深入通用容器：C++ STL的奋斗与实践

引言

在现代编程中，容器作为数据管理的重要工具，为开发者提供了灵活和高效的数据存储和操作方式。C++标准模板库（STL）中的通用容器，以其高效的算法和灵活的适应性，成为了程序设计时不可或缺的一部分。

---

1. 概述

通用容器是用于存储数据的集合，主要分为线性容器和关联容器。线性容器如 `vector`、`deque`和 `list`，可以直接按顺序存储元素。而关联容器如 `set`、`map`，则根据键值对有效存储和查找数据。

1.1 容器的分类

通用容器可以分为以下几类：

- **顺序容器**：如 `vector`、`deque`和 `list`，用于按顺序存储元素。
- **关联容器**：如 `set`和 `map`，通过键值对进行高效的查找与存储。
- **适配器容器**：如 `stack`和 `queue`，对已有容器进行某种特性的封装。

1.2 容器的共性

无论是哪类容器，其一般特性包括：

- 默认构造函数：用于初始化容器。
- 复制构造函数：创建类似于现有容器的副本。
- 空状态检查：可以检测容器是否为空。
- 大小属性：可以返回容器当前的元素个数。

1.3 不同容器的比较

在C++标准模板库（STL）中，容器提供了多种数据存储和操作方式。每种容器都有其独特的特性和适用场景。以下是对 `vector`、`deque`、`list`、`set`、`map`、`multiset`和`multimap`的详细比较。

1. `vector`

- **特性**：
  - 动态数组，支持快速随机访问。
  - 元素存储在连续的内存空间中。
  - 支持尾部插入和删除操作，效率高。
  - 中间插入和删除操作较慢，因为需要移动元素。
- **适用场景**：
  - 适合需要频繁随机访问的场景，如查找和修改元素。

2. `deque`

- **特性**：
  - 双端队列，支持在头部和尾部快速插入和删除。
  - 内部实现为多个小块的连续内存，适合频繁的头尾操作。
  - 随机访问速度略低于 `vector`。
- **适用场景**：
  - 适合需要在两端频繁插入和删除的场景，如实现队列。

3. `list`

- **特性**：
  - 双向链表，插入和删除操作不会导致迭代器失效。
  - 每个元素之间通过指针连接，内存不连续。
  - 不支持随机访问，访问元素时需要遍历。
- **适用场景**：
  - 适合需要频繁插入和删除的场景，尤其是在中间位置。

4. `set`

- **特性**：
  - 存储唯一元素，自动排序。
  - 基于红黑树实现，查找、插入和删除操作的时间复杂度为O(log n)。
  - 不允许重复元素。
- **适用场景**：

- 适合需要保持元素唯一性和有序性的场景，如集合操作。

5. `map`

- **特性**：
  - 存储键值对，键唯一，值可以重复。
  - 基于红黑树实现，提供O(log n)的查找、插入和删除操作。
  - 自动根据键排序。
- **适用场景**：
  - 适合需要快速查找和存储关联数据的场景，如字典。

6. `multiset`

- **特性**：
  - 存储元素的集合，允许重复元素。
  - 元素自动排序，基于红黑树实现，插入、查找和删除操作的时间复杂度为O(log n)。
- **适用场景**：
  - 适合需要存储重复元素并保持有序的场景，如统计数据。

7. `multimap`

- **特性**：
  - 存储键值对，允许同一个键对应多个值。
  - 基于红黑树实现，提供O(log n)的查找、插入和删除操作。
  - 自动根据键排序。
- **适用场景**：
  - 适合需要存储重复键值对的场景，如词频统计。

容器特性汇总表

容器类型	存储方式	随机访问	插入效率	删除效率	元素唯一性	适用场景
`vector`	连续内存	快速	尾部高	尾部高	允许重复	频繁随机访问
`deque`	分散内存	较快	头尾高	头尾高	允许重复	头尾频繁操作

容器类型	存储方式	随机访问	插入效率	删除效率	元素唯一性	适用场景
`list`	链表	不支持	常数时间	常数时间	允许重复	中间频繁插入删除
`set`	红黑树	不支持	O(log n)	O(log n)	唯一	唯一性和排序
`map`	红黑树	不支持	O(log n)	O(log n)	唯一	键值对存储
`multiset`	红黑树	不支持	O(log n)	O(log n)	允许重复	存储重复元素并保持有序
`multimap`	红黑树	不支持	O(log n)	O(log n)	允许重复	存储重复键值对

2. 关于vector容器

2.1 概述

- **特性**:
  - 动态数组，支持快速随机访问。
  - 元素存储在连续的内存空间中。
  - 支持尾部插入和删除操作，效率高。
  - 中间插入和删除操作较慢，因为需要移动元素。
- **适用场景**:
  - 适合需要频繁随机访问的场景，如查找和修改元素。

2.2 vector的常用函数

构造函数v

1. `vector()`

功能: 创建一个空的 `vector`。

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

int main() {
    std::vector<int> vec; // 创建一个空的 vector
    std::cout << "Size: " << vec.size() << ", Capacity: " << vec.capacity() <<
std::endl;
    return 0;
}
```

输出:

```
Size: 0, Capacity: 0
```

2. `vector(int nSize)`

功能: 创建一个指定大小的 `vector`。

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

int main() {
    std::vector<int> vec(5); // 创建一个包含 5 个默认初始化的元素的 vector
    std::cout << "Size: " << vec.size() << ", Capacity: " << vec.capacity() <<
std::endl;
    return 0;
}
```

输出:

```
Size: 5, Capacity: 5
```

3. `vector(int nSize, const T& t)`

功能: 创建一个指定大小且每个元素都初始化为给定值的 `vector`。

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

int main() {
    std::vector<int> vec(5, 42); // 创建一个包含 5 个初始值为 42 的元素的 vector
    for (int num : vec) {
```

```
        std::cout << num << " ";
    }
    return 0;
}
```

输出:

```
42 42 42 42 42
```

4. `vector(const vector& v)`

功能: 通过现有 `vector` 的副本创建一个新的 `vector`。

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

int main() {
    std::vector<int> original(3, 5); // 创建一个包含 3 个初始值为 5 的元素的 vector
    std::vector<int> copy = original; // 复制构造函数

    std::cout << "Copy Size: " << copy.size() << ", Capacity: " << copy.capacity()
<< std::endl;
    return 0;
}
```

输出:

```
Copy Size: 3, Capacity: 3
```

增加元素v

1. `void push_back(const T& x)`

功能: 在 `vector` 的末尾添加一个元素。

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

int main() {
    std::vector<int> vec;
    vec.push_back(10);
```

```
    vec.push_back(20);

    for (int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

输出:

```
10 20
```

2. `iterator insert(iterator it, const T& x)`

功能: 在指定位置插入一个元素。

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

int main() {
    std::vector<int> vec{1, 2, 4};
    vec.insert(vec.begin() + 2, 3); // 在位置 2 插入元素 3

    for (int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

输出:

```
1 2 3 4
```

3. `iterator insert(iterator it, int n, const T& x)`

功能: 在指定位置插入多个相同的元素。

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

int main() {
    std::vector<int> vec{1, 2};
    vec.insert(vec.begin(), 2, 0); // 在位置 0 插入 2 个 0
```

```
    for (int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

输出:

```
0 0 1 2
```

4. `iterator insert(iterator first, iterator last)`

功能: 在指定位置插入一段元素。

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

int main() {
    std::vector<int> vec{1, 2, 5};
    std::vector<int> toInsert{3, 4};
    vec.insert(vec.begin() + 2, toInsert.begin(), toInsert.end()); // 在位置 2 插入
3 和 4

    for (int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

输出:

```
1 2 3 4 5
```

删除元素v

1. `iterator erase(iterator it)`

功能: 删除指定位置的元素。

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

int main() {
    std::vector<int> vec{1, 2, 3, 4};
    vec.erase(vec.begin() + 2); // 删除位置 2 的元素

    for (int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

输出:

```
1 2 4
```

2. `iterator erase(iterator first, iterator last)`

功能: 删除指定范围内的元素。

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

int main() {
    std::vector<int> vec{1, 2, 3, 4, 5};
    vec.erase(vec.begin() + 1, vec.begin() + 4); // 删除位置 1 到 3 的元素

    for (int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

输出:

```
1 5
```

3. `void pop_back()`

功能: 删除 `vector` 最末尾的元素。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    vec.pop_back(); // 删除最后一个元素

    for (int num : vec) {
        std::cout << num << " ";
    }
    return 0;
}
```

输出:

