所有容器

C++ 标准模板库（STL）包含了多种容器，每种容器都有其特定的用途和特点。主要分为三类：**序列式容器**、**关联式容器** 和 **无序容器**。以下是详细的分类和常用容器列表：

1. 序列式容器（Sequence Containers）

这些容器按照严格的线性顺序存储元素，允许快速访问、插入和删除操作。常见的序列式容器包括：

- `std::vector`：动态数组，支持快速的随机访问，元素可以在末尾动态增长。
- `std::deque`：双端队列，支持在两端快速插入和删除操作。
- `std::list`：双向链表，支持在任意位置快速插入和删除操作，但不支持随机访问。
- `std::forward_list`：单向链表，支持单向遍历，内存占用比 `std::list` 小。
- `std::array`：固定大小的数组，大小在编译时确定，提供比原生数组更丰富的接口。
- `std::string`：字符串类，实际上是专门用于字符操作的动态数组。

2. 关联式容器（Associative Containers）

关联容器通过键值对存储数据，数据按一定的排序规则组织和访问，通常使用平衡二叉树实现。常见的关联式容器包括：

- `std::map`：有序键值对容器，键唯一，按键排序。
- `std::multimap`：有序键值对容器，键可重复，按键排序。
- `std::set`：有序集合容器，键唯一，按值排序。
- `std::multiset`：有序集合容器，键可重复，按值排序。

3. 无序容器（Unordered Containers）

无序容器基于哈希表实现，支持常数时间复杂度的插入、查找和删除操作，但元素没有固定的顺序。常见的无序容器包括：

- `std::unordered_map`：无序键值对容器，键唯一，使用哈希表实现。
- `std::unordered_multimap`：无序键值对容器，键可重复，使用哈希表实现。
- `std::unordered_set`：无序集合容器，键唯一，使用哈希表实现。
- `std::unordered_multiset`：无序集合容器，键可重复，使用哈希表实现。

4. 容器适配器（Container Adapters）

容器适配器并不是直接的容器，而是基于现有容器（如 `deque`、`vector` 或 `list`）实现特定行为的接口。

- `std::stack`：LIFO 栈结构，通常基于 `deque` 实现。
- `std::queue`：FIFO 队列，通常基于 `deque` 实现。
- `std::priority_queue`：优先队列，基于最大堆实现，提供按优先级访问元素的能力。

5. 容器的特点比较

- `vector`：适合频繁访问和末尾插入的场景，随机访问性能高。
- `deque`：适合频繁在两端插入和删除的场景，支持双端操作。
- `list` 和 `forward_list`：适合频繁在任意位置插入和删除元素的场景，缺点是随机访问慢。
- `map` 和 `set`：需要有序存储时使用，适合需要快速查找和有序遍历的场景。
- `unordered_map` 和 `unordered_set`：适合快速查找和插入的场景，顺序无关。

以上就是 C++ STL 中的主要容器分类和常见的容器类型。每种容器在不同的应用场景下具有独特的性能优势。

vector

```
// 初始化一个 二维的matrix, 行M,列N,且值为0
vector<vector<int>> matrix(M,vector<int>(N));
//等价于下面的
vector<vector<int> > matrix(M);
for(int i=0;i<M;i++) {
    matrix[i].resize(N);
}
//等价于下面的
vector< vector<int> > matrix;
matrix.resize(M);//M行
for(int i=0;i<matrix.size();i++){
    matrix[i].resize(N);//每一行都是N列
}

// 初始化一个 二维的matrix, 行M,列N,且值自定义为data;
vector<vector<int>> matrix(M,vector<int>(N,data));
```

`std::vector` 是 C++ 标准库中的动态数组，提供了一系列的操作函数，以下是 `std::vector` 的常见操作：

1. 构造函数

- `vector()`：创建一个空的 `vector`。
- `vector(size_t n)`：创建一个包含 `n` 个元素的 `vector`。
- `vector(size_t n, const T& value)`：创建一个包含 `n` 个值为 `value` 的元素的 `vector`。
- `vector(initializer_list<T> il)`：使用初始化列表构造一个 `vector`。
- `vector(const vector& other)`：拷贝构造函数。
- `vector(vector&& other)`：移动构造函数。

2. 容量相关操作

- `size()`：返回当前 `vector` 中的元素个数。
- `max_size()`：返回 `vector` 能容纳的最大元素个数。
- `capacity()`：返回 `vector` 当前分配的内存容量，即可容纳的元素个数。
- `resize(size_t n)`：调整 `vector` 的大小为 `n`。

- `empty()`：判断 `vector` 是否为空。
- `reserve(size_t n)`：预留至少能容纳 `n` 个元素的空间。
- `shrink_to_fit()`：减少容量以适应当前的元素数量。

3. 元素访问

- `operator[]`：随机访问，返回指定位置的元素，**不进行边界检查**。
- `at(size_t index)`：返回指定位置的元素，**进行边界检查**。
- `front()`：返回第一个元素。
- `back()`：返回最后一个元素。
- `data()`：返回底层数组的指针。

4. 修改操作

- `assign()`：给 `vector` 重新赋值（用另一个容器或值进行填充）。
- `push_back(const T& value)`：在 `vector` 末尾添加元素。
- `push_back(T&& value)`：将右值引用添加到 `vector` 末尾。
- `pop_back()`：移除 `vector` 末尾的元素。
- `insert(iterator pos, const T& value)`：在指定位置插入元素。
- `erase(iterator pos)`：移除指定位置的元素。
- `clear()`：清除 `vector` 中的所有元素。
- `emplace_back(Args&&... args)`：在 `vector` 末尾直接构造元素。
- `emplace(iterator pos, Args&&... args)`：在指定位置直接构造元素。
- `swap(vector& other)`：交换两个 `vector` 的内容。

