COMP6771
Advanced C++ Programming

Week 8
Part One: Custom Iterators

2017

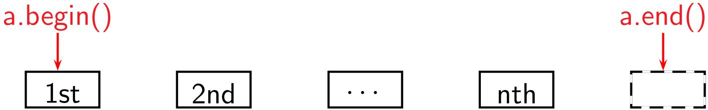
www.cse.unsw.edu.au/˜cs6771

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:

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

Container::iterator

- vector<int> ivec;

```
for (auto first = ivec.begin(); first != ivec.end(); ++first) {

   std::cout << *first << std::endl;

}
```

- list<int> ilist;

```
for (auto first = ilist.begin(); first != ilist.end(); ++first) {

   std::cout << *first << std::endl;

}
```

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

```
#ifndef LIST_HPP
#define LIST_HPP

#include "list_iterator.hpp"

template <typename T> class List {
public:
 friend class List_Iterator<T>;
 typedef List_Iterator<T> iterator;

  List() : head_(nullptr), tail_(nullptr) {}
  ~List() { delete head_; }

  bool isEmpty() const { return head_ == nullptr;  }
  void push_back(const T&);
  iterator begin() { return iterator(head_); }
  iterator end() { return iterator(nullptr); }

private:
  struct Node {
    Node(const T& t, Node *next) : elem_(t), next_(next) {}
```

list.hpp II

```
    ~Node() {
       delete next_;
    };
    T elem_;
    Node *next_;
  };
  Node *head_, *tail_;
};

template <typename T>
void List<T>::push_back(const T& elem) {
  Node *newNode = new Node(elem, nullptr);
  if (!head_)
    head_ = newNode;
  else
    tail_->next_ = newNode;
  tail_ = newNode;
}
// add more member functions here

#endif
```

list_iterator.hpp I

```
#ifndef LIST_ITERATOR_HPP
#define LIST_ITERATOR_HPP

#include <iterator>
#include <cassert>

template <typename T> class List;

template <typename T> class List_Iterator {
public:
  typedef std::ptrdiff_t                     difference_type;
  typedef std::forward_iterator_tag          iterator_category;
  typedef T                                  value_type;
  typedef T*                                 pointer;
  typedef T&                                 reference;

  reference operator*() const;
  pointer operator->() const { return &(operator*()); }
  List_Iterator& operator++();
  bool operator==(const List_Iterator& other) const;
  bool operator!=(const List_Iterator& other) const { return !operator==(other); }

  List_Iterator(typename List<T>::Node *pointee = nullptr) : pointee_(pointee) {}
private:
  typename List<T>::Node *pointee_;
};
```

list_iterator.hpp II

```
template <typename T> typename List_Iterator<T>::reference
List_Iterator<T>::operator*() const {
  return pointee_->elem_;
}

template <typename T> List_Iterator<T>&
List_Iterator<T>::operator++() {
  assert(pointee_ != nullptr);
  pointee_ = pointee_->next_;
  return *this;
}

template <typename T>
bool List_Iterator<T>::operator==(const List_Iterator<T>& other) const {
  return this->pointee_ == other.pointee_;
}

#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

```
#include<iostream>
#include<algorithm>

#include "list.hpp"

template <typename T> void display(T &c) {
  std::cout << "Iterating over List: ";
  for (typename T::iterator i = c.begin(); i != c.end(); ++i)
  // or for (auto i = c.begin(); i != c.end(); ++i)
    std::cout << *i << " ";
  std::cout << std::endl;
}

int main() {
  List<int> l;

  l.push_back(4);
  l.push_back(3);
  l.push_back(2);
  l.push_back(1);

  display(l);

  // use our iterator with a stl algorithm.
  List<int>::iterator i = std::find(l.begin(), l.end(), 3);
  if (i != l.end())
     std::cout << "3 found" << std::endl;
  else
     std::cout << "3 not found" << std::endl;
}
```

Combining Traits and Custom Iterators

- The input:

```
#include <iostream>
#include <algorithm>

#include "list.hpp"

template <typename InputIterator>
typename std::iterator_traits<InputIterator>::value_type
last_value(InputIterator  first, InputIterator last) {
  typename std::iterator_traits<InputIterator>::value_type result = *first;
  for (; first != last; ++first)
    result = *first;
  return result;
}

int main() {
  List<int>  l;

  l.push_back(4);
  l.push_back(3);

  std::cout << last_value(l.begin(), l.end()) << std::endl;
}
```

Combining Traits and Custom Iterators

- The instantiations from the compiler:

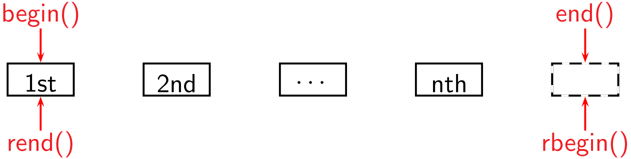
```
#include <iostream>
#include <algorithm>

#include "list.hpp"

int last_value(std::List_Iterator<int> first, std::List_Iterator<int> last) {
  int result = *first;
  for (; first != last; ++first)
    result = *first;
  return result;
}

int main() {
  List<int>  l;

  l.push_back(4);
  l.push_back(3);

  std::cout << last_value(l.begin(), l.end()) << std::endl;
}
```

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.