C++ 迭代器详解

1. 什么是迭代器

迭代器是 C++ 中用于遍历容器（如数组、向量、列表等）元素的对象。它提供了一种统一的方式来访问容器中的元素，而不需要关心容器的具体实现。迭代器可以被视为指向容器中元素的指针，但它们是更高级的抽象。

示例代码

```
#include <iostream>
#include <vector>

int main() {
```

```
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // 使用迭代器遍历 vector
    for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " "; // 解引用迭代器获取元素
    }
    return 0;
}
```

---

2. 迭代器的好处

1. 统一接口

说明：

迭代器为不同类型的容器提供了一种统一的访问接口。这意味着我们可以使用相同的方式来遍历各种容器数据类型，如数组、列表、向量等。这样，我们就不需要为每个不同类型的容器编写不同的遍历代码。

现实生活中的例子：

想象你是一名旅行者，想要参观不同的景点。无论是博物馆、动物园还是公园，你都可以使用相同的方式来获取信息，比如通过导游（迭代器）。导游会带你游览每个景点，讲解它们的特点，而你不需要了解每个景点的具体布局和历史，只需跟随导游即可。

示例代码：

```
#include <iostream>
#include <vector>
#include <list>

int main() {
    std::vector<int> vec = {1, 2, 3};
    std::list<int> lst = {4, 5, 6};

    // 使用统一的迭代器接口遍历不同类型的容器
    std::vector<int>::iterator it_vec = vec.begin();
    while (it_vec != vec.end()) {
        std::cout << *it_vec << " "; // 输出: 1 2 3
        ++it_vec;
    }
    std::cout << std::endl;
```

```
    std::list<int>::iterator it_lst = lst.begin();
    while (it_lst != lst.end()) {
        std::cout << *it_lst << " "; // 输出: 4 5 6
        ++it_lst;
    }
    std::cout << std::endl;

    return 0;
}
```

2. 封装性

说明：

迭代器隐藏了容器的内部结构，用户不需要了解容器是如何存储数据的，只需关注如何通过迭代器访问数据。这种封装使用户的操作变得简洁。

现实生活中的例子：

想象你在一家餐厅用餐，服务员（迭代器）负责将菜单（容器）上的菜品（元素）带到你面前。你只需要告诉服务员你想要什么，而不必关心厨房如何准备这些菜。如果菜单设计得好，服务员会快速找到你想要的菜，无需你了解厨房的具体操作过程。

示例代码：

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> menu = {{"Pizza", 10}, {"Burger", 5}, {"Pasta", 8}};

    // 使用迭代器访问菜单
    std::map<std::string, int>::iterator it = menu.begin();
    while (it != menu.end()) {
        std::cout << it->first << ": $" << it->second << std::endl; // 输出菜品及其
价格
        ++it;
    }

    return 0;
}
```

3. 算法独立性

说明：

迭代器使得算法与容器分离，这意味着可以将相同的算法应用于不同的数据结构。例如，可以对 `std::vector` 使用排序算法，也可以对 `std::list` 使用相同的排序逻辑。

现实生活中的例子：

想象你在一家超市，想要计算不同商品的总价。无论你购买的是水果、蔬菜还是零食，你都可以使用相同的计算方法来得出总价。这种方法的独立性使你能够快速处理不同类型的商品，而不需要为每种商品编写不同的计算逻辑。

示例代码：

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> prices = {10, 20, 15, 30};

    // 使用相同的算法对不同容器进行排序
    std::sort(prices.begin(), prices.end());

    for (std::vector<int>::iterator it = prices.begin(); it != prices.end(); ++it)
{
        std::cout << *it << " "; // 输出: 10 15 20 30
    }
    std::cout << std::endl;

    return 0;
}
```

4. 灵活性

说明：

通过不同类型的迭代器，用户可以实现多种遍历方式，如正向、反向、随机访问等。这种灵活性使得开发者可以根据需要选择适合的迭代器及访问方式。

现实生活中的例子：

想象你在图书馆借书。书架上有很多书，你可以选择从左到右（正向），也可以从右到左（反向）来查找你想要的书。此外，有些书可以从书架的任意位置直接取下（随机访

问），而另一些书可能只能按顺序借阅（顺序访问）。这种灵活性让你能够以不同的方式访问书籍而不受限制。

示例代码：

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // 正向访问
    std::cout << "Forward: ";
    std::vector<int>::iterator it = vec.begin();
    while (it != vec.end()) {
        std::cout << *it << " "; // 输出: 1 2 3 4 5
        ++it;
    }
    std::cout << std::endl;

