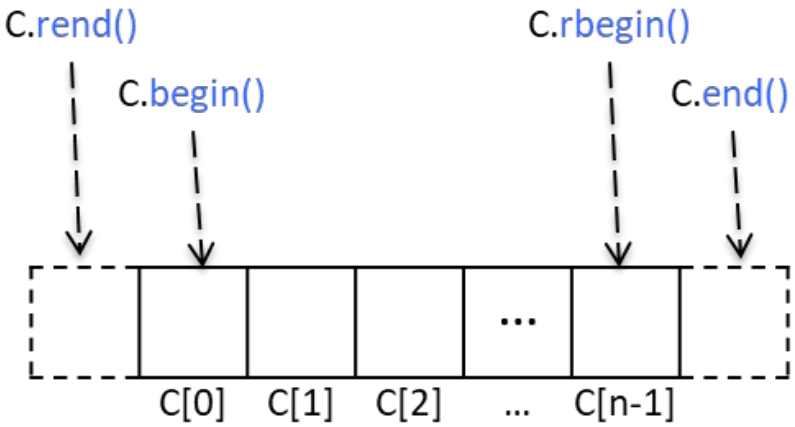
CALIFORNIA STATE UNIVERSITY
FULLERTON

# CPSC 131 – Data Structures

## Iterators

*Professor T. L. Bettens*
*Spring 2023*



# 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

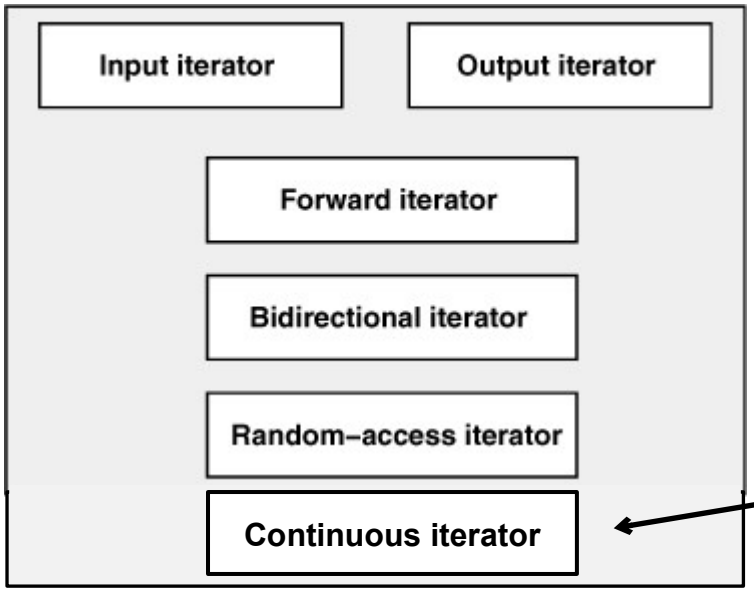
```
for (p = C.begin(); p != C.end(); ++p)
{
  p->do_something()
}
```