5. 迭代器相关操作

- `begin()`：返回指向第一个元素的迭代器。
- `end()`：返回指向最后一个元素之后的迭代器。
- `rbegin()`：返回指向最后一个元素的反向迭代器。
- `rend()`：返回指向第一个元素之前的反向迭代器。
- `cbegin()`：返回指向第一个元素的常量迭代器。
- `cend()`：返回指向最后一个元素之后的常量迭代器。
- `crbegin()`：返回指向最后一个元素的常量反向迭代器。
- `crend()`：返回指向第一个元素之前的常量反向迭代器。

这些操作涵盖了 `std::vector` 大部分的功能，方便进行动态数组的操作和管理。

map

```
#include <iostream>
#include <unordered_map>
```

```cpp
using namespace std;

int main() {
    unordered_map<string, int> mp;
    mp["张三"] = 20;
    mp["李四"] = 18;
    mp["王五"] = 30;

// 方式一、迭代器
cout << "方式一、迭代器" << endl;
for (auto it = mp.begin(); it != mp.end(); it++) {
    cout << it -> first << " " << it -> second << endl;
}

// 方式二、range for C++ 11版本及以上
cout << "方法二、 range for" << endl;
    for (auto it : mp) {
        cout << it.first << " " << it.second << endl;
    }

// 方法三、 C++ 17版本及以上
cout << "方法三" << endl;
    for (auto [key, val] : mp) {
        cout << key  << " " << val << endl;
    }
return 0;

}
```

`std::map` 是 C++ 标准模板库中的一种关联式容器，它按键排序存储键值对。以下是 `std::map` 的常用操作，分类讲解：

1. 构造函数

- `map()`：创建一个空的 `map`。
- `map(const map& other)`：拷贝构造函数，基于另一个 `map` 构造新 `map`。
- `map(map&& other)`：移动构造函数，将另一个 `map` 的内容转移到新的 `map` 中。
- `map(initializer_list<value_type> il)`：使用初始化列表构造一个 `map`。
- `template <class InputIterator> map(InputIterator first, InputIterator last)`：使用区间 `[first, last)` 的元素构造一个 `map`。

2. 容量相关操作

- `size()`：返回 `map` 中的键值对的数量。
- `empty()`：判断 `map` 是否为空。
- `max_size()`：返回 `map` 能容纳的最大键值对数。

3. 元素访问

- `operator[]`：通过键访问元素，如果键不存在，则插入该键并返回默认值。
- `at(const Key& key)`：通过键访问元素，如果键不存在，则抛出 `std::out_of_range` 异常。
- `begin()`：返回指向第一个键值对的迭代器。
- `end()`：返回指向最后一个键值对之后的迭代器。
- `find(const Key& key)`：查找键，返回指向该键的迭代器，如果未找到，返回 `end()`。
- `count(const Key& key)`：返回该键在 `map` 中出现的次数（结果为 0 或 1，因为 `map` 中键唯一）。
- `equal_range(const Key& key)`：返回表示等于 `key` 的元素的区间（对于 `map`，最多包含一个元素）。

4. 修改操作

- `insert(const value_type& val)`：插入键值对，如果键已存在，则不插入。
- `insert(iterator position, const value_type& val)`：在指定位置插入键值对（仅作为提示，不保证插入位置）。
- `insert(InputIterator first, InputIterator last)`：插入区间 `[first, last)` 中的所有元素。
- `emplace(Args&&... args)`：直接构造并插入键值对。
- `emplace_hint(iterator position, Args&&... args)`：在指定位置直接构造并插入键值对。
- `erase(const Key& key)`：删除指定键对应的键值对。
- `erase(iterator position)`：删除指定位置的键值对。
- `erase(iterator first, iterator last)`：删除指定区间的键值对。
- `swap(map& other)`：交换两个 `map` 的内容。
- `clear()`：移除所有键值对，清空 `map`。

5. 观察者操作

- `key_comp()`：返回用于比较键的比较函数对象。
- `value_comp()`：返回用于比较键值对的比较函数对象。

6. 迭代器相关操作

- `begin()`：返回指向第一个键值对的迭代器。
- `end()`：返回指向最后一个键值对之后的迭代器。
- `rbegin()`：返回指向最后一个键值对的反向迭代器。
- `rend()`：返回指向第一个键值对之前的反向迭代器。
- `cbegin()`：返回指向第一个键值对的常量迭代器。
- `cend()`：返回指向最后一个键值对之后的常量迭代器。
- `crbegin()`：返回指向最后一个键值对的常量反向迭代器。
- `crend()`：返回指向第一个键值对之前的常量反向迭代器。

7. 特殊操作

- `lower_bound(const Key& key)`：返回指向第一个不小于 `key` 的键值对的迭代器。
- `upper_bound(const Key& key)`：返回指向第一个大于 `key` 的键值对的迭代器。

8. 比较操作

- `operator==`、`operator!=`、`operator<`、`operator<=`、`operator>`、`operator>=`：用于比较两个 `map` 是否相等或按字典序进行比较。

`std::map` 是一个有序关联容器，自动对键进行排序，并提供高效的查找、插入和删除操作。

unordered_map

`std::unordered_map` 是 C++ 标准库中的哈希表容器，提供常数时间复杂度的查找、插入和删除操作。与 `std::map` 不同，它不保证元素的顺序。以下是 `std::unordered_map` 的常用操作，仿照 `vector` 和 `map` 的回答方式分类讲解：

1. 构造函数

- `unordered_map()`：创建一个空的 `unordered_map`。
- `unordered_map(const unordered_map& other)`：拷贝构造函数，基于另一个 `unordered_map` 构造新容器。
- `unordered_map(unordered_map&& other)`：移动构造函数，将另一个 `unordered_map` 的内容转移到新容器中。
- `unordered_map(initializer_list<value_type> il)`：使用初始化列表构造一个 `unordered_map`。
- `template <class InputIterator> unordered_map(InputIterator first, InputIterator last)`：使用区间 `[first, last)` 的元素构造一个 `unordered_map`。

