22 Iterator

我们发现对于 Linked_list，我们目前并没有一个好办法来 traverse 它.

我们或许会尝试像：

```
for (List<int>::Node *np = list.first; np != nullptr; np = np->next)
  cout << np->datum << endl;
}
```

但是这根本不会 compile，因为Node 和 first 都是 interface，是 user 无法 access 的（都是 private的）

我们回忆如何用 pointer 来 traverse 一个 array:

```
int *end = arr + SIZE;
for (int *ptr = arr; ptr != end; ++ptr)
  cout << *ptr << endl;
```

而现在我们要介绍一种方法：在我们想要 iterate 的 class (比如 Linked List) 中写一个 **public** 的 **nested class** `Iterator`，通过 overload 一些运算符来**让 Iterator 起到像一个 pointer 一样的用途， traverse 整个 sequential container ADT object.**

22.1 `List<type>::Iterator`

22.1.1 Nested class `Iterator`: interface

friend class 关键词: 表示 `List` 这个 class 是 `Iterator` 的 friend class，可以 access `Iterator` 的 private members

```
template <typename T>
class List {
public:
  class Iterator {
    friend class List;  // 声明 friend: List 这个 class 可以 access Iterator 的 private members
  public:
    Iterator(); // ctor
    T & operator*() const;  // overload *
    Iterator & operator++();  // overload ++
    bool operator==(Iterator rhs) const;  // overload ==
    bool operator!=(Iterator rhs) const;  // overload !=
    //...
  private:
    Node *node_ptr;
    Iterator(Node *np);
  }

  Iterator begin(); // at beginning
  Iterator end(); // "past the end"
  //...
};
```

22.1.2 `Iterator` 通过 overload operators 来模拟 ptr arithmetic

我们通过在 Iterator 中，overload 以下这几个函数来达成

1. `operator *`：模拟 dereference
2. `operator ++`： 模拟 pointer arithmetic
3. `operator ==`，`operator !=`：两个 Iterator 的 Node * 是否指向同一个 Node

我们想要的实现的效果是：

假设 `it` 是一个 `Iterator`，那么我们可以 `*it` 来获取当前的 Node，我们可以 `++it` 来让它移动到 `List` 中的下一个 `Node` (即让它的 member `node_ptr` 的值从当前 `Node` 的地址移动到下一个 `Node` 的地址.)

22.2 Iterator 的 implementations

22.2.1 Iterator 的 ctor `List<T>::Iterator::Iterator(\*...*\)`

```
template <typename T>
List<T>::Iterator::Iterator()
    : node_ptr(nullptr) { }

template <typename T>
List<T>::Iterator::Iterator(Node *np)
    : node_ptr(np) { }
```

22.2.2 对 `List` 初始化一个 iterator: 返回 Iterator的 member function

22.2.2.1 `typename` keyword

这里是一个新的语法。

在任何 template function (不止是 member function) 中，只要我们需要获取的值（比如函数返回值，或者 initialize 一个该 nested class 的 object）的类型是一个 **nested type inside a type which depends on template parameter**，那么我们就要**在该语句前加上 `typename` 这个 keyword.**

比如我们下面要写的两个`List` 的member function： 返回一个 `List<T>::Iterator` 的 instance 的 `begin()` 和 `end()` 函数。

由于

1. 这两个函数的返回值的 type 为`Iterator`,
2. `Iterator` 是 `List<T>` 中的 nested type.
3. `List<T>` 取决于 template parameter `T`.

因而我们必须要在函数 definition 前加上 `typename` 这一关键词

```
template <typename T>
typename List<T>::Iterator List<T>::begin() {
  return Iterator(first);
}

template <typename T>
typename List<T>::Iterator List<T>::end() {
  return Iterator();
}
```

不止如此，我们如果写一个普通的 template function,

```
template <typename T>
void func() {
  IntList::Iterator it1;  // does not depend on T
  List<int>::Iterator it2; // does not depend on T
  typename List<T>::Iterator it3; // depend on T
}
```

其中我们 initialize 了一个取决于 T 的 type `List` 中的一个 nested type `Iterator` 的 instance，于是在这句 initialization 前我们也要加上 `typename` keyword.