```
1 2
```

4. `void clear()`

功能: 清除所有元素。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    vec.clear(); // 清除所有元素

    std::cout << "Size: " << vec.size() << std::endl;
    return 0;
}
```

输出:

```
Size: 0
```

遍历函数v

1. `reference at(int pos)`
    - **功能**: 返回 `pos` 位置元素的引用。

- **描述**: 如果 `pos` 超出范围，会抛出一个 `std::out_of_range` 异常。可用于安全访问元素。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    try {
        std::cout << "Element at position 1: " << vec.at(1) << std::endl; // 输出: 2
        std::cout << "Element at position 5: " << vec.at(5) << std::endl; // 超出范围
    } catch (const std::out_of_range& e) {
        std::cout << "Error: " << e.what() << std::endl; // 输出错误信息
    }
    return 0;
}
```

输出:

```
Element at position 1: 2
Error: vector::at: index out of range
```

2. `reference front()`

- **功能**: 返回首元素的引用。
- **描述**: 返回 `vector` 第一个元素的引用。如果 `vector` 为空，会导致未定义行为。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    std::cout << "First element: " << vec.front() << std::endl; // 输出: 1
    return 0;
}
```

输出:

```
First element: 1
```

3. `reference back()`

   - **功能**: 返回尾元素的引用。
   - **描述**: 返回 `vector` 最后一个元素的引用。如果 `vector` 为空，会导致未定义行为。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    std::cout << "Last element: " << vec.back() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Last element: 3
```

4. `iterator begin()`

   - **功能**: 返回一个指向第一个元素的迭代器。
   - **描述**: 返回一个指向当前 `vector` 中第一个元素的迭代器。如果 `vector` 为空，将等于 `end()`。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    std::cout << "First element using begin(): " << *vec.begin() << std::endl;
// 输出: 1
    return 0;
}
```

输出:

```
First element using begin(): 1
```

5. `iterator end()`

- **功能**: 返回一个指向最后一个元素后一个位置的迭代器。
- **描述**: 这个迭代器代表了 `vector` 的结束位置，不指向有效元素。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    std::cout << "End position: " << *(vec.end() - 1) << " (last element)" <<
std::endl; // 输出: 3
    return 0;
}
```

输出:

```
End position: 3 (last element)
```

6. `reverse_iterator rbegin()`

- **功能**: 返回指向最后一个元素的反向迭代器。
- **描述**: 用于逆向遍历 `vector`，指向 `end()` 的前一个元素。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    std::cout << "Last element using rbegin(): " << *vec.rbegin() <<
std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Last element using rbegin(): 3
```

7. `reverse_iterator rend()`

- **功能**: 返回指向前一个位置的反向迭代器。
- **描述**: 这个迭代器在反向遍历时表示 `vector` 的开始位置。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    std::cout << "First element using rend(): " << *(vec.rend() - 1) <<
std::endl; // 输出: 1
    return 0;
}
```

输出:

```
First element using rend(): 1
```

判断函数v

1. `bool empty() const`
   - **功能**: 判断 `vector` 是否为空。
   - **描述**: 如果 `vector` 中没有元素，返回 `true`；否则返回 `false`。

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

int main() {
    std::vector<int> vec;
    std::cout << "Is vector empty? " << (vec.empty() ? "Yes" : "No") <<
std::endl; // 输出: Yes
    vec.push_back(1);
    std::cout << "Is vector empty? " << (vec.empty() ? "Yes" : "No") <<
std::endl; // 输出: No
    return 0;
}
```

输出:

```
Is vector empty? Yes
Is vector empty? No
```

大小函数v

1. `int size() const`

   - **功能**: 返回当前元素的个数。
   - **描述**: 返回 `vector` 中实际存储的元素数量。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    std::cout << "Size of vector: " << vec.size() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Size of vector: 3
```

2. `int capacity() const`

   - **功能**: 返回当前内存分配的容量。
   - **描述**: 返回 `vector` 能够存储的最大元素数量（已分配的空间）。

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

int main() {
    std::vector<int> vec;
    vec.reserve(10); // 预留容量
    std::cout << "Capacity of vector: " << vec.capacity() << std::endl; // 输
出: 10
    return 0;
}
```

输出:

```
Capacity of vector: 10
```

3. `int max_size() const`

   - **功能**: 返回最大可能的元素数量。

- **描述**: 返回当前 `vector` 所能容纳的最大元素数量。

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

int main() {
    std::vector<int> vec;
    std::cout << "Max size of vector: " << vec.max_size() << std::endl; // 根据具体实现输出
    return 0;
}
```

输出:

```
Max size of vector: 4611686018427387903 (具体值可能因平台而异)
```

其他函数v

1. `void swap(vector& v)`

   - **功能**: 交换两个 `vector` 的内容。
   - **描述**: 交换当前 `vector` 和参数 `v` 的内容、大小和容量。

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

int main() {
    std::vector<int> vec1{1, 2, 3};
    std::vector<int> vec2{4, 5, 6};
    vec1.swap(vec2); // 交换内容

    std::cout << "vec1: ";
    for (int num : vec1) {
        std::cout << num << " "; // 输出: 4 5 6
    }

    std::cout << "\nvec2: ";
    for (int num : vec2) {
        std::cout << num << " "; // 输出: 1 2 3
    }

    return 0;
}
```

输出:

```
vec1: 4 5 6
vec2: 1 2 3
```

2. `void assign(int n, const T& x)`

   - **功能**: 将 `x` 赋值给 `vector` 中的前 `n` 个元素。
   - **描述**: 将 `vector` 的大小调整为 `n`，并将所有元素赋值为 `x`。

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

int main() {
    std::vector<int> vec;
    vec.assign(5, 10); // 向量大小调整为 5，赋值为 10

    for (int num : vec) {
        std::cout << num << " "; // 输出: 10 10 10 10 10
    }

    return 0;
}
```

输出:

```
10 10 10 10 10
```

3. `void assign(const_iterator first, const_iterator last)`

   - **功能**: 将 `[first, last)` 中的元素赋值给 `vector`。
   - **描述**: 将当前 `vector` 的内容替换为给定范围内的元素。

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

int main() {
    std::vector<int> vec{1, 2, 3, 4, 5};
    std::vector<int> newVec{9, 8, 7};

    vec.assign(newVec.begin(), newVec.end()); // 将 newVec 中的元素赋值给 vec

    for (int num : vec) {
        std::cout << num << " "; // 输出: 9 8 7
```

```
    }

    return 0;
}
```

输出:

```
9 8 7
```

---

3. 关于deque容器

3.1 概述

- **特性**:
  - 双端队列，支持在头部和尾部快速插入和删除。
  - 内部实现为多个小块的连续内存，适合频繁的头尾操作。
  - 随机访问速度略低于 `vector`。
- **适用场景**:
  - 适合需要在两端频繁插入和删除的场景，如实现队列。

以下是 C++ `deque` 的构造函数、增加功能、删除功能、遍历函数、判断函数、大小函数和其他函数的详细描述，包括代码示例和运行结果。

3.2 deque的常见函数

构造函数d

1. `deque()`
   - **功能**: 创建一个空的 `deque`。
   - **描述**: 默认构造函数，初始化一个没有元素的 `deque`。

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

int main() {
```

```
    std::deque<int> d; // 创建空的 deque
    std::cout << "Size: " << d.size() << std::endl; // 输出: 0
    return 0;
}
```

输出:

```
Size: 0
```

2. `deque(int nSize)`

   - **功能**: 创建一个指定大小的 `deque`。
   - **描述**: `deque` 会包含 `nSize` 个默认初始化的元素。

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

int main() {
    std::deque<int> d(5); // 创建一个包含 5 个默认初始化的元素的 deque
    std::cout << "Size: " << d.size() << std::endl; // 输出: 5
    return 0;
}
```

输出:

```
Size: 5
```

3. `deque(int nSize, const T& t)`

   - **功能**: 创建一个指定大小且每个元素都初始化为给定值的 `deque`。
   - **描述**: `deque` 会被初始化为包含 `nSize` 个元素，所有元素的值都为 `t`。

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

int main() {
    std::deque<int> d(5, 42); // 创建一个包含 5 个初始值为 42 的元素的 deque
    for (int num : d) {
        std::cout << num << " "; // 输出: 42 42 42 42 42
    }
    std::cout << std::endl;
```

```
    return 0;
}
```

输出:

