Module04-05
C++ 标准库：迭代器

- 数据结构简介
- 标准容器
- 常用算法简介
- 标准算法与函数对象
- → 迭代器
- 字符串
- I/O 流
- 数值

关于迭代器

- 关于迭代器 (Iterator)
  - 迭代器是指向序列元素的指针概念的一种抽象，关键属性：
    - 当前被指向的元素（间接，用操作符 * 和 -> 表示）
    - 指向下一个元素（增量，用操作符 ++ 表示）
    - 相等（用操作符 == 表示）
  - 各种标准容器都有其迭代器类型，如：
    - vector<int>::iterator 是 vector 的一个迭代器
    - int* 是 int[]（数组）的一个迭代器
  - 迭代器相关的类型定义于 <iterator> 头文件

迭代器操作

- 迭代器的操作

类别	Output	Input	Forward	Bidirectional	Random Access
简写	Out	In	For	Bi	Ran
Read:		=*p	=*p	=*p	=*p
Access:		->	->	->	-> []
Write:	*p=		*p=	*p=	*p=
Iteration:	++	++	++	++ --	++ -- + - += -=
Comparison:		== !=	== !=	== !=	== != < > <= >=

  - 两个函数：
    - `distance(In first, In last);` // 两个迭代器之间的距离（元素的个数）
    - `advance(In& i, Dist n);` // `i += n`

迭代器特征类

- 迭代器的特征类 - iterator_traits
  - iterator_traits 用来描述指向的对象和序列的一些信息，如：
    - 迭代器的类别
    - 迭代器所指的对象类型
    - 两个迭代器之间距离的类型等
  - iterator_traits 定义：

```
template<typename Iter>
struct iterator_traits {
    typedef typename Iter::iterator_category
iterator_category;
    typedef typename Iter::value_type value_type;
    typedef typename Iter::difference_type difference_type;
    typedef typename Iter::pointer pointer;
    typedef typename Iter::reference reference;
};
```

迭代器基类

- 迭代器基类 - iterator

```
template<typename Category, typename T, typename Distance =
ptrdiff_t, typename Pointer = T*, typename Reference = T&>
struct iterator {
    // One of the iterator_tags tag types.
    typedef Category iterator_category;
    // The type "pointed to" by the iterator.
    typedef T value_type;
    // Distance between iterators is represented as this
type.
    typedef Distance difference_type;
    // This type represents a pointer-to-value_type.
    typedef Pointer pointer;
    // This type represents a reference-to-value_type.
    typedef Reference reference;
};
```

迭代器类别

- 迭代器的类别 (Categories)
  - 随机存取迭代器 (Random Access Iterator)
  - 双向迭代器 (Bidirectional Iterator)
  - 前向迭代器 (Forward Iterator)
  - 输入迭代器 (Input Iteratror)
  - 输出迭代器 (Output Iterator)

插入器

- 插入器 (Inserters)
  - 3 个迭代器模板类和协助函数：

```
template<typename Container>
class back_insert_iterator: public
iterator<output_iterator_tag, void, void,
         void, void> {
};

template<typename Container>
inline back_insert_iterator<Container>
back_inserter(Container& x);
```

- 插入器 (Inserters)
  - 3 个迭代器模板类和协助函数（续）：

```
template<typename Container>
class front_insert_iterator: public
iterator<output_iterator_tag, void, void, void, void> {
};

template<typename Container>
inline front_insert_iterator<Container>
front_inserter(Container& x);

template<typename Container>
class insert_iterator: public iterator<output_iterator_tag,
void, void, void, void> {
};

template<typename Container, typename Iterator>
inline insert_iterator<Container>
inserter(Container& x, Iterator i);
```

插入器示例

- 插入器 (Inserters)
  - 示例：

```
list<int> ls;
front_inserter(ls) = 16;
front_inserter(ls) = 26;
copy(ls.begin(), ls.end(), ostream_iterator<int> (cout, " "));

vector<int> v;
// 注意：下面的操作有问题！
// copy(ls.begin(), ls.end(), v.begin()); // runtime crash!!
copy(ls.begin(), ls.end(), back_inserter(v));
copy(v.begin(), v.end(), ostream_iterator<int> (cout, " "));

inserter(v, ++v.begin()) = 128;
copy(v.begin(), v.end(), ostream_iterator<int> (cout, " "));
```

反向迭代器

- 反向迭代器 (reverse_iterator)
  - 很多标准容器都提供了反向迭代器，如：
    - rbegin() 和 rend() 返回的类型是 reverse_iterator

流迭代器

- 流迭代器 (stream iterator)
  - 4 个流迭代器：
    - ostream_iterator
    - istream_iterator
    - ostreambuf_iterator
    - istreambuf_iterator

流迭代器示例

- 流迭代器 (stream iterator)（续）
  - 示例：

```
ostream_iterator<int> oi(cout);
*oi = 7; // 输出 7 (cout << 7)
cout << endl;
++oi; // 准备下一次输出，不要忘记++
*oi = 12;
cout << endl;

istream_iterator<int> ii(cin);
int i = *ii;
++ii;
int j = *ii;

vector<int> v;
// 注意: istream_iterator<int> ()创建一个空迭代器，通常用作结束标志
copy(istream_iterator<int> (cin), istream_iterator<int> (),
        back_inserter(v));

copy(v.begin(), v.end(), ostream_iterator<int> (cout, " "));
```

迭代器 - Bjarne's Advices

- Bjarne's Advices
  - 在写一个算法时，设法确定需要用何种迭代器才能提供可接受的效率，并（只）使用这种迭代器所支持的操作符去描述算法
  - 利用 iterator_traits 为不同的迭代器类别描述适当的算法
  - 记住在 istream_iterator 和 ostream_iterator 的访问之间使用 ++
  - 用插入器 (inserter) 避免容器溢出