# CS100 Recitation 11

# 聚合类 (aggregate)

# 返回多个值

想返回 $(\min, \max)$ 两个值，怎么办?

```
??? getMinMax(const std::vector<int> &vec) {
  int min = INT_MAX, max = INT_MIN;
  for (auto x : vec) {
    // ...
  }
  return ???
}
```

# 返回多个值

C 里讲过的办法：创建一个结构体

- 在 C++ 里就是一种简单的类。

```
struct MinAndMax {
  int min;
  int max;
};
MinAndMax getMinMax(const std::vector<int> &vec) {
  int min = INT_MAX, max = INT_MIN;
  for (auto x : vec) {
    // ...
  }
  return /* 这里怎么写？ */
}
```

## 聚合类 (aggregate)

聚合类是那些所有 non-`static` 数据成员都是 public 、没有自定义构造函数且满足一些其它条件的类。

- 核心特点一：数据成员都是 public ，看起来就如同是几个变量打包在一起，这些变量本身就是接口。
- 核心特点二：没有自定义构造函数，初始化 = 对所有成员进行初始化。
- 其它条件，例如基类都是 public 继承，没有虚函数等。

这样的类从使用的角度讲，与 C 的 `struct` 非常接近。

# 使用大括号列表

```
struct MinAndMax {
  int min;
  int max;
};
MinAndMax result{a, b};
```

```
MinAndMax getMinMax(/* ... */) {
  int min = INT_MAX, max = INT_MIN;
  // ...
  return {min, max};
}
```

一般地，如果用 `{e1, e2, ...}` 初始化一个 `T` 类型的对象，例如 `T x{a, b, c};`：

- 如果 `T` 是一个聚合体，那么这是一种**聚合初始化 (aggregate initialization)**：`e1`, `e2`, ... 依次用来初始化第一个子对象、第二个子对象、……
- 否则，如果 `T` 含有接受 `std::initializer_list<V>` 的构造函数，则这个列表被传给这个构造函数。
- 否则，就如同 `T x(a, b, c);` ，走正常的构造函数（**直接初始化 (direct initialization)**）。

## 使用大括号列表

如果 `MinAndMax` 有一个用户提供的构造函数：

```
struct MinAndMax {
  int min;
  int max;
  MinAndMax(int min_, int max_) : min{min_}, max{max_} {}
};
```

那么 `MinAndMax` 就不再是聚合类，但是我们仍然可以用 `{a, b}` 初始化一个该类型的对象，也仍然可以 `return {min, max};`

总之，下面这种做法一定是可以避免、需要避免的：

```
MinAndMax result;
result.min = a;
result.max = b;
```

# 结构化绑定 (Structured binding)

# 结构化绑定 (Structured binding)

```
struct MinAndMax { int min, max; };
MinAndMax getMinMax(const std::vector<int> &);
```

调用者怎么获得这个返回值?

```
auto result = getMinMax(numbers);
std::cout << result.min << ' ' << result.max << '\n';
```

这当然可以，但是有更好的办法：**结构化绑定**

```
auto [minVal, maxVal] = getMinMax(numbers);
std::cout << minVal << ' ' << maxVal << '\n';
```

# 结构化绑定 (Structured binding)

结构化绑定的声明可以带有 `const` 、引用等，也可以放在基于范围的 `for` 语句里：

```
enum class Gender { Male, Female };
struct PersonInfo {
  std::string name;
  Gender gender;
  int birthYear;
};
void foo(const std::vector<PersonInfo> &persons) {
  for (const auto &[n, g, by] : persons) {
    // ...
  }
}
```

`std::pair`

## `std::pair`

定义在 `<utility>` 中

一个快速、随意的数据结构

```
std::pair<T1, T2> p1{a, b}; // 用圆括号也可以
auto p2 = std::make_pair(a, b); // 等价的写法
std::pair p3{a, b}; // 编译器根据 a 和 b 的类型来推导模板参数
                    // 这里用圆括号也可以，但 {} 更 modern
```

C++17 有了 CTAD，几乎不再需要 `std::make_pair` 了。

## `std::pair`

返回两个值：装进 `pair` 里就行

```
std::pair<int, std::string> foo() {
  // ...
  return {42, "hello"};
}
```