```
42 42 42 42 42
```

4. `deque(const deque& d)`

   - **功能**: 通过现有 `deque` 的副本创建一个新的 `deque`。
   - **描述**: 这个复制构造函数会复制 `d` 的所有元素。

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

int main() {
    std::deque<int> original{1, 2, 3};
    std::deque<int> copy = original; // 复制构造函数

    std::cout << "Copy Size: " << copy.size() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Copy Size: 3
```

增加函数d

1. `void push_front(const T& x)`

   - **功能**: 在 `deque` 的前面添加一个元素。
   - **描述**: 将 `x` 复制到 `deque` 的前面。

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

int main() {
    std::deque<int> d;
    d.push_front(10);
    d.push_front(20);
```

```
    for (int num : d) {
        std::cout << num << " "; // 输出: 20 10
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
20 10
```

2. `void push_back(const T& x)`

   - **功能**: 在 `deque` 的后面添加一个元素。
   - **描述**: 将 `x` 复制到 `deque` 的后面。

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

int main() {
    std::deque<int> d;
    d.push_back(10);
    d.push_back(20);

    for (int num : d) {
        std::cout << num << " "; // 输出: 10 20
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
10 20
```

3. `iterator insert(iterator it, const T& x)`

   - **功能**: 在指定位置插入一个元素。
   - **描述**: 在 `it` 指向的位置前插入 `x`。

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

```cpp
int main() {
    std::deque<int> d{1, 2, 4};
    d.insert(d.begin() + 2, 3); // 在位置 2 插入元素 3

    for (int num : d) {
        std::cout << num << " "; // 输出: 1 2 3 4
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 2 3 4
```

4. `iterator insert(iterator it, int n, const T& x)`

   - **功能**: 在指定位置插入多个相同的元素。
   - **描述**: 在 `it` 指向的位置插入 `n` 个值为 `x` 的元素。

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

int main() {
    std::deque<int> d{1, 2};
    d.insert(d.begin(), 2, 0); // 在位置 0 插入 2 个 0

    for (int num : d) {
        std::cout << num << " "; // 输出: 0 0 1 2
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
0 0 1 2
```

5. `iterator insert(iterator it, const_iterator first, const_iterator last)`

   - **功能**: 在指定位置插入一段元素。
   - **描述**: 在 `it` 指向的位置插入从 `first` 到 `last` 的元素。

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

int main() {
    std::deque<int> d{1, 2, 5};
    std::deque<int> toInsert{3, 4};
    d.insert(d.begin() + 2, toInsert.begin(), toInsert.end()); // 在位置 2 插入 3 和 4

    for (int num : d) {
        std::cout << num << " "; // 输出: 1 2 3 4 5
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 2 3 4 5
```

删除函数d

1. `iterator erase(iterator it)`
   - **功能**: 删除指定位置的元素。
   - **描述**: 删除 `it` 指向的元素。

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

int main() {
    std::deque<int> d{1, 2, 3, 4};
    d.erase(d.begin() + 2); // 删除位置 2 的元素

    for (int num : d) {
        std::cout << num << " "; // 输出: 1 2 4
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 2 4
```

2. `iterator erase(iterator first, iterator last)`

   - **功能**: 删除指定范围内的元素。
   - **描述**: 删除从 `first` 到 `last` 之间的所有元素。

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

int main() {
    std::deque<int> d{1, 2, 3, 4, 5};
    d.erase(d.begin() + 1, d.begin() + 4); // 删除位置 1 到 3 的元素

    for (int num : d) {
        std::cout << num << " "; // 输出: 1 5
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 5
```

3. `void pop_front()`

   - **功能**: 删除 `deque` 最前面的元素。
   - **描述**: 此操作将删除当前 `deque` 的第一元素。

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

int main() {
    std::deque<int> d{1, 2, 3};
    d.pop_front(); // 删除第一个元素

    for (int num : d) {
        std::cout << num << " "; // 输出: 2 3
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
2 3
```

4. `void pop_back()`

   - **功能**: 删除 `deque` 最后面的元素。
   - **描述**: 此操作将删除当前 `deque` 的最后一个元素。

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

int main() {
    std::deque<int> d{1, 2, 3};
    d.pop_back(); // 删除最后一个元素

    for (int num : d) {
        std::cout << num << " "; // 输出: 1 2
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 2
```

5. `void clear()`

   - **功能**: 清除所有元素。
   - **描述**: 此操作会移除 `deque` 中的所有元素。

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

int main() {
    std::deque<int> d{1, 2, 3};
    d.clear(); // 清除所有元素

    std::cout << "Size after clear: " << d.size() << std::endl; // 输出: 0
    return 0;
}
```

输出:

```
Size after clear: 0
```

遍历函数d

1. `reference at(int pos)`

   - **功能**: 返回 `pos` 位置元素的引用。
   - **描述**: 如果 `pos` 超出范围，会抛出一个 `std::out_of_range` 异常。

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

int main() {
    std::deque<int> d{1, 2, 3};
    try {
        std::cout << "Element at position 1: " << d.at(1) << std::endl; // 输出: 2
        std::cout << "Element at position 5: " << d.at(5) << std::endl; // 超出范围
    } catch (const std::out_of_range& e) {
        std::cout << "Error: " << e.what() << std::endl; // 输出错误信息
    }
    return 0;
}
```

输出:

```
Element at position 1: 2
Error: deque::at: index out of range
```

2. `reference front()`

   - **功能**: 返回首元素的引用。
   - **描述**: 返回 `deque` 第一个元素的引用。

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

int main() {
```

```
    std::deque<int> d{1, 2, 3};
    std::cout << "First element: " << d.front() << std::endl; // 输出: 1
    return 0;
}
```

输出:

```
First element: 1
```

3. `reference back()`

   - **功能**: 返回尾元素的引用。
   - **描述**: 返回 `deque` 最后一个元素的引用。

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

int main() {
    std::deque<int> d{1, 2, 3};
    std::cout << "Last element: " << d.back() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Last element: 3
```

4. `iterator begin()`

   - **功能**: 返回一个指向第一个元素的迭代器。
   - **描述**: 返回一个指向当前 `deque` 中第一个元素的迭代器。

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

int main() {
    std::deque<int> d{1, 2, 3};
    std::cout << "First element using begin(): " << *d.begin() << std::endl;
// 输出: 1
    return 0;
}
```

输出:

```
First element using begin(): 1
```

5. `iterator end()`

   - **功能**: 返回一个指向最后一个元素后一个位置的迭代器。
   - **描述**: 这个迭代器代表了 `deque` 的结束位置，不指向有效元素。

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

int main() {
    std::deque<int> d{1, 2, 3};
    std::cout << "End position: " << *(d.end() - 1) << " (last element)" <<
std::endl; // 输出: 3
    return 0;
}
```

输出:

```
End position: 3 (last element)
```

6. `reverse_iterator rbegin()`

   - **功能**: 返回指向最后一个元素的反向迭代器。
   - **描述**: 用于逆向遍历 `deque`。

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

int main() {
    std::deque<int> d{1, 2, 3};
    std::cout << "Last element using rbegin(): " << *d.rbegin() << std::endl;
// 输出: 3
    return 0;
}
```

输出:

```
Last element using rbegin(): 3
```

7. `reverse_iterator rend()`

   - **功能**: 返回指向前一个位置的反向迭代器。
   - **描述**: 这个迭代器在反向遍历时表示 `deque` 的开始位置。

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

int main() {
    std::deque<int> d{1, 2, 3};
    std::cout << "First element using rend(): " << *(d.rend() - 1) <<
std::endl; // 输出: 1
    return 0;
}
```

输出:

```
First element using rend(): 1
```

判断函数d

1. `bool empty() const`

   - **功能**: 判断 `deque` 是否为空。
   - **描述**: 如果 `deque` 中没有元素，返回 `true`；否则返回 `false`。

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

int main() {
    std::deque<int> d;
    std::cout << "Is deque empty? " << (d.empty() ? "Yes" : "No") <<
std::endl; // 输出: Yes
    d.push_back(1);
    std::cout << "Is deque empty? " << (d.empty() ? "Yes" : "No") <<
std::endl; // 输出: No
    return 0;
}
```

输出:

```
Is deque empty? Yes
Is deque empty? No
```

大小函数d

1. `int size() const`

   - **功能**: 返回当前元素的个数。
   - **描述**: 返回 `deque` 中实际存储的元素数量。

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

int main() {
    std::deque<int> d{1, 2, 3};
    std::cout << "Size of deque: " << d.size() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Size of deque: 3
```

