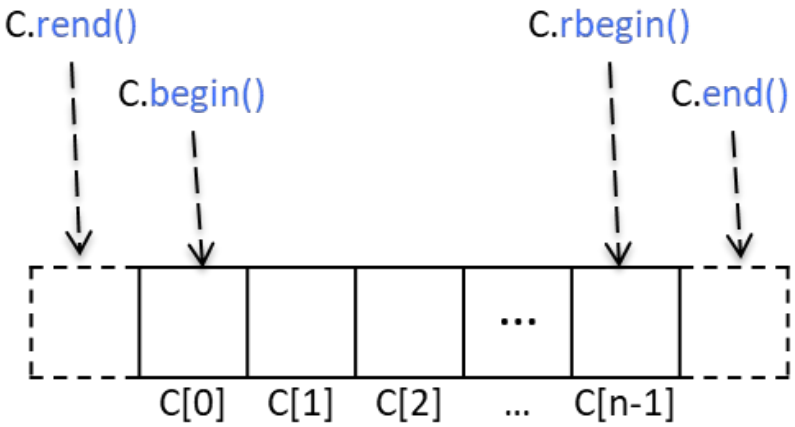
CALIFORNIA STATE UNIVERSITY FULLERTON

CPSC 131 – Data Structures

Iterators

Professor T. L. Bettens

Fall 2020



Containers and Iterators

- **Container** is an abstract data structure that stores a collection of elements
- **Iterator** abstracts the process of looping through the collection of elements
- Let C be a container and p be an iterator over C

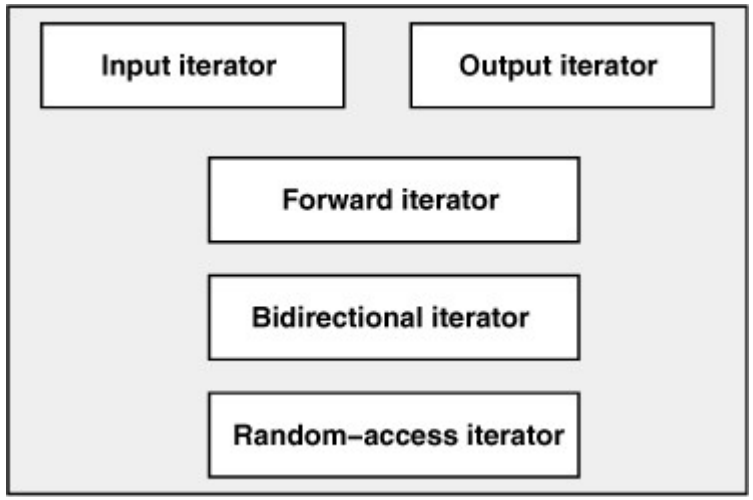
```
for (p = C.begin();  p != C.end();  ++p)
{
  p->do_something()
}
```

Iterator Categories

- Iterators are objects:
  - that can iterate over elements of a sequence via a common interface
  - adapted from ordinary pointers
- Anything that behaves like an iterator is-a iterator
  - ordinary pointer is-a iterator
- However, iterators have even more abilities!

Iterator Categories

Inheritance Hierarchy



Iterator Category	Ability	Providers
Output iterator	Writes forward	Ostream, inserter
Input iterator	Reads forward once	Istream
Forward iterator	Reads forward	Forward list, unordered containers
Bidirectional iterator	Reads forward and backward	List, set, multiset, map, multimap
Random-access iterator	Reads with random access	Array, vector, deque, string, C-style array

Iterator Capability

Forward Iterator

Expression	Effect
**iter*	Provides access to the actual element
iter->member	Provides access to a member of the actual element
++iter	Steps forward (returns new position)
iter++	Steps forward (returns old position)
iter1 == iter2	Returns whether two iterators are equal
iter1 != iter2	Returns whether two iterators are not equal
TYPE()	Creates iterator (default constructor)
TYPE(iter)	Copies iterator (copy constructor)
iter1 = iter2	Assigns an iterator

A Forward Iterator can only go forward one node at a time

Iterator Capability

Bidirectional Iterator
Everything a Forward Iterator can do, plus

Expression	Effect
`--iter`	Steps backward (returns new position)
`iter--`	Steps backward (returns old position)