    // 随机访问
    std::cout << "Random Access (with indexing): ";
    for (size_t i = 0; i < vec.size(); i++) {
        std::cout << vec[i] << " "; // 输出: 1 2 3 4 5
    }
    std::cout << std::endl;

    return 0;
}
```

---

3. 迭代器类位置

在 C++ 标准模板库（STL）中，迭代器被设计为与容器分离的抽象，使得不同容器能够提供自己的迭代器类型。每种容器，如 `std::vector`, `std::list`, `std::set` 等都有其对应的迭代器类和接口，用于适配标准的迭代器操作。这种设计理念使得用户无论使用何种类型的容器，都可以通过统一的方式遍历和操作数据。

特定的容器应该有特定的迭代器

这一核心理念强调了迭代器的设计与容器密切相关。特定的容器由于其内部结构和性能特征，必然需要特定的迭代器来实现最佳访问模式。例如，`std::vector` 提供随机访问

功能，而 `std::list` 提供双向遍历，这要求它们各自的迭代器具备不同的特性和行为。这样的设计确保了代码的灵活性和高效性。

迭代器类的定义与位置

1. **头文件**: 迭代器类通常在相应的容器头文件中定义。例如：
    - `std::vector` 的迭代器在 `<vector>` 头文件中。
    - `std::list` 的迭代器在 `<list>` 头文件中。
    - `std::set` 的迭代器在 `<set>` 头文件中。
2. **迭代器访问**: 每种容器（***除了反向迭代器***）都提供了 `begin()` 和 `end()` 方法，返回对应的迭代器对象。`begin()` 返回指向容器首元素的迭代器，`end()` 返回指向容器尾后一个位置的迭代器。这使得用户能够轻松使用迭代器进行遍历。 反向迭代器本身并不包含 begin() 和 end() 方法。相反，反向迭代器使用特定的函数 rbegin() 和 rend() 来返回容器的反向迭代器。 在 C++ 中，反向迭代器是通过 `rbegin()` 和 `rend()` 方法获得的，这两个方法与正向迭代器的 `begin()` 和 `end()` 有着密切的关系。

> - **正向迭代器**：通过 `begin()` 和 `end()` 获取，用于从容器的头部向尾部迭代。
> - **反向迭代器**：通过 `rbegin()` 和 `rend()` 获取，用于从容器的尾部向头部迭代。
>
> **`rbegin()` 和 `rend()`**
>
> - `rbegin()`:
>   - 返回一个指向容器最后一个元素的反向迭代器。
>   - 可以通过解引用反向迭代器 `*` 来访问该元素。
> - `rend()`:
>   - 返回一个指向容器第一个元素之前位置的反向迭代器。
>   - 这个指针实际上是一个“结束”指针，用于结束迭代。
>
> **`begin()` 和 `end()`**
>
> - `begin()`:
>   - 返回一个指向容器第一个元素的正向迭代器。

- `end()`:
    - 返回一个指向容器最后一个元素后位置的正向迭代器。

反向迭代器与正向迭代器的对应关系

- `rbegin()` 是 `end()` 的对应迭代器：从容器的尾部开始反向迭代。
- `rend()` 是 `begin()` 的对应迭代器：到达容器的头部后，停止迭代。

换言之，`rbegin()` 和 `rend()` 使得迭代器可以从最后一个元素向第一个元素反向遍历，而 `begin()` 和 `end()` 则是从第一个元素向最后一个元素正向遍历。

示例代码

以下示例展示了如何通过容器的迭代器来访问、遍历和修改容器中的数据。

```
#include <iostream>
#include <vector>
#include <list>

int main() {
    // 创建一个 vector 容器并初始化
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // 使用 vector 的迭代器遍历和修改元素
    std::vector<int>::iterator vec_it;
    std::cout << "Original vector: ";
    for (vec_it = vec.begin(); vec_it != vec.end(); ++vec_it) {
        std::cout << *vec_it << " "; // 输出: 1 2 3 4 5
    }
    std::cout << std::endl;