2. 容量相关操作

- `size()`：返回 `unordered_map` 中的键值对的数量。
- `empty()`：判断 `unordered_map` 是否为空。
- `max_size()`：返回 `unordered_map` 能容纳的最大键值对数。

3. 元素访问

- `operator[]`：通过键访问元素，如果键不存在，则插入该键并返回默认值。
- `at(const Key& key)`：通过键访问元素，如果键不存在，则抛出 `std::out_of_range` 异常。
- `begin()`：返回指向第一个键值对的迭代器。
- `end()`：返回指向最后一个键值对之后的迭代器。
- `find(const Key& key)`：查找键，返回指向该键的迭代器，如果未找到，返回 `end()`。
- `count(const Key& key)`：返回该键在 `unordered_map` 中出现的次数（结果为 0 或 1，因为键是唯一的）。
- `equal_range(const Key& key)`：返回表示等于 `key` 的元素的区间（对于 `unordered_map`，最多包含一个元素）。

4. 修改操作

- `insert(const value_type& val)`：插入键值对，如果键已存在，则不插入。
- `insert(iterator position, const value_type& val)`：在指定位置插入键值对（通常作为插入提示）。
- `insert(InputIterator first, InputIterator last)`：插入区间 `[first, last)` 中的所有元素。
- `emplace(Args&&... args)`：直接构造并插入键值对。
- `emplace_hint(iterator position, Args&&... args)`：在指定位置直接构造并插入键值对。
- `erase(const Key& key)`：删除指定键对应的键值对。
- `erase(iterator position)`：删除指定位置的键值对。
- `erase(iterator first, iterator last)`：删除指定区间的键值对。
- `swap(unordered_map& other)`：交换两个 `unordered_map` 的内容。
- `clear()`：移除所有键值对，清空 `unordered_map`。

5. 桶相关操作

- `bucket_count()`：返回桶的数量。
- `max_bucket_count()`：返回最多可以拥有的桶数。
- `bucket_size(size_t n)`：返回桶 `n` 中的元素数量。
- `bucket(const Key& key)`：返回存储 `key` 的桶的编号。

6. 哈希函数和加载因子

- `load_factor()`：返回当前的加载因子（元素数量 / 桶数量）。
- `max_load_factor(float z)`：设置最大加载因子，如果当前加载因子超过这个值，会触发哈希表的重新哈希。
- `rehash(size_t n)`：将桶的数量重新设置为至少 `n`，保证 `bucket_count >= n`。
- `reserve(size_t n)`：调整容器的大小，使其至少能容纳 `n` 个元素而无需重新哈希。

7. 迭代器相关操作

- `begin()`：返回指向第一个键值对的迭代器。
- `end()`：返回指向最后一个键值对之后的迭代器。
- `cbegin()`：返回指向第一个键值对的常量迭代器。
- `cend()`：返回指向最后一个键值对之后的常量迭代器。

8. 比较操作

- `operator==`、`operator!=`：用于比较两个 `unordered_map` 是否相等。

9. 特殊操作

- `hash_function()`：返回 `unordered_map` 使用的哈希函数对象。
- `key_eq()`：返回 `unordered_map` 用于比较键的相等比较函数对象。

10. 性能考虑

`std::unordered_map` 使用哈希表实现，因此其主要操作——插入、删除和查找的时间复杂度在平均情况下为常数时间（O(1)），但在最坏情况下会退化为线性时间（O(n)），例如当所有元素被哈希到同一个桶时。

与 `std::map` 相比，`unordered_map` 不保证元素的顺序，但是在大多数情况下，它的操作速度更快。

set

`std::set` 是 C++ 标准库中的一种关联式容器，用于存储唯一的、排序的元素集合。它底层通常使用平衡二叉搜索树（如红黑树）实现，因此插入、删除、查找等操作的时间复杂度为 O(log n)。以下是 `std::set` 的常用操作，仿照之前的回答方式分类讲解：

1. 构造函数

- `set()`：创建一个空的 `set`。
- `set(const set& other)`：拷贝构造函数，基于另一个 `set` 构造新容器。
- `set(set&& other)`：移动构造函数，将另一个 `set` 的内容转移到新容器中。
- `set(initializer_list<value_type> il)`：使用初始化列表构造一个 `set`。
- `template <class InputIterator> set(InputIterator first, InputIterator last)`：使用区间 `[first, last)` 的元素构造一个 `set`。
- `template <class Compare> set(const Compare& comp)`：使用自定义比较函数构造一个 `set`。

2. 容量相关操作

- `size()`：返回 `set` 中的元素数量。
- `empty()`：判断 `set` 是否为空。
- `max_size()`：返回 `set` 能容纳的最大元素数。

3. 元素访问

- `find(const Key& key)`：查找键，返回指向该键的迭代器；如果未找到，返回 `end()`。
- `count(const Key& key)`：返回键在 `set` 中出现的次数（`set` 中的元素是唯一的，结果为 0 或 1）。
- `equal_range(const Key& key)`：返回表示等于 `key` 的元素的区间（对于 `set`，最多包含一个元素）。
- `lower_bound(const Key& key)`：返回指向第一个不小于 `key` 的元素的迭代器。
- `upper_bound(const Key& key)`：返回指向第一个大于 `key` 的元素的迭代器。
- `begin()`：返回指向第一个元素的迭代器。
- `end()`：返回指向最后一个元素之后的迭代器。