A Bidirectional Iterator is-a Forward Iterator, plus more

Iterator Capability

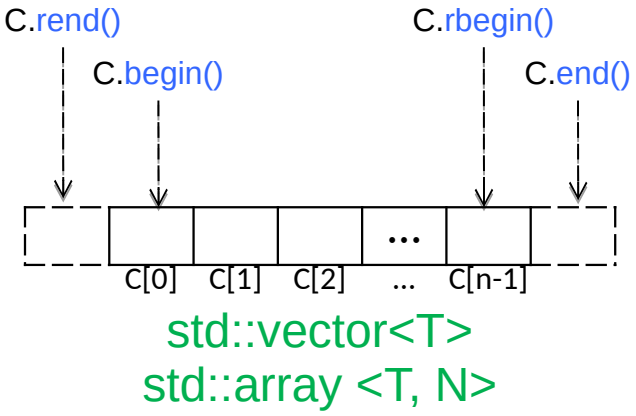
Random-Access Iterator

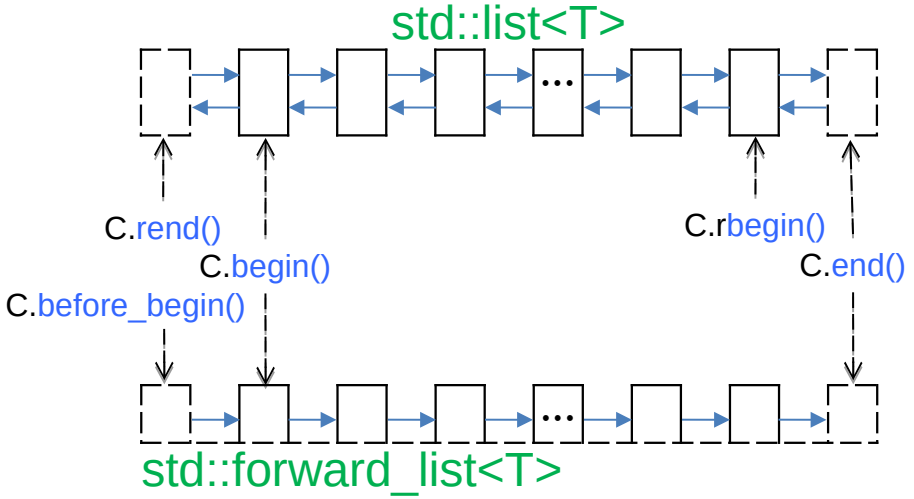
Everything a Bidirectional can do, plus

Expression	Effect
iter[*n*]	Provides access to the element that has index *n*
iter+=*n*	Steps *n* elements forward (or backward, if *n* is negative)
iter−=*n*	Steps *n* elements backward (or forward, if *n* is negative)
iter+*n*	Returns the iterator of the *n*th next element
n+*iter*	Returns the iterator of the *n*th next element
iter−*n*	Returns the iterator of the *n*th previous element
iter1−*iter2*	Returns the distance between *iter1* and *iter2*
iter1<*iter2*	Returns whether *iter1* is before *iter2*
iter1>*iter2*	Returns whether *iter1* is after *iter2*
iter1<=*iter2*	Returns whether *iter1* is not after *iter2*
iter1>=*iter2*	Returns whether *iter1* is not before *iter2*

A Random-Access Iterator is-a Bidirectional Iterator, plus more

STL Containers and Iterators





- Each STL container, call it C, has an associated class Iterator
  - begin(), rbegin(): returns an iterator to the first element
  - end(), rend(): returns an iterator to an imaginary position just after the last element
- An iterator behaves like a pointer to an element
  - *p: returns the element referenced by this iterator; access current element
  - ++p or p++: advances to the next element
- Most STL containers provide the ability to move backwards
  - --p or p--: moves to the previous element

Not a pointer-to-Node

Iterating through Containers

- Let C be a container and p be an iterator over C

```
for (p = C.begin();  p != C.end();  ++p)
{
   p->do_something()
}
```

STL Vector example

```
#include <vector>
int main()
{
  std::vector<int> C = {-2, 5, -7, 0, 10, 100};

  int sum = 0;
  for (vector<int>::iterator p=C.begin();   p != C.end();   ++p)
  {
     sum += *p;
  }
}
```

STL Single Linked List example

```
#include <forward_list>
int main()
{
  std::forward_list<int> C = {-2, 5, -7, 0, 10, 100};

  int sum = 0;
  for (forward_list<int>::iterator p=C.begin();   p !=   C.end();  ++p)
  {
     sum += *p;
  }
}
```

Very similar!