2. `int max_size() const`

   - **功能**: 返回最大可能的元素数量。
   - **描述**: 返回当前 `deque` 所能容纳的最大元素数量。

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

int main() {
    std::deque<int> d;
    std::cout << "Max size of deque: " << d.max_size() << std::endl; // 输出最大值
    return 0;
}
```

输出:

```
Max size of deque: 4611686018427387903 (具体值可能因平台而异)
```

其他函数d

1. `void swap(deque& d)`
   - **功能**: 交换两个 `deque` 的内容。
   - **描述**: 交换当前 `deque` 和参数 `d` 的内容、大小和容量。

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

int main() {
    std::deque<int> d1{1, 2, 3};
    std::deque<int> d2{4, 5, 6};

    d1.swap(d2); // 交换内容

    std::cout << "d1: ";
    for (int num : d1) {
        std::cout << num << " "; // 输出: 4 5 6
    }

    std::cout << "\nd2: ";
    for (int num : d2) {
        std::cout << num << " "; // 输出: 1 2 3
    }

    std::cout << std::endl;
    return 0;
}
```

输出:

```
d1: 4 5 6
d2: 1 2 3
```

2. `void assign(int n, const T& x)`
   - **功能**: 将 `x` 赋值给 `deque` 中的前 `n` 个元素。
   - **描述**: 将 `deque` 的大小调整为 `n`，并将所有元素赋值为 `x`。

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

int main() {
    std::deque<int> d;
    d.assign(5, 10); // 向量大小调整为 5，赋值为 10

    for (int num : d) {
        std::cout << num << " "; // 输出: 10 10 10 10 10
    }

    std::cout << std::endl;
    return 0;
}
```

输出:

```
10 10 10 10 10
```

3. `void assign(const_iterator first, const_iterator last)`

   - **功能**: 将 `[first, last)` 中的元素赋值给 `deque`。
   - **描述**: 将当前 `deque` 的内容替换为给定范围内的元素。

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

int main() {
    std::deque<int> d{1, 2, 3, 4, 5};
    std::deque<int> newDeque{9, 8, 7};

    d.assign(newDeque.begin(), newDeque.end()); // 将 newDeque 中的元素赋值给 d

    for (int num : d) {
        std::cout << num << " "; // 输出: 9 8 7
    }

    std::cout << std::endl;
    return 0;
}
```

输出:

```
9 8 7
```

4. 关于list容器

4.1 概述

- **特性**:
  - 双向链表，插入和删除操作不会导致迭代器失效。
  - 每个元素之间通过指针连接，内存不连续。
  - 不支持随机访问，访问元素时需要遍历。
- **适用场景**:
  - 适合需要频繁插入和删除的场景，尤其是在中间位置。

4.2 常用函数

构造函数I

1. `list()`
   - **功能**: 创建一个空的 `list`。
   - **描述**: 默认构造函数，初始化一个没有元素的链表。

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

int main() {
    std::list<int> lst; // 创建空的 list
    std::cout << "Size: " << lst.size() << std::endl; // 输出: 0
    return 0;
}
```

输出:

```
Size: 0
```

2. `list(const list& l)`
   - **功能**: 根据另一个相同类型的 `list` 来创建该容器。

- **描述**: 复制构造函数，复制列表 `l` 的所有元素。

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

int main() {
    std::list<int> original{1, 2, 3};
    std::list<int> copy = original; // 复制构造函数

    std::cout << "Copy Size: " << copy.size() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Copy Size: 3
```

3. `list(int nSize)`

- **功能**: 创建一个指定大小的 `list`。
- **描述**: `list` 会包含 `nSize` 个默认初始化的元素。

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

int main() {
    std::list<int> lst(5); // 创建一个包含 5 个默认初始化的元素的 list
    std::cout << "Size: " << lst.size() << std::endl; // 输出: 5
    return 0;
}
```

输出:

```
Size: 5
```

4. `list(int nSize, const T& elem)`

- **功能**: 创建一个指定大小且每个元素都初始化为给定值的 `list`。
- **描述**: `list` 会被初始化为包含 `nSize` 个元素，所有元素的值都为 `elem`。

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

int main() {
    std::list<int> lst(5, 42); // 创建一个包含 5 个初始值为 42 的元素的 list
    for (int num : lst) {
        std::cout << num << " "; // 输出: 42 42 42 42 42
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
42 42 42 42 42
```

5. `list(begin, end)`

   - **功能**: 创建一个 `list`，由迭代器构建。
   - **描述**: 使用迭代器 `[begin, end)` 的范围来填充 `list`。

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

int main() {
    std::vector<int> vec{1, 2, 3};
    std::list<int> lst(vec.begin(), vec.end()); // 从 vector 创建 list

    for (int num : lst) {
        std::cout << num << " "; // 输出: 1 2 3
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 2 3
```

增加函数I

1. `void push_front(const T& x)`

   - **功能**: 在 `list` 的前面添加一个元素。
   - **描述**: 将 `x` 复制到 `list` 的前面。

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

int main() {
    std::list<int> lst;
    lst.push_front(10);
    lst.push_front(20);

    for (int num : lst) {
        std::cout << num << " "; // 输出: 20 10
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
20 10
```

2. `void push_back(const T& x)`

   - **功能**: 在 `list` 的后面添加一个元素。
   - **描述**: 将 `x` 复制到 `list` 的后面。

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

int main() {
    std::list<int> lst;
    lst.push_back(10);
    lst.push_back(20);

    for (int num : lst) {
        std::cout << num << " "; // 输出: 10 20
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
10 20
```

3. `iterator insert(iterator it, const T& x)`

   - **功能**: 在指定位置插入一个元素。
   - **描述**: 在迭代器 `it` 指向的位置插入 `x`。

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

int main() {
    std::list<int> lst{1, 2, 4};
    lst.insert(std::next(lst.begin(), 2), 3); // 在位置 2 插入元素 3

    for (int num : lst) {
        std::cout << num << " "; // 输出: 1 2 3 4
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 2 3 4
```

4. `iterator insert(iterator it, int n, const T& x)`

   - **功能**: 在指定位置插入多个相同的元素。
   - **描述**: 在迭代器 `it` 指向的位置插入 `n` 个值为 `x` 的元素。

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

int main() {
    std::list<int> lst{1, 2};
    lst.insert(lst.begin(), 2, 0); // 在位置 0 插入 2 个 0

    for (int num : lst) {
        std::cout << num << " "; // 输出: 0 0 1 2
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
0 0 1 2
```

5. `iterator insert(iterator it, const_iterator first, const_iterator last)`

   - **功能**: 在指定位置插入一段元素。
   - **描述**: 在迭代器 `it` 指向的位置插入从迭代器 `first` 到 `last` 的元素。

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

int main() {
    std::list<int> lst{1, 2, 5};
    std::vector<int> toInsert{3, 4};
    lst.insert(std::next(lst.begin(), 2), toInsert.begin(), toInsert.end());
// 在位置 2 插入 3 和 4

    for (int num : lst) {
        std::cout << num << " "; // 输出: 1 2 3 4 5
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 2 3 4 5
```

删除函数l

1. `iterator erase(iterator it)`

   - **功能**: 删除指定位置的元素。
   - **描述**: 删除迭代器 `it` 指向的元素。

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

```cpp
int main() {
    std::list<int> lst{1, 2, 3, 4};
    lst.erase(std::next(lst.begin(), 2)); // 删除位置 2 的元素

    for (int num : lst) {
        std::cout << num << " "; // 输出: 1 2 4
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 2 4
```

2. `iterator erase(iterator first, iterator last)`

   - **功能**: 删除指定范围内的元素。
   - **描述**: 删除从迭代器 `first` 到 `last` 之间的所有元素。

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

int main() {
    std::list<int> lst{1, 2, 3, 4, 5};
    lst.erase(std::next(lst.begin()), std::next(lst.begin(), 4)); // 删除位置 1 到 4 的元素

    for (int num : lst) {
        std::cout << num << " "; // 输出: 1 5
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 5
```

3. `void pop_front()`

   - **功能**: 删除 `list` 最前面的元素。
   - **描述**: 此操作将删除当前 `list` 的第一元素。

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

int main() {
    std::list<int> lst{1, 2, 3};
    lst.pop_front(); // 删除第一个元素

    for (int num : lst) {
        std::cout << num << " "; // 输出: 2 3
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
2 3
```