4. 修改操作

- `insert(const value_type& val)`：插入元素 `val`，如果元素已存在，则不插入。
- `insert(iterator position, const value_type& val)`：在指定位置插入元素（仅作为提示，插入位置不一定在提示位置）。
- `insert(InputIterator first, InputIterator last)`：插入区间 `[first, last)` 中的所有元素。
- `emplace(Args&&... args)`：直接构造并插入元素。
- `emplace_hint(iterator position, Args&&... args)`：在指定位置直接构造并插入元素。
- `erase(const Key& key)`：删除指定的元素（通过键）。
- `erase(iterator position)`：删除指定位置的元素。
- `erase(iterator first, iterator last)`：删除指定区间的元素。
- `clear()`：移除所有元素，清空 `set`。
- `swap(set& other)`：交换两个 `set` 的内容。

5. 迭代器相关操作

- `begin()`：返回指向第一个元素的迭代器。
- `end()`：返回指向最后一个元素之后的迭代器。
- `rbegin()`：返回指向最后一个元素的反向迭代器。
- `rend()`：返回指向第一个元素之前的反向迭代器。
- `cbegin()`：返回指向第一个元素的常量迭代器。
- `cend()`：返回指向最后一个元素之后的常量迭代器。
- `crbegin()`：返回指向最后一个元素的常量反向迭代器。
- `crend()`：返回指向第一个元素之前的常量反向迭代器。

6. 观察者操作

- `key_comp()`：返回用于比较键的比较函数对象。
- `value_comp()`：返回用于比较值的比较函数对象。

7. 比较操作

- `operator==`、`operator!=`、`operator<`、`operator<=`、`operator>`、`operator>=`：用于比较两个 `set` 是否相等或按字典序进行比较。

8. 性能考虑

- `std::set` 是基于平衡二叉树的有序集合，插入、删除和查找操作的时间复杂度为 O(log n)。
- 因为 `set` 中的元素是唯一的，所以重复元素会被忽略。
- `std::set` 保持元素的顺序，这使得它适用于需要有序存储的场景，比如需要有序遍历集合的场景。

9. 特殊操作

- `merge(set& source)`：将 `source` 中的元素合并到当前 `set`，不保留重复元素。

总结来说，`std::set` 是一个高效的、有序的唯一元素集合，适合需要自动排序和快速查找元素的场景。

unordered_set

`std::unordered_set` 是 C++ 标准库中的一种无序关联容器，用于存储唯一的元素集合。与 `std::set` 不同，`unordered_set` 是基于哈希表实现的，因此不保证元素的顺序，但大多数操作（如插入、删除、查找）的时间复杂度在平均情况下为常数时间 O(1)。以下是 `std::unordered_set` 的常用操作，仿照之前的 `set` 的回答方式分类讲解：

1. 构造函数

- `unordered_set()`：创建一个空的 `unordered_set`。
- `unordered_set(const unordered_set& other)`：拷贝构造函数，基于另一个 `unordered_set` 构造新容器。
- `unordered_set(unordered_set&& other)`：移动构造函数，将另一个 `unordered_set` 的内容转移到新容器中。
- `unordered_set(initializer_list<value_type> il)`：使用初始化列表构造一个 `unordered_set`。
- `template <class InputIterator> unordered_set(InputIterator first, InputIterator last)`：使用区间 `[first, last)` 的元素构造一个 `unordered_set`。

2. 容量相关操作

- `size()`：返回 `unordered_set` 中的元素数量。
- `empty()`：判断 `unordered_set` 是否为空。
- `max_size()`：返回 `unordered_set` 能容纳的最大元素数。

3. 元素访问

- `find(const Key& key)`：查找元素 `key`，返回指向该元素的迭代器；如果未找到，返回 `end()`。
- `count(const Key& key)`：返回键在 `unordered_set` 中出现的次数（因为 `unordered_set` 中元素是唯一的，结果为 0 或 1）。
- `equal_range(const Key& key)`：返回表示等于 `key` 的元素的区间（对于 `unordered_set`，最多包含一个元素）。

4. 修改操作

- `insert(const value_type& val)`：插入元素 `val`，如果元素已存在，则不插入。
- `insert(iterator position, const value_type& val)`：在指定位置插入元素（插入位置不一定是提示位置，因为 `unordered_set` 不保证顺序）。
- `insert(InputIterator first, InputIterator last)`：插入区间 `[first, last)` 中的所有元素。

- `emplace(Args&&... args)`：直接构造并插入元素。
- `emplace_hint(iterator position, Args&&... args)`：在指定位置直接构造并插入元素。
- `erase(const Key& key)`：删除指定的元素（通过键）。
- `erase(iterator position)`：删除指定位置的元素。
- `erase(iterator first, iterator last)`：删除指定区间的元素。
- `clear()`：移除所有元素，清空 `unordered_set`。
- `swap(unordered_set& other)`：交换两个 `unordered_set` 的内容。

5. 桶相关操作

- `bucket_count()`：返回桶的数量。
- `max_bucket_count()`：返回最多可以拥有的桶数。
- `bucket_size(size_t n)`：返回桶 `n` 中的元素数量。
- `bucket(const Key& key)`：返回存储 `key` 的桶的编号。

6. 哈希函数和加载因子

- `load_factor()`：返回当前的加载因子（元素数量 / 桶数量）。
- `max_load_factor(float z)`：设置最大加载因子。如果当前加载因子超过这个值，哈希表会重新哈希。
- `rehash(size_t n)`：将桶的数量重新设置为至少 `n`，保证 `bucket_count >= n`。
- `reserve(size_t n)`：调整容器的大小，使其至少能容纳 `n` 个元素而无需重新哈希。

7. 迭代器相关操作

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

8. 观察者操作

- `hash_function()`：返回 `unordered_set` 使用的哈希函数对象。
- `key_eq()`：返回 `unordered_set` 用于比较键的相等比较函数对象。

