COMP6771
Advanced C++ Programming

Week 8
Part One: Custom Iterators

2016

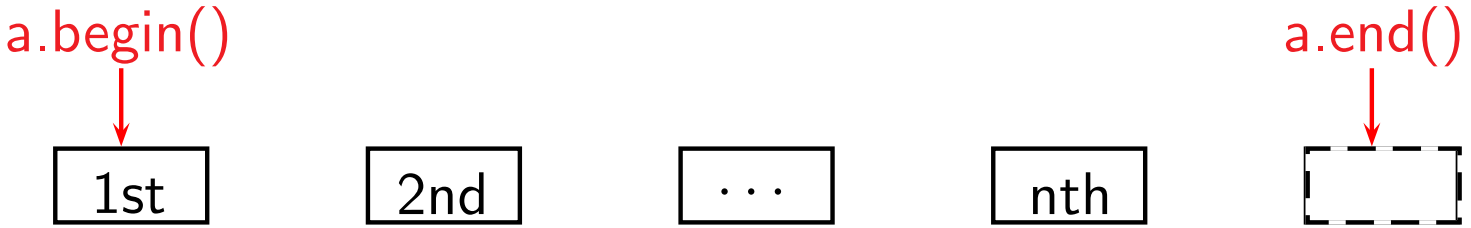
www.cse.unsw.edu.au/~cs6771

Iterators

- Iterators that are classes (or types) support an abstract model of data as a sequence of objects
- An iterator is an abstract notion of pointers
- Glue between containers and generic algorithms:
  - The designer of algorithms do not have to be concerned with details about various data structures
  - The designer of containers do not have to provide extensive access operations

An Iterator for a Container

- a is a container with all its n objects ordered



- a.begin(): a "pointer" to the first element
- a.end(): a "pointer" to one past the last element
- if p "points" to the k-th element, then
  - *p is the object pointed to
  - ++p "points" to $(k+1)$-st element
- The loop:

```
1 for (first = a.begin(); first != a.end(); ++first)
2    do something on *first
```

Container::iterator

- `vector<int> ivec;`

```
1 for (auto first = ivec.begin(); first != ivec.end(); ++first) {
2
3    std::cout << *first << std::endl;
4
5 }
```

- `list<int> ilist;`

```
1 for (auto first = ilist.begin(); first != ilist.end(); ++first) {
2
3    std::cout << *first << std::endl;
4
5 }
```

The for loops look "identical".

Five Categories of Iterators

Operation	Iterators				
	OUTPUT	INPUT	FORWARD	BI-DIR	RANDOM
Read		`=*p`	`=*p`	`=*p`	`=*p`
Access		`->`	`->`	`->`	`-> []`
Write	`*p=`		`*p=`	`*p=`	`*p=`
Iteration	`++`	`++`	`++`	`++ --`	`++ -- + - += -=`
Compare		`== !=`	`== !=`	`== !=`	`== != < > >= <=`

Different algorithms require different kinds of iterators for their operations:

- input: `find()`, `equal()`, ...
- output: `copy()`
- forward: `replace()`, ...
- bi-directional: `next_permutation()`, `reverse()`, ...
- random: `sort`, `binary_search()`, `nth_element()`, ...

Prefer ++first to first++

```
self& operator++() {    // prefix
   node = (link_type)((*node).next);
   return *this;
 }
 self operator++(int) { // postfix
   self tmp = *this;
   ++*this;
   return tmp;
 }
```

- ++first is more efficient than first++
- −−first is more efficient than first−−

Iterator Traits

- Traits define/describe the class properties for an iterator.
- Defined as nested typedefs in #include <iterator>:

```
template <typename Iterator>
 struct iterator_traits {
   typedef typename Iterator::iterator_category iterator_category;
   typedef typename Iterator::value_type        value_type;
   typedef typename Iterator::difference_type   difference_type;
   typedef typename Iterator::pointer           pointer;
   typedef typename Iterator::reference         reference;
 };

 template <typename T>   // specialised for the pointer iterator
 struct iterator_traits<T*> {
   typedef random_access_iterator_tag iterator_category;
   typedef T                          value_type;
   typedef ptrdiff_t                  difference_type;
   typedef T*                         pointer;
   typedef T&                         reference;
 };
```

- http://www.sgi.com/tech/stl/Iterators.html
- When we write a custom iterator we have to define these 5 type defs for our types.