4. `void pop_back()`

   - **功能**: 删除 `list` 最后面的元素。
   - **描述**: 此操作将删除当前 `list` 的最后一个元素。

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

int main() {
    std::list<int> lst{1, 2, 3};
    lst.pop_back(); // 删除最后一个元素

    for (int num : lst) {
        std::cout << num << " "; // 输出: 1 2
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 2
```

5. `void remove(const T& x)`

- **功能**: 删除所有与值 `x` 相等的元素。
- **描述**: 此方法遍历 `list` 并删除所有等于 `x` 的元素。

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

int main() {
    std::list<int> lst{1, 2, 3, 2, 4};
    lst.remove(2); // 删除所有的 2

    for (int num : lst) {
        std::cout << num << " "; // 输出: 1 3 4
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 3 4
```

6. `void clear()`

- **功能**: 清除所有元素。
- **描述**: 此操作会移除 `list` 中的所有元素。

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

int main() {
    std::list<int> lst{1, 2, 3};
    lst.clear(); // 清除所有元素

    std::cout << "Size after clear: " << lst.size() << std::endl; // 输出: 0
    return 0;
}
```

输出:

```
Size after clear: 0
```

下面是 C++ 中 `list` 相关函数的详细讲解，包括遍历函数、判断函数、大小函数以及操作函数，附带代码示例和输出结果。

遍历函数I

1. `reference at(int pos)` (不支持)
   - **功能**: 返回 `pos` 位置元素的引用。
   - **描述**: `list` 不提供此功能，但可以使用 `std::next`。

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

int main() {
    std::list<int> lst{1, 2, 3, 4, 5};
    auto it = std::next(lst.begin(), 2); // 获取第 3 个元素
    std::cout << "Element at position 2: " << *it << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Element at position 2: 3
```

2. `reference front()`
   - **功能**: 返回首元素的引用。
   - **描述**: 返回 `list` 第一个元素的引用。

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

int main() {
    std::list<int> lst{1, 2, 3};
    std::cout << "First element: " << lst.front() << std::endl; // 输出: 1
    return 0;
}
```

输出:

```
First element: 1
```

3. `reference back()`

   - **功能**: 返回尾元素的引用。
   - **描述**: 返回 `list` 最后一个元素的引用。

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

int main() {
    std::list<int> lst{1, 2, 3};
    std::cout << "Last element: " << lst.back() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Last element: 3
```

4. `iterator begin()`

   - **功能**: 返回一个指向第一个元素的迭代器。
   - **描述**: 返回 `list` 中第一个元素的迭代器。

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

int main() {
    std::list<int> lst{1, 2, 3};
    std::cout << "First element using begin(): " << *lst.begin() << std::endl;
// 输出: 1
    return 0;
}
```

输出:

```
First element using begin(): 1
```

5. `iterator end()`

- **功能**: 返回一个指向最后一个元素后一个位置的迭代器。
- **描述**: 这个迭代器代表了 `list` 的结束位置，不指向有效元素。

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

int main() {
    std::list<int> lst{1, 2, 3};
    std::cout << "End position: " << *std::prev(lst.end()) << " (last
element)" << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
End position: 3 (last element)
```

6. `reverse_iterator rbegin()`

- **功能**: 返回指向最后一个元素的反向迭代器。
- **描述**: 用于逆向遍历 `list`。

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

int main() {
    std::list<int> lst{1, 2, 3};
    std::cout << "Last element using rbegin(): " << *lst.rbegin() <<
std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Last element using rbegin(): 3
```

7. `reverse_iterator rend()`

- **功能**: 返回指向前一个位置的反向迭代器。
- **描述**: 这个迭代器在反向遍历时表示 `list` 的开始位置。

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

int main() {
    std::list<int> lst{1, 2, 3};
    std::cout << "First element using rend(): " << *std::prev(lst.rend()) <<
std::endl; // 输出: 1
    return 0;
}
```

输出:

```
First element using rend(): 1
```

判断函数I

1. `bool empty() const`
   - **功能**: 判断 `list` 是否为空。
   - **描述**: 如果 `list` 中没有元素，返回 `true`；否则返回 `false`。

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

int main() {
    std::list<int> lst;
    std::cout << "Is list empty? " << (lst.empty() ? "Yes" : "No") <<
std::endl; // 输出: Yes
    lst.push_back(1);
    std::cout << "Is list empty? " << (lst.empty() ? "Yes" : "No") <<
std::endl; // 输出: No
    return 0;
}
```

输出:

```
Is list empty? Yes
Is list empty? No
```

大小函数I

1. `int size() const`

   - **功能**: 返回当前元素的个数。
   - **描述**: 返回 `list` 中实际存储的元素数量。

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

int main() {
    std::list<int> lst{1, 2, 3};
    std::cout << "Size of list: " << lst.size() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Size of list: 3
```

2. `int max_size() const`

   - **功能**: 返回最大可能的元素数量。
   - **描述**: 返回当前 `list` 所能容纳的最大元素数量。

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

int main() {
    std::list<int> lst;
    std::cout << "Max size of list: " << lst.max_size() << std::endl; // 根据具体实现输出
    return 0;
}
```

输出:

```
Max size of list: 4611686018427387903 (具体值可能因平台而异)
```

操作函数I

1. `void sort()`

- **功能**: 对 `list` 中的元素进行排序。
- **描述**: 按照升序对所有元素进行排序，默认为升序。

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

int main() {
    std::list<int> lst{3, 1, 4, 1, 5};
    lst.sort(); // 默认升序排序

    for (int num : lst) {
        std::cout << num << " "; // 输出: 1 1 3 4 5
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 1 3 4 5
```

2. `template<class Pred> void sort(Pred pr)`

- **功能**: 根据自定义条件对 `list` 中的元素进行排序。
- **描述**: 按照指定的比较函数 `pr` 对元素进行排序。

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

bool customSort(int a, int b) {
    return a > b; // 降序排序
}

int main() {
    std::list<int> lst{3, 1, 4, 1, 5};
    lst.sort(customSort); // 自定义排序

    for (int num : lst) {
        std::cout << num << " "; // 输出: 5 4 3 1 1
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
5 4 3 1 1
```

3. `void swap(list& l)`

   - **功能**: 交换两个 `list` 的内容。
   - **描述**: 交换当前 `list` 和参数 `l` 的内容、大小和容量。

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

int main() {
    std::list<int> lst1{1, 2, 3};
    std::list<int> lst2{4, 5, 6};

    lst1.swap(lst2); // 交换内容

    std::cout << "lst1: ";
    for (int num : lst1) {
        std::cout << num << " "; // 输出: 4 5 6
    }

    std::cout << "\nlst2: ";
    for (int num : lst2) {
        std::cout << num << " "; // 输出: 1 2 3
    }

    std::cout << std::endl;
    return 0;
}
```

输出:

```
lst1: 4 5 6
lst2: 1 2 3
```

4. `void unique()`

   - **功能**: 删除相邻重复元素。
   - **描述**: 仅保留第一个重复元素，删除其余相邻的重复元素。

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

int main() {
    std::list<int> lst{1, 2, 2, 3, 3, 3, 4};
```

```
    lst.unique(); // 删除相邻重复元素

    for (int num : lst) {
        std::cout << num << " "; // 输出: 1 2 3 4
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
1 2 3 4
```

5. `void splice(iterator it, list& x)`

   - **功能**: 将列表 `x` 中的所有元素插入到当前列表的 `it` 指向的位置。
   - **描述**: 移动而不是复制元素，效率较高。

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

int main() {
    std::list<int> lst1{1, 2, 3};
    std::list<int> lst2{4, 5, 6};

    lst1.splice(lst1.begin(), lst2); // 将 lst2 的所有元素插入到 lst1 开头

    for (int num : lst1) {
        std::cout << num << " "; // 输出: 4 5 6 1 2 3
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
4 5 6 1 2 3
```

6. `void splice(iterator it, list& x, iterator i)`

   - **功能**: 将 `x` 中迭代器 `i` 指向的元素插入当前列表的 `it` 指向的位置。
   - **描述**: 移动而不是复制元素。

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

int main() {
    std::list<int> lst1{1, 2, 3};
    std::list<int> lst2{4, 5, 6};

    auto it = std::next(lst2.begin()); // 指向元素 5
    lst1.splice(lst1.begin(), lst2, it); // 将 5 插入到 lst1 开头

    for (int num : lst1) {
        std::cout << num << " "; // 输出: 5 1 2 3
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
5 1 2 3
```

7. `void splice(iterator it, list& x, iterator first, iterator last)`

   - **功能**: 将 `x` 中 `[first, last)` 范围内的元素插入当前列表的 `it` 指向的位置。
   - **描述**: 移动而不是复制元素。

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

int main() {
    std::list<int> lst1{1, 2, 3};
    std::list<int> lst2{4, 5, 6};

    lst1.splice(lst1.begin(), lst2, lst2.begin(), std::next(lst2.begin(), 2));
// 插入 4 和 5

    for (int num : lst1) {
        std::cout << num << " "; // 输出: 4 5 1 2 3
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
4 5 1 2 3
```