9. 比较操作

- `operator==`、`operator!=`：用于比较两个 `unordered_set` 是否相等。

10. 性能考虑

- `std::unordered_set` 的插入、删除和查找操作的平均时间复杂度为 O(1)，但在最坏情况下（哈希冲突严重时），会退化为 O(n)。
- 与 `std::set` 不同，`unordered_set` 不保证元素顺序，适用于对顺序没有要求但希望快速查找、插入和删除元素的场景。

11. 特殊操作

- `merge(unordered_set& source)`：将 `source` 中的元素合并到当前 `unordered_set`，不保留重复元素。

总结来说，`std::unordered_set` 是一个基于哈希表实现的无序唯一元素集合，提供了快速的查找、插入和删除操作。适用于不关心元素顺序、但对性能要求较高的场景。

`<algorithm>` 是 C++ 标准库中的头文件，包含了许多通用的算法函数。这些算法提供了对容器、数组和其他数据结构的操作支持，涵盖了排序、搜索、修改、计算等功能。以下是一些常用的 `algorithm` 操作，分类讲解：

priority_queue

`priority_queue` 是 C++ 标准模板库（STL）中的一种容器适配器，用于实现优先队列。优先队列是一种特殊的队列，它的特点是每次取出的元素都是优先级最高的元素。C++ 中的 `priority_queue` 是基于最大堆（默认情况下）的结构，因此默认情况下每次出队的元素都是当前队列中数值最大的元素。

使用方法

要使用 `priority_queue`，需要包含头文件：

```
#include <queue>
```

基本语法

```
std::priority_queue<int> pq; // 默认是最大堆
```

自定义优先级（最小堆）

默认情况下，`priority_queue` 是最大堆。若要实现最小堆，可以通过指定比较函数来实现：

```
std::priority_queue<int, std::vector<int>, std::greater<int>> pq_min; // 最小堆
```

```
1. priority_queue<int>
这是标准的定义方式，等价于 priority_queue<int, vector<int>, less<int>>，默认是基于
less<int> 的大根堆。


priority_queue<int> q;
底层容器：vector<int>（默认的存储容器）
默认比较器：less<int>（大根堆）
用途：维护堆顶为当前最大值。
2. priority_queue<int, vector<int>>
这是显式指定底层容器的方式，实际上与 priority_queue<int> 没有区别。


priority_queue<int, vector<int>> q;
底层容器：仍是 vector<int>（显式指定）
默认比较器：仍是 less<int>（大根堆）
它和 priority_queue<int> 的效果完全一致，仅仅是显式写出了底层容器。
```

```cpp
// 小根堆
priority_queue<int, vector<int>, greater<int>> minHeap;

// 自定义比较器
struct CustomComparator {
    bool operator()(const int& a, const int& b) const {
        return a > b; // 小根堆
    }
};
priority_queue<int, vector<int>, CustomComparator> customHeap;
```

在数据结构中，**堆（Heap）**是一种特殊的完全二叉树结构，用于实现优先队列。堆主要有两种类型：**最大堆（Max Heap）**和**最小堆（Min Heap）**。它们的特点和用途各有不同。

1. 最大堆（Max Heap）

在最大堆中：

- 每个节点的值都**大于或等于其子节点**的值。
- 堆顶（根节点）的值是整个堆中最大的元素。

特点：适用于需要快速找到最大值的场景。比如，通过最大堆实现的 `priority_queue`，可以在 (\mathcal{O}(1)) 时间复杂度内获取当前最大元素，并且插入和删除操作的时间复杂度是 (\mathcal{O}(\log n))。

最大堆示例：

```
       50
      /   \
    30     20
   /  \   /  \
  10  15 5   7
```

在这个最大堆中，每个父节点的值都比它的子节点大，堆顶 50 是最大值。

2. 最小堆（Min Heap）

在最小堆中：

- 每个节点的值都**小于或等于其子节点**的值。
- 堆顶（根节点）的值是整个堆中最小的元素。

特点：适用于需要快速找到最小值的场景。比如，通过最小堆实现的 `priority_queue`，可以在 (\mathcal{O}(1)) 时间复杂度内获取当前最小元素，插入和删除操作的时间复杂度也是 (\mathcal{O}(\log n))。

最小堆示例：

```
      5
    /   \
   10    15
  /  \   /  \
 20  30 25  40
```

在这个最小堆中，每个父节点的值都比它的子节点小，堆顶 5 是最小值。

最大堆与最小堆的应用场景

- **最大堆**：用于实现优先级队列，快速获取数据流中的最大值，解决类似 Top-K 问题。
- **最小堆**：同样用于优先级队列，快速获取最小值，解决类似找到数据流中第 K 大元素的问题。

如何在 C++ 中使用最大堆和最小堆

在 C++ 的 `priority_queue` 中，默认的优先队列实现就是最大堆。要实现最小堆，可以传入一个 `greater` 比较器：

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

int main() {
    // 默认的 priority_queue 是最大堆
    std::priority_queue<int> max_heap;
    max_heap.push(30);
    max_heap.push(10);
    max_heap.push(50);

    std::cout << "最大堆的顶元素: " << max_heap.top() << std::endl; // 输出50

    // 使用 greater 比较器创建最小堆
    std::priority_queue<int, std::vector<int>, std::greater<int>> min_heap;
    min_heap.push(30);
    min_heap.push(10);
    min_heap.push(50);

    std::cout << "最小堆的顶元素: " << min_heap.top() << std::endl; // 输出10

    return 0;
}
```

在以上代码中：

- `max_heap` 是默认的最大堆。
- `min_heap` 是通过 `std::greater<int>` 创建的最小堆。

1. `push` - 插入元素

将一个新元素插入到优先队列中，并自动调整堆结构以保持优先级顺序。