访问 `pair` 中的元素：`p.first`, `p.second`

```
auto p = foo();
std::cout << foo.first << ", " << foo.second << std::endl;
```

## `std::pair`

结构化绑定也可以用于 `std::pair` ：

```
auto [ival, sval] = foo();
std::cout << ival << ", " << sval << std::endl;
```

`std::pair` 有比较运算符：按照 `.first` 为第一关键字、`.second` 为第二关键字比较。

不要滥用 `std::pair` ！

# STL 总结、补充

# STL 概览

诞生于 1994

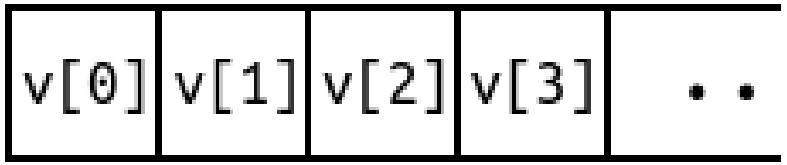
- 容器 containers：顺序 (sequence) 容器，关联 (associative) 容器
- 迭代器 iterators
- 算法 algorithms
- 适配器 adaptors：iterator adaptors, container adaptors
- 仿函数 function objects (functors)
- 空间分配器 allocators

# 容器

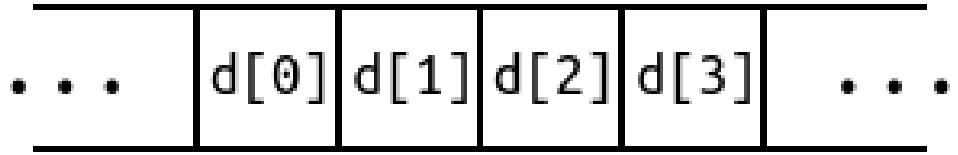
- 顺序容器 sequence containers
  - `vector`, `deque`, `list`, `array`, `forward_list`
- 关联容器 associative containers
  - `map`, `set`, `multimap`, `multiset`
  - `unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`

# 顺序容器

- `vector<T>`：可变长数组，支持在末尾快速地添加、删除元素，支持随机访问 (random access)

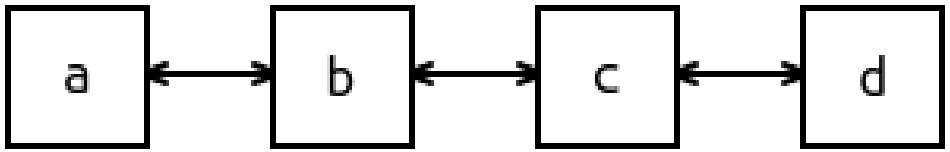



- `deque<T>`：双端队列 (**d**ouble **e**nded **que**ue)，支持在开头和末尾快速地添加、删除元素，支持随机访问

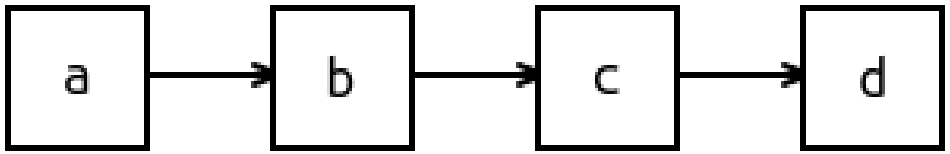


# 顺序容器

- `list<T>`：双向链表，支持在任意位置快速地添加、删除元素，支持双向遍历，不支持随机访问




- `forward_list<T>`：单向链表，支持在任意位置快速地添加、删除元素，仅支持单向遍历，不支持随机访问




- `array<T, N>`：内置数组 `T[N]` 的套壳，提供和其它 STL 容器一致的接口（包括迭代器等），可以直接拷贝，不会退化为 `T *`。
- 特别的：`string` 不是容器，但非常像一个容器

# 统一的接口：构造

1. `Container c`：一个空的容器
2. `Container c(n, x)`：`n` 个 `x`
3. `Container c(n)`：`n` 个元素，每个元素都被值初始化
   - `string` 不支持这一操作，为什么？
4. `Container c(begin, end)`：从迭代器范围 `[begin, end)` 中拷贝元素。

