ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Module 4

Iterators

Dayu Wang

- **Linked list does not support index.**

→ **How to iterate through a `List`?**

→ **You cannot use `D_Node` for iteration.**

```
private:
    /** A doubly-linked list node */
    struct D_Node;
```

→ **There is no `D_Node` class outside the `List` class.**

→ **People use another data structure to iterate through linked lists.**

The implemented data structure that be used to iterate through linked lists is called iterator.

- **What is iterator?**

→ **Iterator can be understood as cursor (position marker).**

In a word processor (e.g., Microsoft Word), user uses a cursor to represent the current position in the document.

→ **What are the properties of an iterator?**

- **In C++, iterators are always on elements, never between elements.**
- **An iterator points to an element in the data structure.**
- **An iterator can move forward (advance).**

  If an iterator is at the end of the data structure, then it cannot move forward.
- **An iterator can move backward.**

  If an iterator is at the beginning of the data structure, then it cannot move backward.
- **User can insert an element at the iterator's position.**
- **User can delete the element the iterator is on (DELETE key).**

- **The Iterator Class In Linked List**

→ **Iterator is a class inside some data structure.**

- **List iterator**
- **Vector iterator**

→ **Private section**

In "List.h"

```
private:
    // Data fields
    const List<T>* parent;  // A pointer to the parent list
    D_Node* current;  // A pointer to the node the iterator is on
    size_t offset;  // Position compared to the first element

    // Constructor
    Iterator(const List<T>*, D_Node*, size_t);

    friend class List<T>;
```

```
// Constructor of iterator
template<class T>
List<T>::Iterator::Iterator(const List<T>* parent, D_Node* current, size_t
offset) : parent(parent), current(current), offset(offset) {}  // Time
complexity: O(1)
```

→ Overloading operators for iterators

Operator	Behavior
`++ () / ++ (int)`	Moves the iterator forward for one position.
`-- () / -- (int)`	Moves the iterator backward for one position.
`+ (int n)`	Creates an iterator that is **n** positions after current position.
`+= (int n)`	Moves the iterator forward for **n** positions.
`- (int n)`	Creates an iterator that is **n** positions before current position.
`-= (int n)`	Moves the iterator backward for **n** positions.
`- (Iterator)`	Finds the position difference of two iterators.
`* ()`	Returns the element the iterator is pointing to.
`-> ()`	Returns a pointer to the element the iterator is pointing to.
`== (Iterator)`	Tests whether 2 iterators are pointing to the same element.
`!= (Iterator)`	Tests whether 2 iterators are pointing to different elements.

→ Increment operator

```
// Moves the iterator forward for one position (prefix).
template<class T>
typename List<T>::Iterator& List<T>::Iterator::operator ++ () {
    if (!current) { throw exception("Iterator out of range"); }
    current = current->next;
    offset++;
    return *this;
}  // Time complexity: O(1)
// Moves the iterator forward for one position (postfix).
template<class T>
typename List<T>::Iterator List<T>::Iterator::operator ++ (int) {
    typename List<T>::Iterator result = *this;
    ++(*this);
    return result;
}  // Time complexity: O(1)
```

Why we need typename?

- **Class** Iterator **is an inner class in** List**.**
- **If the "outer class" (List) is a template class, the compiler may be confused when we call the "inner class" (Iterator).**
- **Using keyword typename to tell the compiler that we are using inner class** Iterator **as a type.**

→ Decrement operator

```cpp
// Moves the iterator backward for one position (prefix).
template<class T>
typename List<T>::Iterator& List<T>::Iterator::operator -- () {
    if (current == parent->head) {
        throw exception("Iterator out of range");
    }
    current = current ? current->prev : parent->tail;
    offset--;
    return *this;
}  // Time complexity: O(1)

// Moves the iterator backward for one position (postfix).
template<class T>
typename List<T>::Iterator List<T>::Iterator::operator -- (int) {
    typename List<T>::Iterator result = *this;
    --(*this);
    return result;
}  // Time complexity: O(1)
```

→ Dereferencing operators

```cpp
// Returns the element at current iterator position (lvalue).
template<class T>
T& List<T>::Iterator::operator * () {
    if (!current) { throw exception("Dereferencing NULL pointer"); }
    return current->data;
}  // Time complexity: O(1)

// Returns the element at current iterator position (rvalue).
template<class T>
const T& List<T>::Iterator::operator * () const {
    if (!current) { throw exception("Dereferencing NULL pointer"); }
    return current->data;
}  // Time complexity: O(1)

// Returns a pointer to the element at current iterator position.
template<class T>
T* List<T>::Iterator::operator -> () {
    if (!current) { throw exception("Dereferencing NULL pointer"); }
    return &(current->data);
}  // Time complexity: O(1)
```