```
std::priority_queue<int> pq;
pq.push(10);
pq.push(20);
pq.push(15);
```

2. `top` - 访问优先级最高的元素

返回优先队列中的最高优先级元素（最大堆中为最大值，最小堆中为最小值），但不移除该元素。该操作时间复杂度为 (\mathcal{O}(1))。

```
int max_element = pq.top(); // 若是最大堆，返回当前最大元素
```

3. `pop` - 删除优先级最高的元素

移除优先队列中的最高优先级元素，同时调整堆结构以保持顺序。该操作时间复杂度为 (\mathcal{O}(\log n))。

```
pq.pop(); // 移除当前最大元素
```

4. `empty` - 检查是否为空

返回一个布尔值，表示优先队列是否为空。

```
if (pq.empty()) {
    std::cout << "Priority queue is empty." << std::endl;
}
```

5. `size` - 获取元素数量

返回优先队列中当前元素的数量。

```
std::cout << "Size of priority queue: " << pq.size() << std::endl;
```

常用操作

- `push`：将元素加入优先队列。
- `top`：返回优先队列中优先级最高的元素。
- `pop`：移除优先级最高的元素。
- `empty`：判断优先队列是否为空。
- `size`：返回优先队列的元素数量。

pair

`std::pair` 是 C++ 标准库中的一个模板类，用于将两个数据组合成一个单一的数据结构。这在需要将两种类型的数据一起存储时非常有用，比如键值对、坐标点等。它定义在头文件 `<utility>` 中。

下面是 `std::pair` 的基本使用方法和常见操作。

1. 创建 `std::pair`

可以通过构造函数、工厂函数 `std::make_pair` 或初始化列表创建 `pair`。

```
#include <iostream>
#include <utility>  // std::pair, std::make_pair

int main() {
    // 直接初始化
    std::pair<int, std::string> p1(1, "apple");

    // 使用 make_pair 函数创建
    auto p2 = std::make_pair(2, "banana");

    // 使用初始化列表
    std::pair<int, std::string> p3 = {3, "cherry"};

    std::cout << p1.first << ", " << p1.second << std::endl;
    std::cout << p2.first << ", " << p2.second << std::endl;
    std::cout << p3.first << ", " << p3.second << std::endl;

    return 0;
}
```

输出：

```
1, apple
2, banana
3, cherry
```

2. 访问 `pair` 中的元素

`std::pair` 提供了两个公有成员变量：`first` 和 `second`，分别存储两个元素。可以通过这两个变量访问 `pair` 的值。

```
std::pair<int, std::string> p = {1, "apple"};
std::cout << "First: " << p.first << ", Second: " << p.second << std::endl;
```

3. 比较 `std::pair`

`std::pair` 提供了常见的比较操作，包括 `<`, `>`, `<=`, `>=`, `==`, `!=`。比较是按词典顺序进行的，即先比较 `first`，若 `first` 相等，则比较 `second`。

```cpp
std::pair<int, int> p1 = {1, 5};
std::pair<int, int> p2 = {1, 10};

if (p1 < p2) {
    std::cout << "p1 is less than p2" << std::endl;
}
```

4. 用于自定义数据类型

`std::pair` 可以用于任何两种类型的组合，并且可以嵌套。比如将 `pair<int, pair<int, int>>` 用于存储多层数据。

```cpp
std::pair<int, std::pair<int, int>> nested_pair = {1, {2, 3}};
std::cout << nested_pair.first << ", "
          << nested_pair.second.first << ", "
          << nested_pair.second.second << std::endl;
```

5. 在容器中使用 `std::pair`

`std::pair` 常用于 `std::map`、`std::vector` 等容器中，例如：

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

int main() {
    std::map<int, std::string> m;
    m.insert(std::make_pair(1, "apple"));
    m[2] = "banana";

    for (const auto& p : m) {
        std::cout << p.first << ": " << p.second << std::endl;
    }

    std::vector<std::pair<int, int>> v = {{1, 2}, {3, 4}};
    for (const auto& p : v) {
        std::cout << p.first << ", " << p.second << std::endl;
    }

    return 0;
}
```

6. 交换 `pair` 的值

可以使用 `std::swap` 函数交换两个 `pair` 的值。

```cpp
std::pair<int, std::string> p1 = {1, "apple"};
std::pair<int, std::string> p2 = {2, "banana"};

std::swap(p1, p2);
std::cout << p1.first << ", " << p1.second << std::endl;
std::cout << p2.first << ", " << p2.second << std::endl;
```

7.特殊写法

在 C++ 中，可以使用结构化绑定（structured binding）来简化 `std::pair` 在范围 `for` 循环中的解构。`for (auto& [x, y] : q)` 的意思是将 `q` 中的每个 `std::pair<int, int>` 的 `first` 和 `second` 分别绑定到 `x` 和 `y`，使代码更具可读性。

```
vector<pair<int, int>> q;
for (auto& [x, y] : q)
```

algorithm

v.begin(),v.end() 不包括v.end()

1. 排序和排列相关操作

- `sort()`：对区间 `[first, last)` 进行升序排序，默认使用 `<` 比较。

  ```
  std::sort(v.begin(), v.end()); // 对容器v进行排序
  ```

- `stable_sort()`：与 `sort()` 类似，但保持相等元素的相对顺序。
- `partial_sort()`：对区间 `[first, middle)` 进行部分排序，使得 `[first, middle)` 是有序的。
- `nth_element()`：重排区间，使得第 `n` 个元素处于它在有序序列中的正确位置，左边的元素比它小，右边的元素比它大。
- `is_sorted()`：检查区间是否已经排序。
- `is_sorted_until()`：找到第一个使区间不再有序的迭代器。
- `reverse()`：反转区间内的元素顺序。