`array<T, N>` 只支持 1 ，为什么？

# 统一的接口：完整列表

相同的功能在所有容器上都具有相同的接口（除了 `forward_list` 有一点点特殊）

- 拷贝构造、拷贝赋值、移动构造、移动赋值
- `c.size()`, `c.empty()`, `c.resize(n)`
- `c.capacity()`, `c.reserve(n)`, `c.shrink_to_fit()`
- `c.push_back(x)`, `c.emplace_back(args...)`, `c.pop_back()`
- `c.push_front(x)`, `c.emplace_front(args...)`, `c.pop_front()`
- `c.at(i)`, `c[i]`, `c.front()`, `c.back()`
- `c.insert(pos, ...)`, `c.emplace(pos, args...)`, `c.erase(...)`, `c.clear()`

# 统一的接口

记忆的关键：

1. 理解每一种容器的底层数据结构，自然就明白为何支持/不支持某种操作
    - 为何链表不支持下标访问？为何 `vector` 不支持在开头添加元素？
2. 记住这些接口的名字
3. 如果一个操作应该被支持，它就必然叫那个名字、是那个用法

# `string` 和 `vector` 的“容量”

`string` 大概率和 `vector` 采用类似的增长方式，分配的内存可能比当前存储的元素所占用的内存大。

- 当前所拥有的内存能放下多少个元素，称为“容量” (capacity)，可以通过 `c.capacity()` 查询。
- `c.reserve(n)` 为至少 `n` 个元素预留内存。
  - 如果 `n <= c.capacity()`，什么都不会发生。
  - 否则，重新分配内存使得 `c` 能装得下至少 `n` 个元素。
  - **务必区分 `reserve` 和 `resize`。**
- `c.shrink_to_fit()`：**请求** `c` 释放多余的容量（可能重新分配一块更小的内存）
  - 这个函数在 `deque` 上也有。

# `insert` 和 `erase`

`c.insert(pos, ...)`，**`c.emplace(pos, args...)`**，其中 `pos` 是一个迭代器。

在 `pos` 所指的位置**之前**添加元素。

- `insert` 有很多种写法，可以 `c.insert(pos, x)`，`c.insert(pos, begin, end)`，`c.insert(pos, {a, b, c, ...})` 等等，用到的时候再查。

`c.erase(...)` 也有很多种写法，用到的时候再查。

# 特殊的 `forward_list`

`forward_list` 的功能完全被 `list` 包含，那为何我们还需要 `forward_list`？

**为了省时间，省空间。**

- 单向链表的结点比双向链表的结点少存一个指针
- 维护单向链表上的链接关系也比维护双向链表少一些操作

因此，`forward_list` 采取最简的实现：能省则省

- 它不能 `push_back` / `pop_back`。
  - 如果需要，你可以自己保存指向末尾元素的迭代器，然后用 `insert_after`
- 它甚至不支持 `size()`。如果需要，你可以自己用一个变量记录。

# 特殊的 `forward_list`

`insert`，`emplace` 和 `erase` 变成了 `insert_after`，`emplace_after` 和 `erase_after`

- 单向链表上，操作“下一个元素”比操作“当前元素”或“前一个元素”更方便。

# 越界检查

`c.at(i)` 在越界时抛出 `std::out_of_range` 异常

`c[i]`，`c.front()`，`c.back()`，`c.pop_back()`，`c.pop_front()` 统统不检查越界，一切为了效率。

但是我们在自己设计自己的容器时，不一定要采用标准库的这种方式。

- 也许下面这种设计更合理?

```
auto &operator[](size_type n) {
  assert(n < size());
  return data[n];
}
```