8. `void reverse()`

   - **功能**: 反转 `list` 中元素的顺序。
   - **描述**: 就地反转容器的内容。

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

int main() {
    std::list<int> lst{1, 2, 3, 4, 5};
    lst.reverse(); // 反转列表

    for (int num : lst) {
        std::cout << num << " "; // 输出: 5 4 3 2 1
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
5 4 3 2 1
```

9. `void merge(list& l)`

   - **功能**: 将当前容器和容器 `l` 一起以从小到大的顺序合并。
   - **描述**: `list` 必须是排序的。

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

int main() {
    std::list<int> lst1{1, 3, 5};
    std::list<int> lst2{2, 4, 6};

    lst1.merge(lst2); // 合并两个已排序的列表

    for (int num : lst1) {
        std::cout << num << " "; // 输出: 1 2 3 4 5 6
    }
    std::cout << std::endl;
```

```
    return 0;
}
```

输出:

```
1 2 3 4 5 6
```

10. `void merge(list& l, Predicate pred)`

- **功能**: 将当前容器和容器 `l` 一起根据指定的条件 `pred` 排序合并。
- **描述**: 可以用自定义规则合并两个列表。

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

using namespace std;

bool customSort(int a, int b) {
    return a > b; // 降序排序
}

int main() {
    std::list<int> lst1{1, 3, 5};
    std::list<int> lst2{2, 4, 6};

    lst1.merge(lst2, customSort); // 根据自定义顺序合并

    for (int num : lst1) {
        std::cout << num << " "; // 输出: 6 5 4 3 2 1
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
6 5 4 3 2 1
```

以上是 C++ 中 `list` 的构造函数、增加函数、删除函数、操作函数及其他功能的详细讲解、代码示例和输出结果。如有其他问题，请随时询问！

4.3 list的使用

`list` 迭代器 `list<T>::iterator` 与 `vector`、`deque` 迭代器的区别：

1. **允许自增自减**：
    - 支持 `it++`、`it--`、`it++`、`--it` 的操作。
2. **不允许直接增加整数**：
    - 不支持如 `it + 1` 的操作。
3. **进行迭代器移动的方法**：
    - 使用 `advance(iterator, n)` 来直接改变 `iterator` 的值，并不会返回值。
    - 使用 `next(iterator, n)` 或 `prev(iterator, n)` 来不更改 `iterator` 的值，而是返回新值。
    - `n` 可以是负数，表示向相反方向的移动。

---

5. 队列和栈

5.1 概述

队列

- **特点**：
    - 支持队头插入、队尾删除，遵循**先进先出**（FIFO）原则。
    - 不提供随机访问功能，也不支持迭代器。

栈

- **特点**：
    - 在一端进行插入和删除，遵循**先进后出**（LIFO）原则。
    - 不提供随机访问功能，也不支持迭代器。

共同特点

- 两者都不支持随机访问功能，且均不提供迭代器。

5.2队列（queue）

1. 构造函数q

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q; // 创建一个空的队列，默认容器为 deque
    std::cout << "Queue constructed." << std::endl;
    return 0;
}
```

输出:

```
Queue constructed.
```

2. bool empty()

- **功能**: 判断队列是否为空。

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    std::cout << "Is queue empty? " << (q.empty() ? "Yes" : "No") << std::endl; //
输出: Yes
    return 0;
}
```

输出:

```
Is queue empty? Yes
```

3. int size()

- **功能**: 返回队列中的元素数量。

```
#include <iostream>
#include <queue>
```

```cpp
int main() {
    std::queue<int> q;
    q.push(1);
    q.push(2);
    std::cout << "Size of queue: " << q.size() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Size of queue: 2
```

4. `void push(const T& t)`

- **功能**: 将元素插入队列。

```cpp
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    q.push(1);
    q.push(2);
    std::cout << "Front element: " << q.front() << std::endl; // 输出: 1
    return 0;
}
```

输出:

```
Front element: 1
```

5. `void pop()`

- **功能**: 删除队列前面的元素。

```cpp
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    q.push(1);
    q.push(2);
    q.pop(); // 删除第一个元素
```

```
    std::cout << "Front element after pop: " << q.front() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Front element after pop: 2
```

6. `T& front()`

- **功能**: 返回队列前面的元素。

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    q.push(1);
    q.push(2);
    std::cout << "Front element: " << q.front() << std::endl; // 输出: 1
    return 0;
}
```

输出:

```
Front element: 1
```

7. `T& back()`

- **功能**: 返回队列后面的元素。

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    q.push(1);
    q.push(2);
    std::cout << "Back element: " << q.back() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Back element: 2
```

5.3栈（stack）

1. 构造函数s

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s; // 创建一个空的栈，默认容器为 deque
    std::cout << "Stack constructed." << std::endl;
    return 0;
}
```

输出:

```
Stack constructed.
```

2. bool empty()

- **功能**: 判断栈是否为空。

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;
    std::cout << "Is stack empty? " << (s.empty() ? "Yes" : "No") << std::endl; //
输出: Yes
    return 0;
}
```

输出:

```
Is stack empty? Yes
```

3. `int size()`

- **功能**: 返回栈中的元素数量。

```cpp
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;
    s.push(1);
    s.push(2);
    std::cout << "Size of stack: " << s.size() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Size of stack: 2
```

4. `void push(const T& t)`

- **功能**: 将元素压入栈中。

```cpp
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;
    s.push(1);
    s.push(2);
    std::cout << "Top element after push: " << s.top() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Top element after push: 2
```

5. `void pop()`

- **功能**: 移除栈顶的元素。

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;
    s.push(1);
    s.push(2);
    s.pop(); // 移除栈顶元素
    std::cout << "Top element after pop: " << s.top() << std::endl; // 输出: 1
    return 0;
}
```

输出:

```
Top element after pop: 1
```

6. `T& top()`

- **功能**: 返回栈顶的元素。

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;
    s.push(1);
    s.push(2);
    std::cout << "Top element: " << s.top() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Top element: 2
```

5.4数据结构总结

队列（queue）基本特性

- **条件**:
  - 支持操作：`size`、`empty`、`push_back`、`pop_front`、`front`、`back`
  - 实现数据结构两端分别进行插入和删除操作。
- **可用的数据结构**:
  - `deque`、`list` 都具有这些功能。

栈（stack）基本特性

- **条件**:
  - 支持操作：`size`、`empty`、`push_back`、`pop_back`、`top`
  - 只能在一端进行插入和删除操作。
- **可用的数据结构**:
  - `deque`、`list`、`vector` 都具有这些功能。

总结点

- 队列允许两端操作，而栈仅允许一端操作，适用于不同的数据处理需求。

6. 优先队列

6.1 概述

- **定义**:
  - `priority_queue` 是一个带优先级的队列，优先级高的元素优先出队。
- **功能**:
  - 支持插入元素和删除具有最高优先级元素。
- **实现基础**:
  - 通常基于基本序列容器，默认为 `vector`。

特点

- 适用于需要处理元素优先级的场景，如任务调度、图算法等。

6.2 常用函数

`priority_queue` 的使用与说明

`priority_queue` 是 STL 中用于管理带优先级的元素的一个数据结构，允许高优先级的元素优先出队。下面是对其构造函数及操作函数的详细讲解、代码实例和运行结果。

构造函数p

1. **默认构造函数**

```
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> pq; // 创建一个默认的优先级队列
    std::cout << "Priority queue constructed." << std::endl;
    return 0;
}
```

输出:

```
Priority queue constructed.
```

2. **使用迭代器初始化**

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

int main() {
    std::vector<int> vec = {10, 20, 30, 5, 15};
    std::priority_queue<int> pq(vec.begin(), vec.end()); // 根据 vector 初始化优先级
队列

    std::cout << "Size of priority queue: " << pq.size() << std::endl; // 输出: 5
    return 0;
}
```