2. 搜索和查找相关操作

- `find()`：在区间 `[first, last)` 中查找等于指定值的元素，返回第一个匹配元素的迭代器。
- `find_if()`：根据条件谓词 `pred` 查找第一个匹配的元素。
- `find_if_not()`：查找第一个不满足条件的元素。
- `binary_search()`：在有序区间中执行二分查找，返回是否存在目标元素。
- `lower_bound()`：在有序区间中查找第一个不小于目标值的元素的迭代器。
- `upper_bound()`：在有序区间中查找第一个大于目标值的元素的迭代器。
- `equal_range()`：在有序区间中返回与目标值相等的元素的区间。

3. 集合相关操作

- `merge()`：合并两个有序区间并将结果存储到新位置，结果区间同样有序。
- `includes()`：检查一个有序区间是否包含另一个有序区间的所有元素。
- `set_union()`：计算两个有序集合的并集。
- `set_intersection()`：计算两个有序集合的交集。
- `set_difference()`：计算两个有序集合的差集。

- `set_symmetric_difference()`：计算两个有序集合的对称差集（即存在于一个集合但不存在于另一个集合的元素）。

4. 修改和替换相关操作

- `copy()`：将区间 `[first, last)` 的元素复制到另一个区间。
- `copy_if()`：将符合条件的元素复制到另一个区间。
- `move()`：将区间内的元素移动到另一个区间（源区间元素可能失效）。
- `transform()`：使用一元或二元操作符作用于区间中的每个元素，并将结果存储到另一区间。
- `replace()`：将区间中的所有指定值替换为另一个值。
- `replace_if()`：根据条件谓词替换区间中的值。
- `fill()`：用指定值填充区间中的所有元素。
- `swap_ranges()`：交换两个区间内的元素。
- `rotate()`：将区间的元素进行旋转，使得中间的元素移至区间开头。

5. 删除相关操作

- `remove()`：删除区间中等于指定值的元素，但不会缩减容器大小，需要配合 `erase` 使用。
- `remove_if()`：根据条件谓词删除元素，同样需要配合 `erase` 使用。
- `unique()`：删除区间中相邻的重复元素。
- `erase()`：结合容器的成员函数，用于真正删除移除的元素。

```
v.erase(std::remove(v.begin(), v.end(), value), v.end()); // 删除值为value的所有元素
```

6. 比较相关操作

- `equal()`：比较两个区间的元素是否相等。
- `lexicographical_compare()`：按字典序比较两个区间。

7. 数值算法

- `accumulate()`：计算区间内元素的总和或使用二元操作符计算结果。

```
int sum = std::accumulate(v.begin(), v.end(), 0); // 计算总和
```

- `inner_product()`：计算两个区间的内积。
- `adjacent_difference()`：计算相邻元素之间的差值。
- `partial_sum()`：计算区间的部分和序列。

8. 随机化操作

- `random_shuffle()`（C++17前）：随机打乱区间中的元素。
- `shuffle()`（C++11起）：用随机数生成器打乱区间中的元素。

```
std::shuffle(v.begin(), v.end(), std::default_random_engine(seed));
```

9. 堆操作

- `push_heap()`：将元素加入堆中，保持堆结构。
- `pop_heap()`：将堆中最大的元素移动到最后，并调整堆结构。
- `make_heap()`：将区间转化为堆。
- `sort_heap()`：对堆进行排序。

10. 其他常用算法

- `for_each()`：对区间中的每个元素执行指定操作。
- `min()`、`max()`：返回较小/较大的元素。
- `min_element()`、`max_element()`：返回区间中最小/最大的元素的迭代器。
- `clamp()`：将值限制在指定的范围内。
- `partition()`：将区间分为满足谓词的和不满足谓词的两部分。

11. 条件检查

- `all_of()`：检查区间中是否所有元素都满足某条件。
- `any_of()`：检查区间中是否至少有一个元素满足某条件。
- `none_of()`：检查区间中是否没有元素满足某条件。

12. 其他辅助函数

- `iota()`：生成连续递增的数值填充区间。

```
std::iota(v.begin(), v.end(), 0); // 填充从0开始的递增数列
```

使用示例

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric> // for accumulate

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

    // 计算总和
    int sum = std::accumulate(v.begin(), v.end(), 0);
    std::cout << "Sum: " << sum << std::endl;

    // 对数组排序
    std::sort(v.begin(), v.end(), std::greater<int>());
    std::cout << "Sorted in descending order: ";
    for (int i : v) std::cout << i << " ";
    std::cout << std::endl;
```

```
    // 查找元素
    auto it = std::find(v.begin(), v.end(), 3);
    if (it != v.end()) {
        std::cout << "Found: " << *it << std::endl;
    } else {
        std::cout << "Not found" << std::endl;
    }

    return 0;
}
```

总结来说，`<algorithm>` 头文件为开发者提供了丰富的通用算法，能够极大地简化对容器的操作。这些算法具有高度的灵活性和可复用性，适用于大多数标准容器。

string

由单引号括起来的一个字符被称作 char 型字面值，双引号括起来的零个或多个字符则构成字符串型字面值。字符串字面值的类型实际上就是由常量字符构成的数组，，编译器在每一个字符串后面添加一个空字符（'\0'），因此字符串的实际长度要比他的内容多1。

如字面值 'A' 表示的就是单独字符 A，而字符串 "A" 代表了一个包含两个字符的字符数组，分别是字母 A 和空字符。

0、常用功能汇总