  - `assert` 在 Debug 模式下生效，而在 Release 模式下（定义了 `NDEBUG` 宏）无效，不会影响 Release 模式下的效率。

# 选择正确的容器

顺序容器：

- 能维持元素的先后顺序
- 某些情况下的插入、删除、查找可能较慢

关联容器（不带 `unordered` 的）：

- 元素总是有序的，默认是升序（因为底层数据结构通常是红黑树）
- 支持 $O(\log n)$ 地插入、删除、查找元素

无序关联容器（`unordered`）：

- 元素是无序的（因为底层数据结构是哈希表）
- 支持平均情况下 $O(1)$ 地插入、删除、查找元素

# 迭代器

# 假如没有迭代器...

不同的容器，根据底层数据结构不同，遍历方式自然也不同：

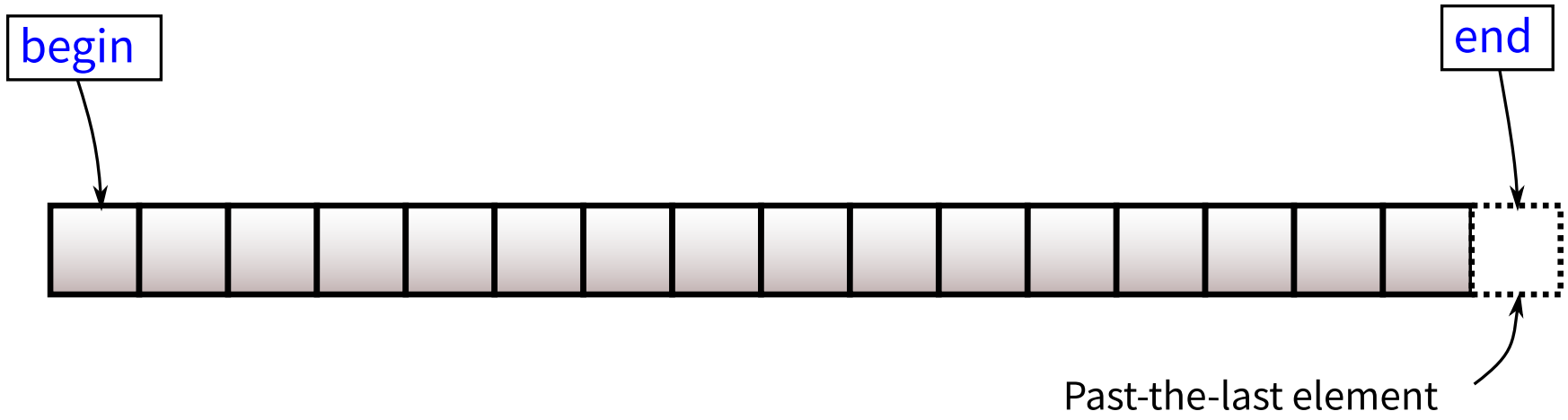
```
for (std::size_t i = 0; i != a.size(); ++i)
  do_something(a[i]);
// 可能的方式：通过指向结点的“句柄”（指针）遍历一个链表
for (node_handle node = l.first_node(); node; node = node.next())
  do_something(node.value())
```

如果是个更复杂的容器呢，比如基于哈希表/红黑树实现的关联容器？

# 使用统一的方式访问元素、遍历容器

所有容器都有其对应的迭代器类型 `Container::iterator`，例如 `std::string::iterator`，`std::vector<int>::iterator`。

所有容器都支持 `c.begin()`，`c.end()`，分别返回指向**首元素**和指向**尾后位置**的迭代器。




* **我们总是使用左闭右开区间 `[begin, end)` 表示一个“迭代器范围”**

# 使用统一的方式访问元素、遍历容器

- `it1 != it1` 比较两个迭代器是否相等（指向相同位置）
- `++it` 让 `it` 指向下一个位置。
- `*it` 获取 `it` 指向的元素的**引用**。

```
for (auto it = c.begin(); it != c.end(); ++it)
  do_something(*it);
```

基于范围的 `for` 语句：完全等价于上面的使用迭代器的遍历

```
for (auto &x : c)
  do_something(x);
```

## `const_iterator`

带有“**底层 `const`**”的迭代器。

- `Container::const_iterator`
- `c.cbegin()`，`c.cend()`
- 在一个 `const` 对象上，`c.begin()` 和 `c.end()` **也返回 `const_iterator`**。