→ Other class-member functions related to iterators

Function	Behavior
`Iterator begin() const;`	Creates an iterator positioned at the beginning of the data structure.
`Iterator end() const;`	Creates an iterator positioned **right after the last element** of the data structure.
`Iterator& insert(Iterator&, const T&);`	Inserts an element to the data structure at the iterator position.
`Iterator& erase(Iterator&);`	Deletes the element in the data structure at the iterator position.
`Iterator find(const T&) const;`	Searches for a target value in the data structure and returns an iterator positioned at that value (if found); or positioned right after the last element of the data structure (if **not** found).

```
// Generates an iterator on the first element of the list.
template<class T>
typename List<T>::Iterator List<T>::begin() const {
    return Iterator(this, head, 0);
}  // Time complexity: O(1)
```

```
// Generates an iterator just after the last element of the list.
template<class T>
typename List<T>::Iterator List<T>::end() const {
    return Iterator(this, NULL, size());
}  // Time complexity: O(1)
```

```cpp
// Inserts an element at iterator position.
template<class T>
typename List<T>::Iterator& List<T>::insert(Iterator& pos, const T&
value) {
    if (pos == begin()) {
        push_front(value);
        pos = ++begin();
    } else if (pos == end()) {
        push_back(value);
        pos = end();
    } else {
        D_Node* new_node = new D_Node(value);
        new_node->prev = pos.current->prev;
        new_node->prev->next = new_node;
        new_node->next = pos.current;
        new_node->next->prev = new_node;
        num_of_items++;
        pos.offset++;
    }
    return pos;
}  // Time complexity: O(1)
```

```cpp
// Deletes the element at iterator position.
template<class T>
typename List<T>::Iterator& List<T>::erase(Iterator& pos) {
    if (pos == end()) {
        throw exception("Dereferencing NULL pointer");
    }
    if (pos == begin()) {
        pop_front();
        pos = begin();
    } else if (pos == --end()) {
        pop_back();
        pos = end();
    } else {
        D_Node* to_be_deleted = pos.current;
        pos.current->prev->next = pos.current->next;
        pos.current->next->prev = pos.current->prev;
        pos.current = pos.current->next;
        num_of_items--;
        delete to_be_deleted;
    }
    return pos;
}  // Time complexity: O(1)
```

```
// Searches for a value in the list.
template<class T>
typename List<T>::Iterator List<T>::find(const T& value) const {
    for (Iterator it = begin(); it != end(); it++) {
        if (*it == value) { return it; }
    }
    return end();
}  // Time complexity: O(n)
```

- **Const Iterator**

→ **The const_iterator class can be used to iterate over the list.**

However, the list itself cannot be changed.

→ **If the linked list is passed by reference (modifiable), then you can use either iterator or const_iterator to iterate over the list.**

- **If you want to change the list, then you need to use iterator.**
- **If you do not want to change the list, then you need to use const_iterator (good habit).**

```
int max_val(list<int>& li) {
    int result = INT_MIN;
    for (list<int>::iterator it = li.begin(); it != li.end(); it++) {
        if (*it > result) { result = *it; }
    }
    return result;
}
```

OK, but not good habit

```
int max_val(list<int>& li) {
    int result = INT_MIN;
    for (list<int>::const_iterator it = li.begin(); it != li.end(); it++) {
        if (*it > result) { result = *it; }
    }
    return result;
}
```

Good habit

→ **If the linked list is passed by const reference (unmodifiable), then you must use const_iterator to iterate over the list.**

```
int max_val(const list<int>& li) {
    int result = INT_MIN;
    for (list<int>::iterator it = li.begin(); it != li.end(); it++) {
        if (*it > result) { result = *it; }
    }
    return result;
}
```

Code will not compile.

```
int max_val(const list<int>& li) {
    int result = INT_MIN;
    for (list<int>::const_iterator it = li.begin(); it != li.end(); it++) {
        if (*it > result) { result = *it; }
    }
    return result;
}
```

Correct

- **The Iterator Class In Vector**

→ **Private section**

In "Vector.h"

```
private:
    // Data field
    const Vector<T>* parent;  // A pointer to the parent vector
    size_t index;  // Index of the current element

    // Constructor
    Iterator(const Vector<T>*, size_t);

    friend class Vector<T>;
```

```
// Constructor of class "Iterator"
template<class T>
Vector<T>::Iterator::Iterator(const Vector<T>* parent, size_t index) :
    parent(parent), index(index) {}
// Time complexity: O(1)
```

→ Overloading operators for iterators

Operator	Behavior
`++ () / ++ (int)`	Moves the iterator forward for one position.
`-- () / -- (int)`	Moves the iterator backward for one position.
`+ (int n)`	Creates an iterator that is **n** positions after current position.
`+= (int n)`	Moves the iterator forward for **n** positions.
`- (int n)`	Creates an iterator that is **n** positions before current position.
`-= (int n)`	Moves the iterator backward for **n** positions.
`- (Iterator)`	Finds the position difference of two iterators.
`* ()`	Returns the element the iterator is pointing to.
`-> ()`	Returns a pointer to the element the iterator is pointing to.
`== (Iterator)`	Tests whether 2 iterators are pointing to the same element.
`!= (Iterator)`	Tests whether 2 iterators are pointing to different elements.