Writing a Forward Iterator

```
list.hpp:                a singly linked list
list_iterator.hpp:       a non-const forward iterator
                           (as a class template)
                           (the const version is similar)
list-user.cpp:             client code
```

- List is a class template
- List_Iterator is also class template

list.hpp I

```
1  #ifndef LIST_HPP
2  #define LIST_HPP
3
4  #include "list_iterator.hpp"
5
6  template <typename T> class List {
7  public:
8    friend class List_Iterator<T>;
9    typedef List_Iterator<T> iterator;
10
11     List() : head_(nullptr), tail_(nullptr) {}
12     ~List() { delete head_; }
13
14     bool isEmpty() const { return head_ == nullptr;  }
15     void push_back(const T&);
16     iterator begin() { return iterator(head_); }
17     iterator end() { return iterator(nullptr); }
18
19 private:
20     struct Node {
21       Node(const T& t, Node *next) : elem_(t), next_(next) {}
```

list.hpp II

```
22      ~Node() {
23         delete next_;
24      };
25      T elem_;
26      Node *next_;
27    };
28    Node *head_, *tail_;
29  };
30
31  template <typename T>
32  void List<T>::push_back(const T& elem) {
33    Node *newNode = new Node(elem, nullptr);
34    if (!head_)
35      head_ = newNode;
36    else
37      tail_->next_ = newNode;
38    tail_ = newNode;
39  }
40  // add more member functions here
41
42  #endif
```

list_iterator.hpp I

```
1  #ifndef LIST_ITERATOR_HPP
2  #define LIST_ITERATOR_HPP
3
4  #include <iterator>
5  #include <cassert>
6
7  template <typename T> class List;
8
9  template <typename T> class List_Iterator {
10 public:
11   typedef std::ptrdiff_t                    difference_type;
12   typedef std::forward_iterator_tag         iterator_category;
13   typedef T                                 value_type;
14   typedef T*                                pointer;
15   typedef T&                                reference;
16
17   reference operator*() const;
18   pointer operator->() const { return &(operator*()); }
19   List_Iterator& operator++();
20   bool operator==(const List_Iterator& other) const;
21   bool operator!=(const List_Iterator& other) const { return !operator==(other); }
22
23   List_Iterator(typename List<T>::Node *pointee = nullptr) : pointee_(pointee) {}
24 private:
25   typename List<T>::Node *pointee_;
26 };
27
28
29
```

list_iterator.hpp II

```
30 template <typename T> typename List_Iterator<T>::reference
31 List_Iterator<T>::operator*() const {
32   return pointee_->elem_;
33 }
34
35 template <typename T> List_Iterator<T>&
36 List_Iterator<T>::operator++() {
37   assert(pointee_ != nullptr);
38   pointee_ = pointee_->next_;
39   return *this;
40 }
41
42 template <typename T>
43 bool List_Iterator<T>::operator==(const List_Iterator<T>& other) const {
44   return this->pointee_ == other.pointee_;
45 }
46
47 #endif
```

Dissecting the Custom Iterator

Key points in the Iterator Class:

- The typedefs
- The overloaded operators (*, ->)
- The equality operators
- The constructor (default to nullptr)
- The private data - a pointer to a private inner class (friend)
- The ++ operator knows how the inner class works to move onto the next item in the sequence.

Key points in the List Class:

- begin() – returns an Iterator object
- end() – returns an Iterator object (with nullptr as private data)

Note: The Iterator Class does not modify the List/Node data except through returning references.

A Custom InputIterator

- Custom InputIterator required for Assignment Three!
- To be valid it needs to adhere correctly to the requirements for the InputIterator defined in the links below:
- See: http://en.cppreference.com/w/cpp/concept/InputIterator
- See: http://www.sgi.com/tech/stl/InputIterator.html
- Will be checked/tested with test case(s) that use type traits.
- Note: an InputIterator does not allow the data to be modified (hint: use const!).
- Tip: Use smart pointers! Maybe store a collection of pointers in your Iterator to get the sort order right?
- Tip: Define ++ in terms of remove the head of the collection? Define * as return the const head?

list-user.cpp

```
1  #include<iostream>
2  #include<algorithm>
3
4  #include "list.hpp"
5
6  template <typename T> void display(T &c) {
7    std::cout << "Iterating over List: ";
8    for (typename T::iterator i = c.begin(); i != c.end(); ++i)
9    // or for (auto i = c.begin(); i != c.end(); ++i)
10     std::cout << *i << " ";
11   std::cout << std::endl;
12 }
13
14 int main() {
15   List<int> l;
16
17   l.push_back(4);
18   l.push_back(3);
19   l.push_back(2);
20   l.push_back(1);
21
22   display(l);
23
24   // use our iterator with a stl algorithm.
25   List<int>::iterator i = std::find(l.begin(), l.end(), 3);
26   if (i != l.end())
27      std::cout << "3 found" << std::endl;
28   else
29      std::cout << "3 not found" << std::endl;
30 }
```