对 `const_iterator` 解引用会得到 `const T &` 而非 `T &`，无法通过它修改元素的值。

## `begin`, `end`, `cbegin`, `cend`

再次出现了 `const` 和 non-`const` 的重载。

```
class MyContainer {
public:
  using iterator = /* unspecified */;
  using const_iterator = /* unspecified */;
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
  const_iterator cbegin() const;
  const_iterator cend() const;
};
```

# 迭代器型别 (iterator category)

- ForwardIterator 前向迭代器: 支持基本操作 `*it`, `it->mem`, `++it`, `it++`, `it1 == it2`, `it1 != it2`
- BidirectionalIterator 双向迭代器: 在 ForwardIterator 的基础上，支持 `--it`, `it--`。
- RandomAccessIterator 随机访问迭代器: 在 BidirectionalIterator 的基础上，支持算术运算和大小比较 `it[n]`, `it + n`, `it += n`, `n + it`, `it - n`, `it -= n`, `it1 - it2`, `<`, `<=`, `>`, `>=`。

`vector`, `string`, `array`, `deque` 的迭代器是 RandomAccessIterator； `list` 的迭代器是 BidirectionalIterator； `forward_list` 的迭代器是 ForwardIterator。

# 迭代器型别 (iterator category)

还有两种迭代器型别：InputIterator 和 OutputIterator。

- InputIterator 表示**可以通过这个迭代器获得元素**（不要求能修改它所指向的元素）
- OutputIterator 表示**可以通过这个迭代器写入元素**（不要求能获得它所指向的元素）
- A ForwardIterator **is an** InputIterator.

稍后我们会看到一些例子。

# 从迭代器范围初始化

`std::string` 以及大多数容器都支持从一个迭代器范围初始化：

```
std::vector<char> v = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i'};
std::vector v2(v.begin() + 2, v.end() - 3);  // {'c', 'd', 'e', 'f'}
std::string s(v.begin(), v.end());           // "abcdefghi"
```

- CTAD 同样能发挥作用：`v2` 的类型是 `std::vector<char>`，这里的元素类型 `char` 是根据我们提供的迭代器推断出来的。

# 基于范围的 `for` 语句

基于范围的 `for` 语句本质上是用**迭代器**遍历：

```
std::vector<int> v = something();
for (auto x : v) {
  do_something_with(x);
}
```

```
std::vector<int> v = something();
{
  auto begin = v.begin();
  auto end = v.end();
  for (; begin != end; ++begin) {
    do_something_with(*begin);
  }
}
```

* 如何让 `Dynarray` 也能用基于范围的 `for` 语句遍历？

# 标准库算法

通常接受一对迭代器表示范围 `[begin, end)`

```
std::sort(a, a + N);                  // a 可能是一个数组
std::copy(v.begin(), v.end(), a); // v 可能是一个 std::vector<...>
```

带 `_n` 后缀的函数使用 `[begin, begin + n)`

```
class Dynarray {
public:
  Dynarray(const Dynarray &other)
      : m_length{other.size()}, m_storage{new int[other.size()]} {
    std::copy_n(other.m_storage, other.size(), m_storage);
  }
  Dynarra(std::size_t n, int x) : m_length{n}, m_storage{new int[n]} {
    std::fill_n(m_storage, m_length, x);
  }
};
```

# 标准库算法

通常用迭代器表示位置

```
auto pos = std::find(v.begin(), v.end(), val); // pos 指向 val 第一次出现的位置
auto maxPos = std::max_element(v.begin(), v.end()); // maxPos 指向最大值所在的位置
```

部分算法对迭代器型别有要求，比如

- `std::sort` 接受 RandomAccessIterator ，比较严格。
- `std::copy` 接受任何 InputIterator 作为前两个参数，要求第三个参数是 OutputIterator 。

部分算法对元素类型有要求，比如

- `std::sort` 要求元素类型具有 `<` 运算符。
- `std::equal` 要求元素类型具有 `==` 运算符。

# 算法不修改容器

标准库算法绝不会改变容器的大小（除非传给它的迭代器是某些特殊的迭代器适配器）

例如，`std::copy(begin, end, dest)` 只是**复制**元素，但并不向容器**插入**元素！

```
std::vector<int> a = someValues();
std::vector<int> b(a.size());
std::vector<int> c{};
std::copy(a.begin(), a.end(), b.begin()); // 正确
std::copy(a.begin(), a.end(), c.begin()); // 未定义行为!
```