# 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

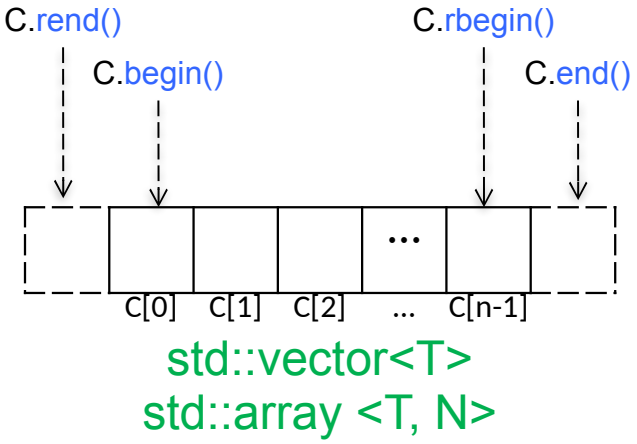
## Random-Access Iterator

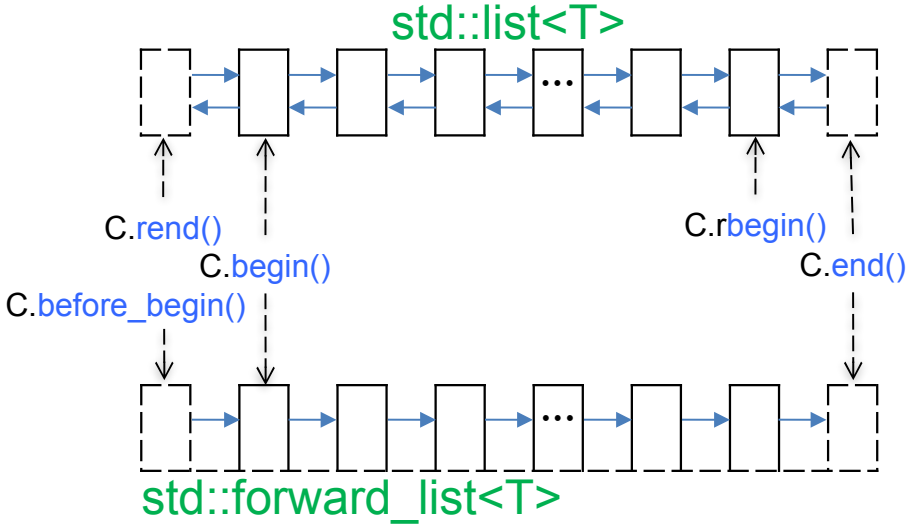
## Everything a Bidirectional can do, plus

| 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 (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 (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()`) or backwards (`prev()`) n positions.
- Does not modify 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 *Useage*

```
DoublyLinkedList<string> list;
DoublyLinkedList<string>::iterator itor;  //iterator is a
nested type

// Example 1:
itor = list.begin();
while (itor != list.end())
{
  std::cout << *itor << std::endl;
  ++itor;
}

// Example 2:  (equivalent, but with a traditional for loop)
for (itor = list.begin();  itor != list.end(); ++itor)
{
  cout << *itor << endl;
}
```

# Example of Iterator *Use*

**Display every other element of a doubly linked list in reverse using reverse iterators.**

```
int main()
{
  std::list<char> letters = {'A', 'B', 'C', 'D', 'E', 'F',
'G'};

  for( auto p = letters.rbegin(); p != letters.rend(); ++p)
  {
    std::cout << *p << '\n';
    if( std::next(p) != letters.rend() ) ++p; //be careful not
to
  }                                           //over increment
}
```

# Example of Iterator *Use*

## Given the following definition, which element is displayed by the expression listed on the left?

```
std::list<char> col = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
```

| | |
|---|---|
| `std::cout << *( col.begin() + 1 ) << '\n';` | Syntax Error |
| `std::cout << *( ++col.begin() ) << '\n';` | B |
| `std::cout << col.rbegin()[4] << '\n';` | Syntax Error |
| `std::cout << *( col.crend() ) << '\n';` | Logic Error |
| `std::cout << *( col.before_begin() ) << '\n';` | Syntax Error |
| `std::cout << col[std::distance( col.begin() + 1, col.end() - 3 )] << '\n';` | Syntax Error |
| `std::cout << *( std::next( col.begin(), 3 ) ) << '\n';` | D |
| `auto p = col.end();`<br>`std::advance( p, -2 );`<br>`std::cout << *p << '\n';`<br>*Logic Error if implemented as a null terminated list* | F |
| `auto p = col.rend();`<br>`std::advance( p, -5 );`<br>`std::cout << *p << '\n';`<br>*Logic Error if implemented as a null terminated list* | E |

# Example of Iterator *Use*

**Given the following definition, which element is displayed by the expression listed on the left?**

```
std::vector<char> col = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
```

| | |
|---|---|
| `std::cout << *( col.begin() + 1 ) << '\n';` | B |
| `std::cout << *( ++col.begin() ) << '\n';` | B |
| `std::cout << col.rbegin()[4] << '\n';` | C |
| `std::cout << *( col.crend() ) << '\n';` | Logic Error |
| `std::cout << *( col.before_begin() ) << '\n';` | Syntax Error |
| `std::cout << col[std::distance( col.begin() + 1, col.end() - 3 )] << '\n';` | D |
| `std::cout << *( std::next( col.begin(), 3 ) ) << '\n';` | D |
| `auto p = col.end();`<br>`std::advance( p, -2 );`<br>`std::cout << *p << '\n';` | F |
| `auto p = col.rend();`<br>`std::advance( p, -5 );`<br>`std::cout << *p << '\n';` | E |