→ Increment operator

```
// Moves the iterator forward for one position (prefix).
template<class T>
typename Vector<T>::Iterator& Vector<T>::Iterator::operator ++ () {
    if (index == parent->size()) {
        throw exception("Iterator out of range");
    }
    index++;
    return *this;
}  // Time complexity: O(1)

// Moves the iterator forward for one position (postfix).
template<class T>
typename Vector<T>::Iterator Vector<T>::Iterator::operator ++ (int) {
    typename Vector<T>::Iterator result = *this;
    ++(*this);
    return result;
}  // Time complexity: O(1)
```

→ Decrement operator

```
// Moves the iterator backward for one position (prefix).
template<class T>
typename Vector<T>::Iterator& Vector<T>::Iterator::operator -- () {
    if (!index) { throw exception("Iterator out of range"); }
    index--;
    return *this;
}  // Time complexity: O(1)

// Moves the iterator backward for one position (postfix).
template<class T>
typename Vector<T>::Iterator Vector<T>::Iterator::operator -- (int) {
    typename Vector<T>::Iterator result = *this;
    --(*this);
    return result;
}  // Time complexity: O(1)
```

→ Dereferencing operators

```
// Returns the element at iterator position (lvalue).
template<class T>
T& Vector<T>::Iterator::operator * () {
    if (index == parent->size()) {
        throw exception("Index out of range");
    }
    return parent->data[index];
}  // Time complexity: O(1)

// Returns the element at iterator position (rvalue).
template<class T>
const T& Vector<T>::Iterator::operator * () const {
    if (index == parent->size()) {
        throw exception("Index out of range");
    }
    return parent->data[index];
}  // Time complexity: O(1)

// Returns a pointer to the element at iterator position.
template<class T>
T* Vector<T>::Iterator::operator -> () {
    if (index == size()) { throw exception("Index out of range"); }
    return &(parent->data[index]);
}  // Time complexity: O(1)
```

→ Other class-member functions related to iterators

Function	Behavior
`Iterator begin() const;`	Creates an iterator positioned at the beginning of the data structure.
`Iterator end() const;`	Creates an iterator positioned **right after the last element** of the data structure.
`Iterator& insert(Iterator&, const T&);`	Inserts an element to the data structure at the iterator position.
`Iterator& erase(Iterator&);`	Deletes the element in the data structure at the iterator position.
`Iterator find(const T&) const;`	Searches for a target value in the data structure and returns an iterator positioned at that value (if found); or positioned right after the last element of the data structure (if **not** found).

```
// Generates an iterator on the first element of the vector.
template<class T>
typename Vector<T>::Iterator Vector<T>::begin() const {
    return Iterator(this, 0);
}  // Time complexity: O(1)
```

```
// Generates an iterator just after the last element of the vector.
template<class T>
typename Vector<T>::Iterator Vector<T>::end() const {
    return Iterator(this, size());
}  // Time complexity: O(1)
```

```
// Inserts an element at iterator position.
template<class T>
typename Vector<T>::Iterator& Vector<T>::insert(Iterator& pos, const
T& value) {
    if (size() == capacity) { resize(); }
    for (size_t i = size(); i > pos.index; i--) {
        data[i] = data[i - 1];
    }
    data[pos.index] = value;
    num_of_items++;
    return ++pos;
}  // Time complexity: O(n)
```

- **In linked list,** `insert()` **function has time complexity** $O(1)$**.**
- **In vector,** `insert()` **function has time complexity** $O(n)$**.**
- **Linked list supports fast insertion.**

```
// Deletes the element at iterator position.
template<class T>
typename const Vector<T>::Iterator& Vector<T>::erase(const Iterator&
pos) {
    for (size_t i = pos.index; i < size() - 1; i++) {
        data[i] = data[i + 1];
    }
    num_of_items--;
    return pos;
}  // Time complexity: O(n)
```

- **In linked list,** `erase()` **function has time complexity** $O(1)$**.**
- **In vector,** `erase()` **function has time complexity** $O(n)$**.**
- **Linked list supports fast deletion.**

```cpp
// Finds a target value in the vector.
template<class T>
typename Vector<T>::Iterator Vector<T>::find(const T& value) const {
    for (size_t i = 0; i < size(); i++) {
        if (at(i) == value) { return Iterator(this, i); }
    }
    return end();
}  // Time complexity: O(n)
```

If the vector is sorted (e.g., Ordered_Vector class):

```cpp
template<class T>
int Ordered_Vector<T>::find(const T& target) const {
    size_t bottom = 0, top = size() - 1;
    while (bottom <= top) {
        size_t mid = (bottom + top) / 2;
        if (target < at(mid)) { top = mid - 1; }
        else if (target > at(mid)) { bottom = mid + 1; }
        else { return Iterator(this, mid); }
    }
    return end();
}  // Time complexity: O(log(n))
```

- **Ordered vector supports fast search.**
- **Search in linked list is slow.**