    // 修改元素
    vec_it = vec.begin();
    *vec_it = 10; // 修改第一个元素
    std::cout << "Modified vector: ";
    for (vec_it = vec.begin(); vec_it != vec.end(); ++vec_it) {
        std::cout << *vec_it << " "; // 输出: 10 2 3 4 5
    }
    std::cout << std::endl;

    // 创建一个 list 容器
    std::list<int> lst = {5, 4, 3, 2, 1};

    // 使用 list 的迭代器遍历元素
    std::list<int>::iterator lst_it;
    std::cout << "List elements: ";
    for (lst_it = lst.begin(); lst_it != lst.end(); ++lst_it) {
        std::cout << *lst_it << " "; // 输出: 5 4 3 2 1
```

```
    }
    std::cout << std::endl;

    // 遍历并删除元素
    lst_it = lst.begin();
    while (lst_it != lst.end()) {
        if (*lst_it == 3) {
            lst_it = lst.erase(lst_it); // 删除元素3
        } else {
            ++lst_it; // 只有在没有删除元素时才递增
        }
    }

    // 输出删除后的列表
    std::cout << "List after deletion: ";
    for (lst_it = lst.begin(); lst_it != lst.end(); ++lst_it) {
        std::cout << *lst_it << " "; // 输出: 5 4 2 1
    }
    std::cout << std::endl;

    return 0;
}
```

代码分析

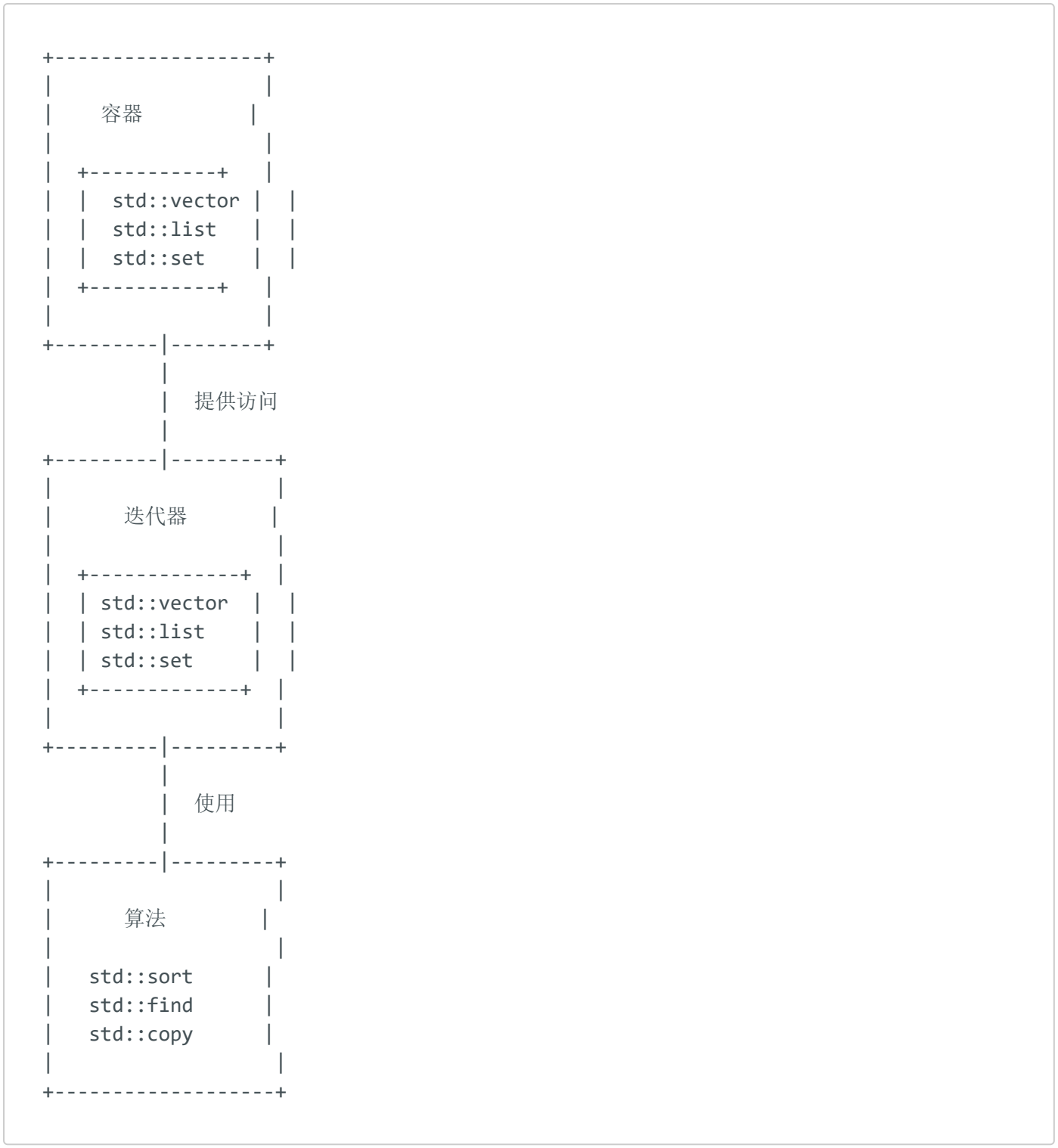
1. **初始化容器及迭代器**:
   - 创建一个 `std::vector` 容器并用整数初始化，同时创建一个 `std::vector<int>::iterator` 迭代器。
   - 遍历容器并打印元素，体现了迭代器的使用方式。
2. **修改元素**:
   - 演示如何使用迭代器直接访问和修改容器中的元素。
3. **操作 list 容器**:
   - 创建一个 `std::list` 容器，使用迭代器遍历并打印元素。
   - 删除元素的示例展示了如何在遍历过程中安全地操作容器。

---

4. 容器、迭代器、算法的关系

在 C++ 的标准模板库（STL）中，容器、迭代器和算法是相互关联的重要组成部分。理解它们之间的关系有助于我们有效地利用 STL 提供的功能。

容器、迭代器和算法的关系示意图



关系描述

1. **容器**:
   - 容器是用于存储数据的结构，如 `std::vector`, `std::list`, `std::set` 等。
   - 它们负责在内存中管理数据，并提供相关的操作接口。
2. **迭代器**:

- 迭代器是访问容器元素的方式，提供了一种统一的接口来遍历和操作存储在容器中的数据。
- 迭代器与容器紧密相连，每种容器都有其特定的迭代器类型，这样可以确保访问方法与容器特性相匹配。

3. **算法**:

- 算法定义了对容器中的元素执行的操作，如排序、查找等。
- 算法不直接了解容器的具体实现，而是通过迭代器来访问元素。这使得同一算法可以适用于不同类型的容器。

示例代码

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    // 创建一个 vector 容器并初始化
    std::vector<int> vec = {5, 3, 4, 1, 2};

    // 使用 std::sort 算法，迭代器提供访问方式
    std::sort(vec.begin(), vec.end());

    // 输出排序后的结果
    for (int num : vec) {
        std::cout << num << " "; // 输出: 1 2 3 4 5
    }
    return 0;
}
```