Combining Traits and Custom Iterators

- The input:

```
 1 #include <iostream>
 2 #include <algorithm>
 3
 4 #include "list.hpp"
 5
 6 template <typename InputIterator>
 7 typename std::iterator_traits<InputIterator>::value_type
 8 last_value(InputIterator  first, InputIterator last) {
 9   typename std::iterator_traits<InputIterator>::value_type result = *first;
10   for (; first != last; ++first)
11     result = *first;
12   return result;
13 }
14
15 int main() {
16   List<int>  l;
17
18   l.push_back(4);
19   l.push_back(3);
20
21   std::cout << last_value(l.begin(), l.end()) << std::endl;
22 }
```

Combining Traits and Custom Iterators

- The instantiations from the compiler:

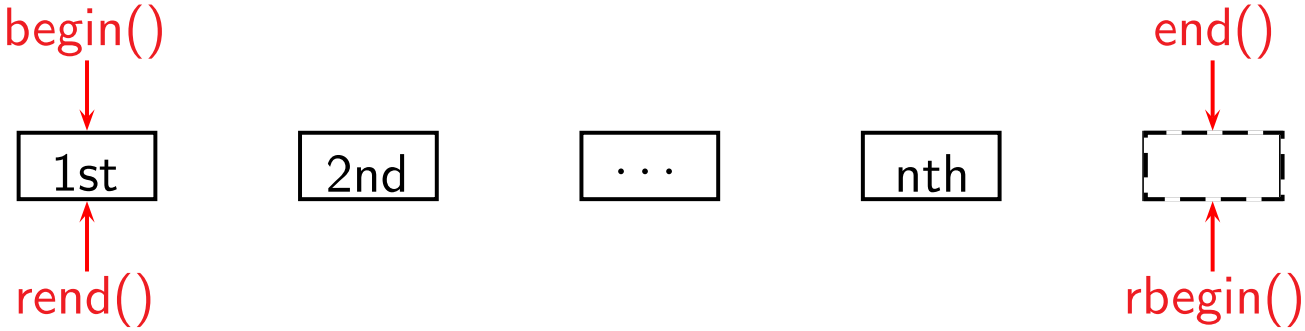
```
 1 #include <iostream>
 2 #include <algorithm>
 3
 4 #include "list.hpp"
 5
 6 int last_value(std::List_Iterator<int> first, std::List_Iterator<int> last) {
 7   int result = *first;
 8   for (; first != last; ++first)
 9     result = *first;
10   return result;
11 }
12
13 int main() {
14   List<int>  l;
15
16   l.push_back(4);
17   l.push_back(3);
18
19   std::cout << last_value(l.begin(), l.end()) << std::endl;
20 }
```

What about Generalising to a Bidirectional Iterator?

- Must define `operator--()`
- Must be able to move from `c.end()` to the last element of the list!
  $\Longrightarrow$ the list should be doubly linked

Add Reverse Iterators

- Modify List so that it is doubly linked



- Create reverse iterators by using the reverse iterator adaptor:
  - The original iterator must be bidirectional or random_access
  - BEWARE: end() must bring us to the last element in the list!
  - Add the following to the list class:

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;
typedef reverse_iterator<iterator>       reverse_iterator;
reverse_iterator rbegin()
  { return reverse_iterator(end()); }
const_reverse_iterator rbegin() const
  { return const_reverse_iterator(end()); }
reverse_iterator rend()
  { return reverse_iterator(begin()); }
const_reverse_iterator rend() const
  { return const_reverse_iterator(begin()); }
```

  - IMPORTANT: `&*reverse_iterator(i)==&*(i-1)`
  - How does the reverse adaptor work (see stl_iterator.h)?

Add C++11-Style `const` Iterators

Implement

`cbegin()`
`cend()`
`crbegin()`
`crend()`

in terms of begin(), end(), rbegin() and rend() (in a few minutes)

Summary

- Iterators are classes
- You can define your iterators as class templates (as demonstrated here)
- You can also define your iterators as template or ordinary classes nested inside other classes
- Do it in stages
  1. Develop a forward iterator first
  2. Add `operator--()` to obtain a bidirectional iterator first
  3. Use reverse_iterator to adapt your iterators to obtain reverse iterators (const and non-const)
  4. Add cbegin(), cend(), crbegin() and crend()

Reading

- Section 15.2, text
- Chapter 10, Thinking in C++, Vol. 2
- Chapter 15, Stroustrup's C++ Book, 3rd Ed.