而如果我们需要一个 template type 的 nested type 的 instance，但是我们为这个 template class 确定了 T，那么我们就不需要加上 `typename`，因为这个时候这个 class 已经不 depend on T 了。

22.2.2.2 `List<T>` 的 `begin` 和 `end` function: return 一个相应位置的 Iterator

```
template <typename T>
typename List<T>::Iterator List<T>::begin() {
  return Iterator(first);
  // return an iterator at the beginning.
}

template <typename T>
typename List<T>::Iterator List<T>::end() {
  return Iterator();
  // return an iterator "past the end".
}
```

22.2.3 Iterator: `*` 的 overload

从这里开始就是 nested class `Itertor` 的函数了。由于是 nested class，在 definition 时我们需要先加上 `List<T>` 的 scope 再加上 `Iterator` 的 scope:

```
template <typename T>
T & List<T>::Iterator::operator*() const {
  assert(node_ptr); // 确保 != nullptr, 即list不为空, 且Iterator 当前在 list 之内
  return node_ptr->datum; // 返回 Iterator 的 node_ptr 指向的 Node 值
}
```

注意：我们 overload 的 `*` 返回的是 reference，也就是说我们既可以用它来获取值，又可以用它来改变原来的值。这正模拟了 `*` 运算符原本的用处。

22.2.4 Iterator: `++` 的 overload

注意：**`operator++` 做了两件事，一件事是 increment，另一件事是返回值**（pre-increment `operator++` 返回的是 reference，post-increment `operator++` 返回的是 value.）

因而我们这里需要加上 `typename` 关键词！

22.2.4.1 pre-increment evaluated: `++it`

在这里我们 overload 的是 pre-increment 的 `++`，用法为 `++it`.

pre-increment 的 `++` 会先 increment 再返回值，并且返回的是引用值。

```
template <typename T>
typename List<T>::Iterator & List<T>::Iterator::operator++() {
  assert(node_ptr);
  node_ptr = node_ptr->next;
  return *this;
}
```

22.2.4.2 post-increment: `it++`

我们也可以写一个 post-increment 的 `operator++` 的 overload.

带有参数 `int`，并且不返回 reference 而是返回值的是 post-increment 的 `++`，用法为 `it++`.

Post-increment 的 `++` 会返回 increment 前的值，且是一个 copy.

```
template <typename T>
typename List<T>::Iterator List<T>::Iterator::operator++(int) {
  assert(node_ptr);
  Node *copy = node_ptr;
  node_ptr = node_ptr->next;
  return *copy;
}
```

22.2.5 Iterator: `==` 和 `!=` 的 overload

```
template <typename T>
bool List<T>::Iterator::operator==(Iterator rhs) const {
  return node_ptr == rhs.node_ptr;
}

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

22.3 check for the Big Three 以及使用 `Iterator`

22.3.1 check for the Big Three

我们发现，我们并没有为 `Iterator` custom Big Three. 这是合理的，因为我们在 implement `Iterator` 时并没有动用 dynamic memory.

因而我们做的任何 `Iterator` 的 copy 都是 shallow copy 而不是 deep copy.

```
List<int>::Iterator it2 = it1;
```

这种做法是正确的，因为我们如果要 deep copy `Iterator` 就会建出莫名其妙的新的 linked list. 这不是我们想要的。

22.3.2 完成并使用 `Iterator` 来 traverse 一个 linked list

我们在 check 完 Big Three 之后就完成了 `Iterator` 的构建。

实际上我们写的 `Iterator` matches the `Iterator` in the STL.

通过为不同的 containers define 相同 interface 的 iterators，我们可以对不同的 containers 使用相同的 looping code.

现在我们用我们写完的 `Iterator` 来 traverse 一个 linked list:

```
int main() {
  List<int> list;       // ( )
  list.push_front(1);   // ( 1 )
  list.push_front(3);   // ( 3 2 1 )
  list.front() = 4;     // ( 4 2 1 )
  list.insert(99, 0);   // ( 99 4 2 1 )
  list.insert(88, 3);   // ( 99 4 2 88 1 )
  list.pop_front();     // ( 4 2 88 1 )

  for (List<int>::Iterator it = list.begin(); it != list.end(); ++it)
    cout << *it << endl;
    cout << endl;
  // 4
  // 2
  // 88
  // 1
}
```

22.3.3 Iterator Invalidation

```
int main() {
  List<int> list;
  // … fill with values
  List<int>::Iterator it = list.begin();
  List<int>::Iterator it2 = list.begin();
  cout << *it << endl; // OK
  cout << *it2 << endl; // OK

  list.pop_front(); // 删除了 it, it2 正 point to 的 element

  cout << *it << endl; // Undefined behavior
  cout << *it2 << endl; // Undefined behavior
}
```

Invalidated iterators 就像 dangling pointers，dereference 它们会导致 Undefined Behavior，很不安全。

`pop_front` 这种安全的操作，也会导致 iterator invalidation. 因而一个 function 的 documentation 应该明确写明 iterators 的类型，以及它会 invalidated 的情景。

22.3.4 Traversal 练习: Check for duplicates

```
template <typename T>
bool hasDuplicates(List<T> list) {
  auto end = list.end();
  for (auto i = list.begin(); i != end; ++i) {
    auto j = i;
    ++j;
    for ( ; j!=end; ++j) {
      if (*i == *j) return true;
    }
  }
  return false;
}
```