Iterating through Containers

- Let C be a container and p be an iterator over C

```
for (p = C.begin();  p != C.end();  ++p)
{
   p->do_something()
}
```

STL Vector example

```
#include <vector>
int main()
{
  std::vector<int> C = {-2, 5, -7, 0, 10, 100};

  int sum = 0;
  for (const auto & p=C.begin();   p != C.end();   ++p)
  {
     sum += *p;
  }
}
```

STL Single Linked List example

```
#include <forward_list>
int main()
{
  std::forward_list<int> C = {-2, 5, -7, 0, 10, 100};

  int sum = 0;
  for (const auto & p=C.begin();   p !=   C.end();  ++p)
  {
     sum += *p;
  }
}
```

Identical!!

Iterating through Containers

- Let C be a container and element be an item within C

```
for (const auto & element : C)
{
   element.do_something()
}
```

STL Vector example

```
#include <vector>
int main()
{
  std::vector<int> C = {-2, 5, -7, 0, 10, 100};

  int sum = 0;
  for (const auto & element : C)
  {
     sum += element;
  }
}
```

STL Single Linked List example

```
#include <forward_list>
int main()
{
  std::forward_list<int> C = {-2, 5, -7, 0, 10, 100};

  int sum = 0;
  for (const auto & element : C)
  {
     sum += element;
  }
}
```

Identical!!

STL Iterators in C++

- Each STL container type C supports iterators:
  - C::iterator – read/write iterator type
  - C::const_iterator – read-only iterator type
  - C.begin(), C.end() – return start and end iterators, respectively
  - C.rbegin(), C.rend() – return start and end reverse iterators, respectively
    - NOT FOR std::forward_list

- Various notions of iterator:
  - ***(standard) iterator***: allows read-write access to elements
  - ***const iterator***: provides read-only access to elements
  - ***forward iterator***: supports ++p
  - ***bidirectional iterator***: supports both ++p and –p
  - ***random access iterator***: supports both p+n, p-n (vectors and arrays)

Auxiliary Iterator Functions

- advance(), next(), prev(), distance()
- gives all iterators some abilities usually provided only for random-access iterators
  - to step more than one element forward (or backward)
  - to process the difference between iterators

advance()

```
void advance (InputIterator& pos, Dist n)
```

https://en.cppreference.com/w/cpp/iterator/advance

- Modifies the iterator `pos`
- Increments (or decrements) `pos` `n` times
- lets the iterator step forward (or backward) more than one element

- Still an O(n) operation for lists and O(1) for vectors and arrays

next() and prev()

```
ForwardIterator       next (ForwardIterator       pos, Dist n=1)
BidirectionalIterator prev (BidirectionalIterator pos, Dist n=1)
```

https://en.cppreference.com/w/cpp/iterator/next

https://en.cppreference.com/w/cpp/iterator/prev

- Returns the position `pos` would have if moved forward (next()) ir backwards (prev()) n positions.
- Does not modifies the iterator `pos`
- next() works with forward, bidirectional, or random-access iterator
- prev() works with bidirectional, or random-access iterators, but not forward iterators

distance()

```
Dist distance (InputIterator pos1, InputIterator pos2)
```

https://en.cppreference.com/w/cpp/iterator/distance

- Returns the difference between two iterators
- Still an O(n) operation for lists and O(1) for vectors and arrays
- Consider:
  - std::distance(c.begin(), c.end()) == c.size()
  - std::distance(c.end(), c.begin()) is a logic error

Example of Iterator *Use*

```
List<string> list;
List<string>::Iterator iterator;

iterator = list.begin();
while (iterator != list.end()) {
 cout << *iterator << endl;
 ++iterator;
}

for(iterator = list.begin(); iterator != list.end(); ++iterator)
{
 cout << *iterator << endl;
}
```