某种神秘的迭代器适配器：它的 `++` 和赋值操作十分不同寻常，会将 `*iter++ = x` 变为 `c.push_back(x)`

```
std::copy(a.begin(), a.end(), std::back_inserter(c)); // 这是可以的
```

# 一些常见算法

- 不修改序列：`count`，`find`，`find_end`，`find_first_of`，`search` 等
- 修改序列：`copy`，`fill`，`reverse`，`unique` 等
- 划分、排序、归并：`partition`，`sort`，`nth_element`，`merge` 等
- 二分查找：`lower_bound`，`upper_bound`，`binary_search`，`equal_range`
- 堆相关：`make_heap`，`push_heap`，`pop_heap`，`sort_heap` 等
- 大小比较：`min` / `max`，`min_element` / `max_element`，`equal`，`lexicographical_compare` 等
- 数值运算（`<numeric>`）：`accumulate`，`inner_product` 等

# 谓词 (Predicate)

许多算法接受一个谓词 (predicate)，即一个返回 `bool` 的**可调用对象**，来定制操作。

- 需要比较元素的算法通常默认采用 `<` 进行比较，但也提供接受一个二元谓词 `cmp` 的版本，用 `cmp(a, b)` 代替 `a < b` 。
  - `std::sort(begin, end, cmp)`
  - `std::max_element(begin, end, cmp)`
- 带有 `_if` 后缀的函数接受一个一元谓词 `cond` ，只关心那些 `cond(element)` 为真的元素 `element` 。
  - `std::find_if(b, e, [](int x) { return x % 2 == 0; })` 找第一个偶数
  - `std::copy_if(b, e, d, [](int x) { return x % 2 == 0; })` 只拷贝偶数

# 可调用对象

C++ 中的可调用对象有函数、函数指针、类型为重载了调用运算符 `operator()` 的类类型的对象。

Lambda 本质上是让编译器帮你合成一个重载了 `operator()` 的类型并创建一个该类型的对象。

```
auto cmp =
    [](const std::string &a,
       const std::string &b) {
         return a.size() < b.size();
       };
```

```
struct LambdaType_cmp {
  bool operator()(const std::string &a,
                  const std::string &b) {
    return a.size() < b.size();
  }
};
LambdaType_cmp cmp;
```

# 迭代器的辅助函数

- `std::advance(iter, n)` 将 `iter` 前进 `n` 步。 `n` 也可以是负的。
- `std::distance(iter1, iter2)` 返回 `iter` 和 `iter2` 之间的**距离**：相距几个元素。
- `std::next(iter)` 返回 `iter` 的“下一个位置”， `std::prev(iter)` 返回 `iter` 的“上一个位置”，它们都不会改变 `iter` 本身。
  - 用来代替手动的 `auto tmp = iter; ++tmp;` 。

以上函数对于**各类迭代器型别**都有支持，并且会针对不同的型别提供不同的实现。

- 不管 `iter1` 和 `iter2` 是什么型别，你都可以用 `std::distance(iter1, iter2)` 获取它们的距离，不用担心 `iter1 - iter2` 这个表达式能不能编译之类的问题。

# 迭代器适配器 (iterator adaptors)

# 迭代器适配器 (iterator adaptors)

一种用起来像迭代器的东西

- 可能是根据某个迭代器和/或一些别的东西“变出来”的
- 用起来像迭代器，但多多少少有点区别，比如
  - 反向迭代：`++` 实际上是 `--`
  - 帮助移动元素：`*iter` 返回右值引用
  - 看似迭代，实则插入元素：`*iter++ = x` 会被“变”成 `c.push_back(x)`
  - 将某些特殊的过程也抽象为“迭代”：`x = *iter++` 被“变”成 `std::cin >> x`
  - ……

