **`sizeof(string)`** bytes ahead. The **`iterator`** we want here is something that knows how to update itself to point to the next element in the list. A normal pointer can't do that, but a class that responds to clever implementations of **`operator*`** and **`operator++`** can.

The idea of returning a class as an iterator seems a little off. However, as long as the type being returned responds to pointer syntax and exhibits normal pointer semantics, we shouldn't really care whether the iterator is a true pointer or something that just pretends to be. We're interested in appearances—once again, it all comes down to looks. Shallow!

At this point, we can update the **`mylist`** template to export a new iterator type—one that has a better chance of visiting all of the nodes:

```
template <typename T>
class mylist {

   public:
      typedef mylist_iterator<T> iterator;

   public:
      mylist() : head(NULL), tail(NULL) {}
      ~mylist() { delete head; }

      bool empty() const { return head == NULL; }
      void push_back(const T& elem);

      iterator begin();
      iterator end();

   private:
      mylist_node<T> *head, *tail;
};
```

Notice the new **`typedef`** for iterator—there's a big difference. **`begin`** and **`end`** now return an instance of this **`mylist_iterator<T>`** thing—a type we'll more fully flesh out in a page or two.

While the details of how the **`mylist_iterator`** aren't clear yet, we do know that the code snippet we wrote earlier will now be translated/interpreted as follows:

```
mylist<string> ivies;
ivies.push_back("Harvard");
ivies.push_back("Yale");
ivies.push_back("Princeton");
ivies.push_back("Penn");

mylist<string>::iterator begin = ivies.begin();
mylist<string>::iterator end = ivies.end();

while (begin != end) {
   if (*begin == "Stanford") {
      cout << "Stanford's an Ivy."
           << endl;
      return;
   }
   ++begin;
}

cout << "They're all snobs anyway."
     << endl;
```

what we write

```
mylist<string> ivies;
ivies.push_back("Harvard");
ivies.push_back("Yale");
ivies.push_back("Princeton");
ivies.push_back("Penn");

mylist<string>::iterator begin = ivies.begin();
mylist<string>::iterator end = ivies.end();

while (begin.operator!=(end)) {
   if (begin.operator*() == "Stanford") {
      cout << "Stanford's an Ivy."
           << endl;
      return;
   }
   begin.operator++();
}

cout << "They're all snobs anyway."
     << endl;
```

how the compiler interprets it when `mylist<string>::iterator` is a direct class instance.

The placement of the new and improved **`mylist<string>::iterator`** in this context forces it to respond to **`operator!=`**, **`operator*`**, and **`operator++`** in order to make the compiler happy. And because we want the iterator to traverse the **`mylist<string>`** and identify references to all of the **`string`**s inside, the implementations of **`operator!=`**, **`operator*`**, and **`operator++`** need to mimic whatever functionality comes when these operators are applied to real pointers. Naturally, we also want the **`begin`** iterator to point to the first element of the list, the next iterator to somehow associate with the second element in the list, and so on.

Therefore, we need the following:

```
template <typename T>
class mylist_iterator : public iterator<forward_iterator_tag, T> {

   friend class mylist<T>;

   public:
      T& operator*();
      const mylist_iterator<T>& operator++();
      bool operator!=(const mylist_iterator<T>& other) const;

   private:
      mylist_node<T> *pointee;
      mylist_iterator(mylist_node<T> *pointee) : pointee(pointee) {}
};

template <typename T>
T& mylist_iterator<T>::operator*()
{
   return pointee->elem;
}

template <typename T>
const mylist_iterator<T>& mylist_iterator<T>::operator++()
{
   assert(pointee != NULL);
   pointee = pointee->next;
   return *this;
}

template <typename T>
bool mylist_iterator<T>:: operator!=(const mylist_iterator<T>& other) const
{
   return this->pointee != other.pointee;
}
```

Forget about **`public`** versus **`private`** for the moment; pay attention to what this **`mylist_iterator<T>`** class encapsulates and how it behaves like a pointer to an object of type **`T`**. Each instance of this **`mylist_iterator<T>`** class wraps around a pointer to a **`mylist_node<T>`**. **`operator++`** doesn't advance the **`iterator`** to point to the next node in sequential memory, but instead to point to the next node in the list (note the update setting **`pointer`** to **`pointer->next`**). **`operator!=`** mismatches **`mylist_iterators`** if and only if the **`mylist_node<T>`** *s they surround are different. **`operator*`** returns a reference to the **`T`** object embedded inside the **`node`** whose address it's storing.

You'll should make note of my decision to make all three operator methods **`public`**; marking them as **`private`** would prevent client code from handling **`iterator`**s produced by the **`begin`** and **`end`** methods. See the private constructor? That means no one except the **`mylist<T>`** class can create **`mylist_iterator<T>`**'s around a **`mylist_node<T> *`**. (The copy constructor, **`operator=`** method, and destructor are all **`public`** and compiler-synthesized, and the compiler-synthesized ones work just fine.)

The implementation of **`operator++`** is such that the following idiom would carry the iterator over all of the **`mylist_node<T> *`**s of a **`mylist<T>`**. **`begin`** would need to create an iterator storing the very first address, and then **`operator++`** would update the iterator to hold the **`next`** field, and then the next **`next`** field, and then the NEXT **`next`** field, and so on. Eventually, the **`iterator`** would be updated to store a **`NULL`** and match the iterator produced by the **`end`** method: That would be the signal that we've reached the end of the list.

Assuming this is all true, it makes the implementation of **`begin`** and **`end`** pretty obvious:

```
template <typename T>
class mylist {

   public:
      typedef mylist_iterator<T> iterator;

   public:
      mylist() : head(NULL), tail(NULL) {}
      ~mylist() { delete head; }

      bool empty() const { return head == NULL; }
      void push_back(const T& elem);

      iterator begin() { return mylist_iterator<T>(head); }
      iterator end() { return mylist_iterator<T>(NULL); }

   private:
      mylist_node<T> *head, *tail;
};
```

The Full `mylist<T>` Definition

If you inspect the **`assn-7-ss-and-btmap`** starter folder, you'll see a subfolder name **`list-iterator-code`**. Within that directory is an even more robust definition and implementation of **`mylist`**. This **`mylist`** defines two **`find`** methods (one **`const`**, one non-**`const`**) and how you can use a **`static`** template method to unify their implementations—implementations while are logically identical. It also shows you how to define the **`mylist_iterator`** template class so that both **`iterator`** and **`const_iterator`** can be supported. (Note that you are not required to implement true iterators for the **`btree_map`**. But you're certainly welcome and even encouraged to try.)