代码分析

在上述示例中：

- 创建了一个 `std::vector` 容器来保存整数。
- 使用 `std::sort` 算法对容器中的元素进行排序。
- 通过调用 `begin()` 和 `end()` 方法，迭代器提供了遍历和访问容器中数据的接口。

---

5. STL 迭代器的六大类型

STL 提供了六种主要的迭代器类型，每种类型都有其独特的特点和适用场景。以下是对每种迭代器类型的详细分析：

1. 输入迭代器

- **特点**:
  - 只读，单向迭代器。
  - 允许对元素进行解引用（使用 * 运算符），但不允许修改元素。
  - 只能向前移动，使用自增运算符 ++ 进行前进。
- **用途**:
  - 用于从输入源（如流）中读取数据。
- **示例代码**:

```
#include <iostream>
#include <iterator>

int main() {
    std::cout << "Input Iterator: ";
    std::istream_iterator<int> input_it(std::cin), end;
    while (input_it != end) {
        std::cout << *input_it++ << " "; // 读取并输出输入值
    }
    std::cout << std::endl;
    return 0;
}
```

2. 输出迭代器

- **特点**:
  - 只写，单向迭代器。
  - 允许输出元素并插入值，但不允许读取元素。
  - 只能向前移动，使用自增运算符 ++。
- **用途**:
  - 用于将数据写入输出源（如流）。

- **示例代码**:

```
#include <iostream>
#include <vector>
#include <iterator>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::cout << "Output Iterator: ";
    std::ostream_iterator<int> output_it(std::cout, " ");
    std::copy(vec.begin(), vec.end(), output_it); // 将 vector 中的值输出到控制台
    std::cout << std::endl;
    return 0;
}
```