# Iterator Interface

```
template<typename T1>   template<typename T2>
class DoublyLinkedList<T1>::Iterator_type
{
  friend class DoublyLinkedList<T1>;

  public:
    // Iterator Type Traits - Boilerplate stuff so the iterator can be used with the rest of the standard library
    using iterator_category = std::bidirectional_iterator_tag;
    using value_type        = T2;
    using difference_type   = std::ptrdiff_t;
    using pointer           = std::conditional_t< std::is_const_v<T2>, T2 const *, T2 *>;
    using reference         = std::conditional_t< std::is_const_v<T2>, T2 const &, T2 &>;


    // 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_type           (                             ) = delete;          // Default constructed Iterator_type not allowed
    Iterator_type           ( iterator      const & other );                   // Copy constructor when T is non-const, Conversion constructor
when T is const
// Pre and post Increment operators move the position to the next node in the list
    Iterator_type & operator++();                                              // advance the iterator one node (pre -increment)
    Iterator_type   operator++( int );                                         // advance the iterator one node (post-increment)

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

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

    // Equality operators
    bool operator==( Iterator_type const & rhs ) const;                        // Symmetrically compares all const & non-const iterator
combinations,
                                                                               elp of the Conversion constructor above


  private:
    // Member attributes
    Node * _nodePtr = nullptr;

    // Helper functions
    Iterator_type( Node * position );                                          // Implicit conversion constructor from pointer-to-Node to
iterator-to-Node

};  // DoublyLinkedList<T>::Iterator_type
```

Iterator has-a private (hidden) pointer-to-node

## operator++

```
// operator++   pre-increment
template<typename T1>   template<typename T2>
typename DoublyLinkedList<T1>::template Iterator_type<T2> &   DoublyLinkedList<T1>::Iterator_type<T2>::operator++()
{
  _nodePtr = _nodePtr->_next;
  return *this;
}

// operator++   post-increment
template<typename T1>   template<typename T2>
typename DoublyLinkedList<T1>::template Iterator_type<T2>   DoublyLinkedList<T1>::Iterator_type<T2>::operator++( int )
{
  auto temp{ *this };
  operator++();           // Delegate to pre-increment leveraging error checking
  return temp;
}
```

Notice how post-increment creates and destroys a temp object, but the pre-increment doesn't

## operator--

```
// operator--   pre-decrement
template<typename T1>   template<typename T2>
typename DoublyLinkedList<T1>::template Iterator_type<T2> &   DoublyLinkedList<T1>::Iterator_type<T2>::operator--()
{
  _nodePtr = _nodePtr->_prev;
  return *this;
}


// operator--   post-decrement
template<typename T1>   template<typename T2>
typename DoublyLinkedList<T1>::template Iterator_type<T2>    DoublyLinkedList<T1>::Iterator_type<T2>::operator--( int )
{
  auto temp( *this );
  operator--();           // Delegate to pre-decrement leveraging error checking
  return temp;
}
```

**operator***

```
// operator*
template<typename T1>   template<typename T2> typename DoublyLinkedList<T1>::template
Iterator_type<T2>::reference   DoublyLinkedList<T1>::Iterator_type<T2>::operator*()
const
{
   return _nodePtr->_data;
}
```

Return a reference to the Node's data (not the Node itself)

**operator->**

```
// operator->
template<typename T1>   template<typename T2> typename DoublyLinkedList<T1>::template
Iterator_type<T2>::pointer   DoublyLinkedList<T1>::Iterator_type<T2>::operator->()
const
{
   return &(_nodePtr->_data);
}
```

Return a pointer to the Node's data (not the Node itself)

## operator==

```
 // operator==
 template<typename T1>   template<typename T2>
 bool DoublyLinkedList<T1>::Iterator_type<T2>::operator==( Iterator_type const & rhs )
const
 { return _nodePtr == rhs._nodePtr; }
```