Iterator *Implementation*

```
template <typename Data_t>
class DLinkedList<Data_t>::Iterator  // Forward Iterator
{
  friend class DLinkedList<Data_t>;

  public:
    // Compiler synthesized constructors and destructor are fine, just what we
    // want (shallow copies, no ownership) but needed to explicitly say that
    // because there is also a user defined constructor
    Iterator           (                   )    = default;
    Iterator           ( const Iterator  & )    = default;
    Iterator           (       Iterator && )    = default;
    Iterator & operator=( const Iterator  & )    = default;
    Iterator & operator=(       Iterator && )    = default;
   ~Iterator           (                   )    = default;

    Iterator( Node * position );              // Implicit conversion constructor

    // Pre and post Increment operators move the position to the next node in the list
    Iterator & operator++();                  // advance the iterator one node (pre -increment)
    Iterator   operator++( int );             // advance the iterator one node (post-increment)

    // Pre and post Increment operators move the position to the next node in the list
    Iterator & operator--();                  // retreat the iterator one node (pre -decrement)
    Iterator   operator--( int );             // retreat the iterator one node (post-decrement)

    Iterator   next     ( size_t delta = 1 );  // Return an iterator delta nodes after this node (this iterator doesn't change)
    Iterator   operator+( size_t rhs       );  // Return an iterator delta nodes after this node (this iterator doesn't change)

    Iterator   prev     ( size_t delta = 1 );  // Return an iterator delta nodes after this node (this iterator doesn't change)
    Iterator   operator-( size_t rhs       );  // Return an iterator delta nodes after this node (this iterator doesn't change)

    // Dereferencing and member access operators provide access to data. The
    // iterator can be constant or non-constant, but the iterator, by
    // definition, points to a non-constant linked list.
    Data_t & operator* () const;
    Data_t * operator->() const;

    // Equality operators
    bool operator==( const Iterator & rhs ) const;
    bool operator!=( const Iterator & rhs ) const;

  private:
    Node * node = nullptr;
};
```

operator++

```
template <typename Data_t>
typename DLinkedList<Data_t>::Iterator & DLinkedList<Data_t>::Iterator::operator++()      // pre-increment
{
  if( node == nullptr ) throw std::invalid_argument( "Attempt to increment null Iterator" );
  node = node->next;
  return *this;
}

template <typename Data_t>
typename DLinkedList<Data_t>::Iterator DLinkedList<Data_t>::Iterator::operator++( int ) // post-increment
{
  Iterator temp( *this );
  operator++();  // Delegate to pre-increment leveraging error checking
  return temp;
}
```

operator--

```
template <typename Data_t>
typename DLinkedList<Data_t>::Iterator & DLinkedList<Data_t>::Iterator::operator--()   // pre -decrement
{
  if( node == nullptr ) throw std::invalid_argument( "Attempt to decrement null Iterator" );
  node = node->prev;
  return *this;
}

template <typename Data_t>
typename DLinkedList<Data_t>::Iterator   DLinkedList<Data_t>::Iterator::operator--( int ) // post-decrement
{
  Iterator temp( *this );
  operator--();  // Delegate to pre-decrement leveraging error checking
  return temp;
}
```

operator*

```
template <typename Data_t>
Data_t & DLinkedList<Data_t>::Iterator::operator* () const
{
  if( node == nullptr ) throw std::invalid_argument( "Attempt to dereference null Iterator" );
  return node->data;
}
```

operator->

```
template <typename Data_t>
Data_t * DLinkedList<Data_t>::Iterator::operator->() const
{
  if( node == nullptr ) throw std::invalid_argument( "Attempt to dereference null Iterator" );
  return &(node->data);
}
```

operator==

```
template <typename Data_t>
bool DLinkedList<Data_t>::Iterator::operator==( const Iterator & rhs ) const
{ return node == rhs.node; }
```

operator!=

```
template <typename Data_t>
bool DLinkedList<Data_t>::Iterator::operator!=( const Iterator & rhs ) const
{ return !(*this == rhs);  }
```