3. 前向迭代器

- **特点**:
    - 允许多次读取和写入元素，但只能向前移动。
    - 支持解引用和自增操作。
- **用途**:
    - 适用于需要频繁读取和写入的容器，如 `std::forward_list`。
- **示例代码**:

```
#include <iostream>
#include <forward_list>

int main() {
    std::forward_list<int> fl = {1, 2, 3, 4, 5};
    std::cout << "Forward Iterator: ";
    for (auto it = fl.begin(); it != fl.end(); ++it) {
        std::cout << *it << " "; // 输出元素
    }
    std::cout << std::endl;
    return 0;
}
```

4. 双向迭代器

- **特点**:
  - 支持双向移动（前后均可）。
  - 允许多次读取和写入。
- **用途**:
  - 适用于 `std::list` 和 `std::deque` 等容器。
- **示例代码**:

```
#include <iostream>
#include <list>

int main() {
    std::list<int> lst = {1, 2, 3, 4, 5};
    std::cout << "Bidirectional Iterator: ";
    for (auto it = lst.begin(); it != lst.end(); ++it) {
        std::cout << *it << " "; // 输出元素
    }
    std::cout << std::endl;
    return 0;
}
```

5. 随机访问迭代器

- **特点**:
  - 允许任意位置访问。
  - 支持算术运算（如加减整数）。
  - 既可以读也可以写。
- **用途**:
  - 适用于 `std::vector` 和 `std::deque` 等容器，提供快速随机访问。
- **示例代码**:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> rand_vec = {10, 20, 30, 40, 50};
    std::cout << "Random Access Iterator: ";
    for (auto it = rand_vec.begin(); it != rand_vec.end(); ++it) {
```

```
        std::cout << *it << " "; // 输出元素
    }
    std::cout << std::endl;
    return 0;
}
```

6. 顺序迭代器

- **特点**:
  - 只支持顺序访问，是其他迭代器的一个特殊存在。
  - 对于需要按顺序访问的容器。
- **用途**:
  - 适用于 `std::deque` 等容器，提供线性访问。
- **示例代码**:

```
#include <iostream>
#include <deque>

int main() {
    std::deque<int> deq = {1, 2, 3, 4, 5};
    std::cout << "Sequential Iterator: ";
    for (auto it = deq.begin(); it != deq.end(); ++it) {
        std::cout << *it << " "; // 输出元素
    }
    std::cout << std::endl;
    return 0;
}
```

另一种分类

在 C++ 中，迭代器的不同类型可以分为访问权限和迭代方向两类。前述的输入迭代器、输出迭代器、前向迭代器、双向迭代器、随机访问迭代器和顺序迭代器主要是按功能和移动性分类，而以下四种迭代器则是基于元素访问权限和方向来进行细化。下面对这四种迭代器进行详细的讲解，以及与之前六种迭代器的区别和联系。

1. 正向迭代器（Forward Iterator）

- **定义及特点**:
  - 正向迭代器允许对元素进行多次读取和写入，但只能向前移动。
  - 支持解引用操作（*）和自增运算（++）。
  - 主要用于顺序访问某些容器的元素。
- **与六大类型的关系**:
  - 正向迭代器是前向迭代器的一个实现形式，具有相同的特性，一般适用于 STL 的大多数容器。
- **示例代码**:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::vector<int>::iterator it; // 正向迭代器
    for (it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " "; // 输出元素
    }
    std::cout << std::endl;
    return 0;
}
```

2. 常量正向迭代器（Const Forward Iterator）

- **定义及特点**:
  - 与正向迭代器类似，但不允许修改元素的值。
  - 通过 `容器类名::const_iterator` 定义。
- **与六大类型的关系**:
  - 它是正向迭代器的常量版本，适用于只读访问。
- **示例代码**:

```
#include <iostream>
#include <vector>

int main() {
    const std::vector<int> vec = {1, 2, 3, 4, 5};
```

```
    std::vector<int>::const_iterator it; // 常量正向迭代器
    for (it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " "; // 输出元素
    }
    std::cout << std::endl;
    return 0;
}
```

3. 反向迭代器（Reverse Iterator）

- **定义及特点**:
  - 允许从容器的末尾向前访问元素。
  - 通过 `容器类名::reverse_iterator` 定义，提供从后向前迭代元素的能力。
- **与六大类型的关系**:
  - 反向迭代器与双向迭代器有关，因为它是双向迭代器的一个特殊情况。
- **示例代码**:

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::vector<int>::reverse_iterator rit; // 反向迭代器
    for (rit = vec.rbegin(); rit != vec.rend(); ++rit) {
        std::cout << *rit << " "; // 输出元素
    }
    std::cout << std::endl;
    return 0;
}
```

4. 常量反向迭代器（Const Reverse Iterator）

- **定义及特点**:
  - 与反向迭代器类似，但不允许修改元素的值。
  - 通过 `容器类名::const_reverse_iterator` 定义。
- **与六大类型的关系**:

- 是反向迭代器的常量版本，适合只读访问。

- **示例代码**:

```
#include <iostream>
#include <vector>

int main() {
    const std::vector<int> vec = {1, 2, 3, 4, 5};
    std::vector<int>::const_reverse_iterator crit; // 常量反向迭代器
    for (crit = vec.crbegin(); crit != vec.crend(); ++crit) {
        std::cout << *crit << " "; // 输出元素
    }
    std::cout << std::endl;
    return 0;
}
```

---

6. 所有迭代器的共同特点

> - **解引用**: 允许访问指向的元素。
> - **自增和自减（++、--）**：能够以简单的语法在容器中遍历元素。
> - **相等性比较（==、!=）**：用于判断两个迭代器是否指向相同元素。
> - **构造、缺省构造和拷贝构造**: 支持不同的构造方式。
> - **赋值（=）**：能够安全地复制迭代器的状态。

这些特点和操作使得迭代器在 C++ 标准库（STL）中成为一个强大而灵活的工具。

1. 解引用

特点: 迭代器可以解引用以访问其指向的元素，类似于指针。

示例代码

```
#include <iostream>
#include <vector>

int main() {
    // 创建一个整数向量
    std::vector<int> vec = {10, 20, 30, 40, 50};

    // 创建一个迭代器，指向向量的开始
```

```
    std::vector<int>::iterator it = vec.begin();

    // 解引用迭代器，获取指向的元素的值
    std::cout << "The first element is: " << *it << std::endl; // 输出: The first
element is: 10

    // 修改迭代器所指向的元素
    *it = 100; // 将第一个元素修改为 100

    // 输出修改后的值
    std::cout << "After modification, the first element is: " << *it << std::endl;
// 输出: After modification, the first element is: 100

    // 输出整个向量以查看更改
    std::cout << "The vector now contains: ";
    for (it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " "; // 输出: 100 20 30 40 50
    }
    std::cout << std::endl;

    return 0;
}
```

2. 自增和自减

特点: 迭代器支持使用 `++` 和 `--` 操作符实现前进和后退操作。

示例代码

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {10, 20, 30, 40, 50};
    std::vector<int>::iterator it = vec.begin(); // 创建迭代器

    // 使用自增遍历容器
    std::cout << "Iterating through the vector: ";
    for (; it != vec.end(); ++it) {
        std::cout << *it << " "; // 输出: 10 20 30 40 50
    }
    std::cout << std::endl;

    return 0;
}
```

3. 相等性比较

特点: 迭代器支持 == 和 != 操作符，用于判断迭代器是否指向同一个位置，通常在遍历时用来检查是否到达容器末尾。

示例代码

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {10, 20, 30, 40, 50};
    std::vector<int>::iterator it = vec.begin(); // 创建迭代器

    std::cout << "Elements in the vector: ";
    while (it != vec.end()) {
        std::cout << *it << " "; // 输出: 10 20 30 40 50
        ++it; // 移动到下一个元素
    }
    std::cout << std::endl;

    // 相等性比较示例
    std::cout << "Is the iterator equal to end? " << (it == vec.end() ? "Yes" :
"No") << std::endl; // 输出: Yes

    return 0;
}
```

4. 构造、缺省构造和拷贝构造

特点: 迭代器可以通过各种构造函数来初始化，包括缺省构造函数和拷贝构造函数。

示例代码

```
#include <iostream>

class CustomIterator {
public:
    int* ptr; // 指向整数类型的指针

    // 缺省构造函数
    CustomIterator() {
        ptr = nullptr; // 在构造函数主体中进行赋值
    }

    // 带参数的构造函数
    CustomIterator(int* p) {
        ptr = p; // 在构造函数主体中进行赋值
    }
```