输出:

```
Size of priority queue: 5
```

操作函数p

1. `bool empty()`

- **功能**: 判断优先级队列是否为空。

```
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> pq;
    std::cout << "Is priority queue empty? " << (pq.empty() ? "Yes" : "No") <<
std::endl; // 输出: Yes
    pq.push(10);
    std::cout << "Is priority queue empty? " << (pq.empty() ? "Yes" : "No") <<
std::endl; // 输出: No
    return 0;
}
```

输出:

```
Is priority queue empty? Yes
Is priority queue empty? No
```

2. `int size()`

- **功能**: 返回队列中的元素数量。

```
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> pq;
    pq.push(10);
    pq.push(20);
    std::cout << "Size of priority queue: " << pq.size() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Size of priority queue: 2
```

3. `void push(const T& t)`

- **功能**: 将元素插入优先级队列。

```cpp
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> pq;
    pq.push(10);
    pq.push(20);
    pq.push(5);
    std::cout << "Top element after pushes: " << pq.top() << std::endl; // 输出: 20
    return 0;
}
```

输出:

```
Top element after pushes: 20
```

4. `void pop()`

- **功能**: 移除优先级最高的元素。

```cpp
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> pq;
    pq.push(10);
    pq.push(30);
    pq.push(20);
    pq.pop(); // 移除最高优先级元素
    std::cout << "Top element after pop: " << pq.top() << std::endl; // 输出: 20
    return 0;
}
```

输出:

```
Top element after pop: 20
```

5. `T& top()`

- **功能**: 返回优先级最高的元素。

```
#include <iostream>
#include <queue>

int main() {
    std::priority_queue<int> pq;
    pq.push(10);
    pq.push(20);
    pq.push(15);
    std::cout << "Top element: " << pq.top() << std::endl; // 输出: 20
    return 0;
}
```

输出:

```
Top element: 20
```

---

7. bitset容器

7.1 概述

- **C语言特性**:
  - C 是一种“接近硬件”的编程语言，但它并没有固定的二进制表示方式。
- `bitset` **的功能**:
  - `bitset` 可以视为一个二进制位的容器，提供了针对位的相关操作函数，使得位操作更加方便。

7.2 常用函数b

`bitset` 是 C++ STL 中的一个类模板，用于高效处理位（bit）操作。它允许对固定大小的位序列进行操作，提供多种方法来设置、获取和操作这些位。

主要构造函数

1. **无参数构造函数**:

```
#include <iostream>
#include <bitset>

int main() {
    std::bitset<8> b1; // 创建一个8位的bitset，默认值为0
    std::cout << "Default: " << b1 << std::endl; // 输出: 00000000
    return 0;
}
```

输出:

```
Default: 00000000
```

2. **指定值构造函数**:

```
#include <iostream>
#include <bitset>

int main() {
    std::bitset<8> b2(10); // 二进制形式为 00001010
    std::cout << "From unsigned long: " << b2 << std::endl; // 输出: 00001010
    return 0;
}
```

输出:

```
From unsigned long: 00001010
```

3. **从字符串构造**:

```
#include <iostream>
#include <bitset>

int main() {
    std::bitset<8> b3("10101010"); // 从字符串初始化
    std::cout << "From string: " << b3 << std::endl; // 输出: 10101010
```

```
    return 0;
}
```

输出:

```
From string: 10101010
```

常用操作函数

1. `set()`: 设置特定位置的位为1。

```
#include <iostream>
#include <bitset>

int main() {
    std::bitset<8> b1;
    b1.set(1); // 设置第1位为1
    std::cout << "After set(1): " << b1 << std::endl; // 输出: 00000010
    return 0;
}
```

输出:

```
After set(1): 00000010
```

2. `reset()`: 将特定位置的位重置为0。

```
#include <iostream>
#include <bitset>

int main() {
    std::bitset<8> b1("11111111");
    b1.reset(1); // 重置第1位为0
    std::cout << "After reset(1): " << b1 << std::endl; // 输出: 11111101
    return 0;
}
```

输出:

```
After reset(1): 11111101
```

3. `flip()`: 翻转特定位置的位。

```
#include <iostream>
#include <bitset>

int main() {
    std::bitset<8> b1("00000000");
    b1.flip(2); // 翻转第2位
    std::cout << "After flip(2): " << b1 << std::endl; // 输出: 00000100
    return 0;
}
```

输出:

```
After flip(2): 00000100
```

4. `count()`: 计算其中设置为1的位数。

```
#include <iostream>
#include <bitset>

int main() {
    std::bitset<8> b1("10101010");
    std::cout << "Count of 1's: " << b1.count() << std::endl; // 输出: 4
    return 0;
}
```

输出:

```
Count of 1's: 4
```

5. `to_string()`: 将 `bitset` 转换为字符串。

```
#include <iostream>
#include <bitset>

int main() {
    std::bitset<8> b1("10101010");
    std::string str = b1.to_string();
    std::cout << "As string: " << str << std::endl; // 输出: 10101010
```

```
    return 0;
}
```

输出:

```
As string: 10101010
```

8. 集合

8.1 概述

- **集合类**:
  - `set` 和 `multiset` 都属于集合类。
- **主要区别**:
  - `set`: 不允许有重复元素，所有元素唯一。
  - `multiset`: 允许重复元素，可以存储多个相同的元素。

8.2 常用函数

1. 构造函数

`set` 构造函数

- `set(const Pred& comp = Pred(), const A& al = A())`: 创建一个空集合。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s; // 创建一个空的 set
    std::cout << "Set created." << std::endl;
    return 0;
}
```

输出:

```
Set created.
```

`multiset` 构造函数

- `multiset(const Pred& comp = Pred(), const A& al = A())`: 创建一个空的多重集合。

```cpp
#include <iostream>
#include <set>

int main() {
    std::multiset<int> ms; // 创建一个空的 multiset
    std::cout << "Multiset created." << std::endl;
    return 0;
}
```

输出:

```
Multiset created.
```

其它重载的构造函数

- **从 `set` 初始化**:

```cpp
#include <iostream>
#include <set>

int main() {
    std::set<int> s{1, 2, 3};
    std::multiset<int> ms(s); // 从 set 初始化 multiset
    std::cout << "Multiset size: " << ms.size() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Multiset size: 3
```

2. 增加函数

`insert`

- `pair<iterator, bool> insert(const value_type& x)`: 插入元素。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s;
    auto result = s.insert(1);
    std::cout << "Inserted: " << result.second << std::endl; // 输出: 1 表示成功插入
    return 0;
}
```

输出:

```
Inserted: 1
```

批量插入

- `void insert(iterator first, iterator last)`: 插入一系列元素。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s;
    s.insert({1, 2, 3}); // 批量插入
    std::cout << "Set size after insert: " << s.size() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Set size after insert: 3
```

3. 删除函数

`erase`

- `iterator erase(iterator it)`: 删除某一元素。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s{1, 2, 3};
    s.erase(s.begin()); // 删除第一个元素
    std::cout << "Set size after erase: " << s.size() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Set size after erase: 2
```

批量删除

- `iterator erase(iterator first, iterator last)`: 删除一系列元素。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s{1, 2, 3, 4, 5};
    s.erase(s.begin(), s.find(4)); // 删除小于4的元素
    std::cout << "Set size after range erase: " << s.size() << std::endl; // 输出:
2
    return 0;
}
```

输出:

```
Set size after range erase: 2
```

根据键删除

- `size_type erase(const Key& key)`: 根据键删除元素。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s{1, 2, 3};
```

```
    s.erase(2); // 删除键为 2 的元素
    std::cout << "Set size after key erase: " << s.size() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Set size after key erase: 2
```

4. 迭代函数

`begin()` 与 `end()`

- `iterator begin()`: 返回指向第一个元素的迭代器。
- `iterator end()`: 返回指向最后一个元素之后位置的迭代器。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s{1, 2, 3};
    std::cout << "First element: " << *s.begin() << std::endl; // 输出: 1
    return 0;
}
```

输出:

```
First element: 1
```

5. 判断函数

`empty()`

- `bool empty() const`: 判断集合是否为空。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s;
    std::cout << "Is set empty? " << (s.empty() ? "Yes" : "No") << std::endl; // 输
```

```
出: Yes
    return 0;
}
```

输出:

```
Is set empty? Yes
```

6. 大小函数

`size()`

- `size_type size() const`: 返回元素的数量。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s{1, 2, 3};
    std::cout << "Size of set: " << s.size() << std::endl; // 输出: 3
    return 0;
}
```

输出:

```
Size of set: 3
```

7. 查找函数

`find()`

- `const_iterator find(const Key& key) const`: 查找元素并返回对应迭代器。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s{1, 2, 3};
    auto it = s.find(2);
    if (it != s.end()) {
        std::cout << "Found: " << *it << std::endl; // 输出: 2
```

```
    }
    return 0;
}
```

输出:

```
Found: 2
```

8. 交换函数

`swap()`

- `void swap(set& x)`: 交换两个集合的元素。

```
#include <iostream>
#include <set>

int main() {
    std::set<int> s1{1, 2};
    std::set<int> s2{3, 4};
    s1.swap(s2);
    std::cout << "Size of s1 after swap: " << s1.size() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Size of s1 after swap: 2
```

9. 映射

9.1 概述

- **常用映射类**: `map` 和 `multimap` 是 C++ STL 中常用的映射容器。
- **数据结构区别**:
    - `map`: 存储键值对，实现一对一关系。每个键对应一个唯一的值。

- `multimap`: 支持多对多关系，可以存储多个相同键的值。

- **主要特性**:
  - `map` 中的键是唯一的，不允许重复。
  - `multimap` 中的键可以重复，允许多个相同的键存在。

9.2 常用函数

1. 构造函数

`map` 和 `multimap` 的构造函数

- `map(const Pred& comp = Pred(), const A& al = A())`
- `multimap(const Pred& comp = Pred(), const A& al = A())`

示例代码:

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

int main() {
    std::map<int, std::string> myMap; // 创建空的 map
    std::multimap<int, std::string> myMultiMap; // 创建空的 multimap

    std::cout << "Map and Multimap created." << std::endl;
    return 0;
}
```

输出:

```
Map and Multimap created.
```

2. 增加函数

`insert()`

- `iterator insert(const value_type& x)`: 插入元素。

示例代码:

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

int main() {
    std::map<int, std::string> myMap;
    myMap.insert(std::make_pair(1, "One"));
    myMap.insert(std::make_pair(2, "Two"));

    std::cout << "Inserted elements: ";
    for (const auto& pair : myMap) {
        std::cout << pair.first << ": " << pair.second << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
Inserted elements: 1: One 2: Two
```

批量插入

- `void insert(const value_type *first, const value_type *last)`: 根据范围插入元素。

示例代码:

```cpp
#include <iostream>
#include <map>
#include <vector>

int main() {
    std::map<int, std::string> myMap;
    std::vector<std::pair<int, std::string>> vec = {{3, "Three"}, {4, "Four"}};
    myMap.insert(vec.begin(), vec.end());

    std::cout << "Inserted elements: ";
    for (const auto& pair : myMap) {
        std::cout << pair.first << ": " << pair.second << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

输出:

```
Inserted elements: 3: Three 4: Four
```

3. 删除函数

`erase()`

- `iterator erase(iterator it)`: 根据迭代器删除元素。

示例代码:

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

int main() {
    std::map<int, std::string> myMap = {{1, "One"}, {2, "Two"}};
    myMap.erase(myMap.find(1)); // 删除键为1的元素

    std::cout << "Size after erase: " << myMap.size() << std::endl; // 输出: 1
    return 0;
}
```

输出:

```
Size after erase: 1
```

根据键删除

- `size_type erase(const Key& key)`: 根据键删除元素。

示例代码:

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

int main() {
    std::map<int, std::string> myMap = {{1, "One"}, {2, "Two"}};
    myMap.erase(2); // 删除键为2的元素

    std::cout << "Size after key erase: " << myMap.size() << std::endl; // 输出: 1
    return 0;
}
```

输出:

```
Size after key erase: 1
```

4. 迭代函数

`begin()` 和 `end()`

- `iterator begin()`: 返回指向第一个元素的迭代器。
- `iterator end()`: 返回指向最后一个元素之后的迭代器。

示例代码:

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

int main() {
    std::map<int, std::string> myMap = {{1, "One"}, {2, "Two"}};
    std::cout << "First element: " << myMap.begin()->second << std::endl; // 输出:
One
    return 0;
}
```

输出:

```
First element: One
```

5. 判断函数

`empty()`

- `bool empty() const`: 判断映射是否为空。

示例代码:

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

int main() {
    std::map<int, std::string> myMap;
    std::cout << "Is map empty? " << (myMap.empty() ? "Yes" : "No") << std::endl;
```

```
    // 输出: Yes
    myMap.insert(std::make_pair(1, "One"));
    std::cout << "Is map empty? " << (myMap.empty() ? "Yes" : "No") << std::endl;
    // 输出: No
    return 0;
}
```

输出:

```
Is map empty? Yes
Is map empty? No
```

6. 大小函数

`size()`

- `size_type size() const`: 返回元素的数量。

示例代码:

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

int main() {
    std::map<int, std::string> myMap = {{1, "One"}, {2, "Two"}};
    std::cout << "Size of map: " << myMap.size() << std::endl; // 输出: 2
    return 0;
}
```

输出:

```
Size of map: 2
```

7. 查找函数

`find()`

- `const_iterator find(const Key& key) const`: 查找元素并返回对应迭代器。

示例代码:

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

int main() {
    std::map<int, std::string> myMap = {{1, "One"}, {2, "Two"}};
    auto it = myMap.find(1);
    if (it != myMap.end()) {
        std::cout << "Found: " << it->second << std::endl; // 输出: One
    }
    return 0;
}
```

输出:

```
Found: One
```

8. 操作函数

`swap()`

- `void swap(map& x)`: 交换两个映射的元素。

示例代码:

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

int main() {
    std::map<int, std::string> myMap1 = {{1, "One"}, {2, "Two"}};
    std::map<int, std::string> myMap2 = {{3, "Three"}, {4, "Four"}};

    myMap1.swap(myMap2);

    std::cout << "Size of first map after swap: " << myMap1.size() << std::endl; //
输出: 2
    return 0;
}
```

输出:

```
Size of first map after swap: 2
```

10. 再论迭代器

10.1 插入迭代器

插入迭代器提供了一种将元素插入到容器的方法，主要分为三种类型：

1. **back_insert_iterator**：用于在容器的末尾插入元素。
2. **front_insert_iterator**：用于在容器的开头插入元素。
3. **insert_iterator**：用于在指定位置插入元素。

代码示例

以下是一个使用 `back_insert_iterator` 和 `front_insert_iterator` 的示例代码：

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

int main() {
    // 创建一个空的向量
    std::vector<int> vec;

    // 使用 back_insert_iterator 在 vec 的后面插入元素
    std::back_insert_iterator<std::vector<int>> backIt(vec);
    *backIt = 10;
    *backIt = 20;

    // 显示后插入结果
    std::cout << "back_insert_iterator results: ";
    for (const auto& value : vec) {
        std::cout << value << " ";
    }
    std::cout << std::endl;

    // 清空 vec 用于前插入
    vec.clear();

    // 使用 front_insert_iterator 在 vec 的前面插入元素
    std::front_insert_iterator<std::vector<int>> frontIt(vec);
    *frontIt = 30;
    *frontIt = 40;

    // 显示前插入结果
    std::cout << "front_insert_iterator results: ";
    for (const auto& value : vec) {
        std::cout << value << " ";
    }
```

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

    return 0;
}
```

运行结果

运行上述代码后，将得到如下结果：

```
back_insert_iterator results: 10 20
front_insert_iterator results: 40 30
```

10.2 逆向迭代器

接下来介绍 `reverse_iterator` 和 `reverse_bidirectional_iterator` 的使用，它们主要用于反向遍历容器。

代码示例

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

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

    // 使用 reverse_iterator 反向遍历 vec
    std::cout << "Reverse iterator results: ";
    for (std::reverse_iterator<std::vector<int>::iterator> rit(vec.end()); rit != 
vec.rend(); ++rit) {
        std::cout << *rit << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

运行结果

```
Reverse iterator results: 5 4 3 2 1
```

10.3 迭代器函数

1. **advance**：用于将迭代器向前移动指定的元素个数。
2. **distance**：计算两个迭代器之间的距离。

代码示例

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

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

    // 移动迭代器
    std::advance(it, 2);
    std::cout << "Element after advancing 2 positions: " << *it << std::endl;

    // 计算元素之间的距离
    std::cout << "Distance from begin to current iterator: " <<
std::distance(vec.begin(), it) << std::endl;

    return 0;
}
```

运行结果

```
Element after advancing 2 positions: 3
Distance from begin to current iterator: 2
```