# 反向迭代器 `reverse_iterator`

每个容器都有自己的反向迭代器：`Container::reverse_iterator` 和 `Container::const_reverse_iterator`，通常被这样定义：

```
using reverse_iterator = std::reverse_iterator<iterator>;
using const_reverse_iterator = std::reverse_iterator<const_iterator>;
```

- `c.rbegin()`，`c.rend()`，`c.crbegin()`，`c.crend()`
- `++` 和 `--` 在反向迭代器上都是反的。

```
std::vector v{1, 2, 3, 4, 5};
for (auto rit = v.rbegin(); rit != v.rend(); ++rit)
  std::cout << *rit << ' ';
```

输出：`5 4 3 2 1`

# 移动迭代器 `move_iterator`

- `std::make_move_iterator(iter)` 从一个普通的迭代器 `iter` 变出一个“移动迭代器”
- `*mit` 会得到右值引用而非左值引用，从而元素更可能被移动而非被拷贝。

```
std::vector<std::string> words = someValues();
std::vector<std::string> v(words.size());
std::copy(std::make_move_iterator(words.begin()),
          std::make_move_iterator(words.end()), v.begin());
```

`words` 中的每个 `string` 被**移动**进了 `v`，而不是拷贝。

# 从迭代器向迭代器的映射

对于一个给定的迭代器类型 `Iter` ， `std::reverse_iterator<Iter>` 和 `std::move_iterator<Iter>` 都是一个新的迭代器类型:

- 它们是 `Iter` 的“wrapper”：在内部保存一个 `Iter` 类型的原始迭代器，可以通过 `rit.base()` 访问。
- 它们基于 `Iter` 的接口定义自己的接口： `++rit` 会调用 `--it` ， `*mit` 会返回 `std::move(*it)`

C++23 还有个 `std::basic_const_iterator<Iter>` ，用类似的方式定义了 `Iter` 的带有底层 `const` 的版本。

你当然可以用类似的方式实现自己的特殊迭代器。

# 插入迭代器

`insert_iterator`, `front_insert_iterator`, `back_insert_iterator`

典型的 OutputIterator：

- 只可向 `*iter` 写入元素，不能从 `*iter` 读取元素
- 它们会调用容器的 `insert`, `push_front` 或 `push_back`，将“写入”的元素插入容器

```
std::vector<int> numbers = someValues();
std::vector<int> v;
std::copy(numbers.begin(), numbers.end(), v.begin()); // 错误!
std::copy(numbers.begin(), numbers.end(), std::back_inserter(v)); // 正确
```

`std::back_inserter(c)` 生成一个 `std::back_insert_iterator<Container>`，它内部保存一个 `Container &r = c`，会不断调用 `r.push_back(x)` 将向它“写入”的元素添加进 `c`。

# 流迭代器

读入一串数存进一个 `vector<int>`：

```
std::vector<int> v(std::istream_iterator<int>(std::cin),
                   std::istream_iterator<int>{});
```

利用 `std::copy` 将 `v` 中的元素输出，并且每个后面跟一个 `", "`：

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

- `istream_iterator` 是一种 InputIterator，它不断从输入流中获取元素
- `ostream_iterator` 是一种 OutputIterator，它不断将向它"写入"的元素写进输出流

# 迭代器型别

InputIterator 和 ForwardIterator 都要求支持 `++it`, `it++`, `*it`, `it->mem`, `==`, `!=`。

这两类迭代器的区别究竟是什么?

# 迭代器型别

InputIterator 和 ForwardIterator 都要求支持 `++it`, `it++`, `*it`, `it->mem`, `==`, `!=`。

这两类迭代器的区别究竟是什么？—— ForwardIterator 提供 **multi-pass guarantee**

```
auto original = iter;        // 对当前的 iter 做个拷贝
auto value = *iter;          // 现在 iter 指向的元素是 value
++iter;
assert(*original == value);
```

对于一个 ForwardIterator 来说，`original` 指向了 `iter` 在递增之前指向的位置，那个位置上的值始终是 `value`。

InputIterator 不这么认为，它只保证能"input"：`++iter` 就意味着我们打算获取"下一个值"了，先前的值也就无法再被获取了。（考虑"输入"的过程）