```
    // 拷贝构造函数
    CustomIterator(const CustomIterator& other) {
        ptr = other.ptr; // 在构造函数主体中进行赋值
    }

    // 赋值操作符
    CustomIterator& operator=(const CustomIterator& other) {
        if (this != &other) { // 防止自赋值
            ptr = other.ptr; // 在赋值操作符中进行赋值
        }
        return *this;
    }

    // 解引用操作符
    int& operator*() { return *ptr; }

    // 自增操作符
    CustomIterator& operator++() {
        ++ptr; // 自增指针
        return *this;
    }

    // 不等于操作符
    bool operator!=(const CustomIterator& other) const {
        return ptr != other.ptr; // 检查指针是否不同
    }
};

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    CustomIterator begin(arr);
    CustomIterator end(arr + 5);

    // 使用自定义迭代器
    std::cout << "Custom Iterator: ";
    for (CustomIterator it = begin; it != end; ++it) {
        std::cout << *it << " "; // 输出: 10 20 30 40 50
    }
    std::cout << std::endl;

    return 0;
}
```

5. 赋值（=）

特点: 自定义赋值操作符允许将一个迭代器的状态赋值给另一个迭代器。

示例代码

```cpp
#include <iostream>

class CustomIterator {
public:
    int* ptr; // 指向整数类型的指针

    // 缺省构造函数
    CustomIterator() {
        ptr = nullptr; // 在构造函数主体中进行赋值
    }

    // 带参数的构造函数
    CustomIterator(int* p) {
        ptr = p; // 在构造函数主体中进行赋值
    }

    // 拷贝构造函数
    CustomIterator(const CustomIterator& other) {
        ptr = other.ptr; // 在构造函数主体中进行赋值
    }

    // 赋值操作符
    CustomIterator& operator=(const CustomIterator& other) {
        if (this != &other) { // 防止自赋值
            ptr = other.ptr; // 在赋值操作符中进行赋值
        }
        return *this;
    }

    // 解引用操作符
    int& operator*() { return *ptr; }

    // 自增操作符
    CustomIterator& operator++() {
        ++ptr; // 自增指针
        return *this;
    }

    // 不等于操作符
    bool operator!=(const CustomIterator& other) const {
        return ptr != other.ptr; // 检查指针是否不同
    }
};

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    CustomIterator begin(arr);
    CustomIterator end(arr + 5);

    // 测试赋值操作符
    CustomIterator it1(arr);
    CustomIterator it2;

    it2 = it1; // 使用赋值操作符

    std::cout << "After assignment, it2 points to: " << *it2 << std::endl; // 输出:
```

```
After assignment, it2 points to: 10

    return 0;
}
```

---

7. 总结

迭代器类型	特点	示例代码
输入迭代器	只读，单向，支持解引用和自增操作	`std::istream_iterator<int> it(std::cin);`
输出迭代器	只写，单向，支持解引用和自增操作	`std::ostream_iterator<int> out(std::cout, " ");`
前向迭代器	读写，单向，支持多次读取和写入	`std::forward_list<int>::iterator it;`
双向迭代器	读写，双向，支持前后移动	`std::list<int>::iterator it;`
随机访问迭代器	读写，任意访问，支持算术运算	`std::vector<int>::iterator it;`
顺序迭代器	只支持顺序访问	`std::deque<int>::iterator it;`

示例代码

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <iterator>

int main() {
    // 输入迭代器
    std::cout << "Input Iterator: ";
    std::istream_iterator<int> input_it(std::cin), end;
    while (input_it != end) {
        std::cout << *input_it++ << " ";
    }
    std::cout << std::endl;
```

```
    // 输出迭代器
    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::cout << "Output Iterator: ";
    std::ostream_iterator<int> output_it(std::cout, " ");
    std::copy(vec.begin(), vec.end(), output_it);
    std::cout << std::endl;

    // 前向迭代器
    std::forward_list<int> fl = {1, 2, 3};
    std::cout << "Forward Iterator: ";
    for (auto it = fl.begin(); it != fl.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 双向迭代器
    std::list<int> lst = {1, 2, 3};
    std::cout << "Bidirectional Iterator: ";
    for (auto it = lst.begin(); it != lst.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    // 随机访问迭代器
    std::vector<int> rand_vec = {10, 20, 30, 40};
    std::cout << "Random Access Iterator: ";
    for (auto it = rand_vec.begin(); it != rand_vec.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}
```