```
s.insert(pos, args)      在 pos 之前插入 args 指定的字符
s.erase(pos, len)   删除从 pos 开始的 len 个字符。如果 len 省略，则删除 pos 开始的后面所有字符。返回一个指向 s 的引用。
s.assign(args)       将 s 中的字符替换为 args 指定的字符。返回一个指向 s 的引用。
s.append(args)       将 args 追加到 s 。返回一个指向 s 的引用。args 必须是双引号字符串
s.replace(range, args)  将 s 中范围为 range 内的字符替换为 args 指定的字符
s.find(args)     查找 s 中 args 第一次出现的位置
s.rfind(args)    查找 s 中 args 最后一次出现的位置
to_string(val)  将数值 val 转换为 string 并返回。val 可以是任何算术类型（int、浮点型等）
stoi(s) / atoi(c)    字符串/字符 转换为整数并返回
stof(s) / atof(s)    字符串/字符 转换为浮点数并返回
s.substr(pos, n)     从索引 pos 开始，提取连续的 n 个字符，包括 pos 位置的字符
reverse(s2.begin(), s2.end())    反转 string 定义的字符串 s2 （加头文件 <algorithm> ）
```

1、定义一个字符串

使用标准库类型 string 声明并初始化一个字符串，需要包含头文件 string。可以初始化的方式如下：

```
string s1;     // 初始化一个空字符串
string s2 = s1;    // 初始化s2，并用s1初始化
string s3(s2);     // 作用同上
string s4 = "hello world";    // 用 "hello world" 初始化 s4，除了最后的空字符外其他都拷贝到s4中
string s5("hello world");     // 作用同上
string s6(6,'a');  // 初始化s6为：aaaaaa
string s7(s6, 3);  // s7 是从 s6 的下标 3 开始的字符拷贝
string s8(s6, pos, len);  // s7 是从 s6 的下标 pos 开始的 len 个字符的拷贝
```

使用 = 的是拷贝初始化，使用 () 的是直接初始化。当初始值只有一个时，两者都可。当初始值有多个时一般来说要使用直接初始化，如上述最后一个的形式。

2、读写 string 操作

输入时遇到空格或回车键将停止。但需要注意的是只有按下回车键时才会结束输入执行，当按下空格后还能继续输入，但最终存到字符串中的只是第一个空格之前输入的字符串（开头的空白除外，程序会自动忽略开头的空白的），空格操作可以用来同时对多个字符串进行初始化，如下例

```
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
    string s1, s2, s3;     // 初始化一个空字符串
    // 单字符串输入，读入字符串，遇到空格或回车停止
    cin >> s1;
    // 多字符串的输入，遇到空格代表当前字符串赋值完成，转到下个字符串赋值，回车停止
    cin >> s2 >> s3;
    // 输出字符串
    cout << s1 << endl;
    cout << s2 << endl;
    cout << s3 << endl;
    return 0;
}
// 运行结果 //
  abc def hig
abc
def
hig
```

如果希望在最终读入的字符串中保留空格，可以使用 getline 函数，例子如下：

```
#include <iostream>
#include <string>

using namespace std;

int main(void)
{
    string s1 ;     // 初始化一个空字符串
    getline(cin , s1);
    cout << s1 << endl;  // 输出
    return 0;
}
// 结果输出 //
abc def hi
```

abc def hi

3、查询字符串信息、索引

可以用 empty size/length 查询字符串状态及长度，可以用下标操作提取字符串中的字符。

```
#include <iostream>
#include <string>
using namespace std;
int main(void)
{
    string s1 = "abc";     // 初始化一个字符串
    cout << s1.empty() << endl;  // s 为空返回 true，否则返回 false
    cout << s1.size() << endl;   // 返回 s 中字符个数，不包含空字符
    cout << s1.length() << endl;   // 作用同上
    cout << s1[1] << endl;  // 字符串本质是字符数组
    cout << s1[3] << endl;  // 空字符还是存在的
    return 0;
}
// 运行结果 //
0
3
3
b
```

4、拼接、比较等操作

```
s1+s2          // 返回 s1 和 s2 拼接后的结果。加号两边至少有一个 string 对象，不能都是字面值
s1 == s2       // 如果 s1 和 s2 中的元素完全相等则它们相等，区分大小写
s1 != s2
<, <=, >, >=   // 利用字符的字典序进行比较，区分大小写
```

5、cctype 头文件(判断字符类型：大/小写字母、标点、数字等)

cctype 头文件中含有对 string 中字符操作的库函数，如下：

```
isalnum(c)  // 当是字母或数字时为真
isalpha(c)  // 当是字母时为真
isdigit(c)  // 当是数字是为真
islower(c)  // 当是小写字母时为真
isupper(c)  // 当是大写字母时为真
isspace(c)  // 当是空白（空格、回车、换行、制表符等）时为真
isxdigit(c) // 当是16进制数字是为真
ispunct(c)  // 当是标点符号时为真（即c不是 控制字符、数字、字母、可打印空白 中的一种）
isprint(c)  // 当时可打印字符时为真（即c是空格或具有可见形式）
isgraph(c)  // 当不是空格但可打印时为真
iscntrl(c)  // 当是控制字符时为真
tolower(c)  // 若c是大写字母，转换为小写输出，否则原样输出
toupper(c)  // 类似上面的
```

6、for 循环遍历

可以使用 c++11 标准的 for(declaration: expression) 形式循环遍历，例子如下：

（如果想要改变 string 对象中的值，必须把循环变量定义为引用类型）

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;
int main(void)
{
    string s1 = "nice to meet you~";    // 初始化一个空字符串
    // 如果想要改变 string 对象中的值，必须把循环变量定义为引用类型。引用只是个别名，相当于对原始数据进行操作
    for(auto &c : s1)
        c = toupper(c);
    cout << s1 << endl; // 输出
    return 0;
}
// 运行结果 //
NICE TO MEET YOU~
7、修改 string 的操作
在 pos 之前插入 args 指定的字符。pos是一个下标或者迭代器。接受下标的版本返回一个指向 s 的引用；接受迭代器的版本返回一个指向第一个插入字符的迭代器

s.insert(pos, args)
// 在 s 的位置 0 之前插入 s2 的拷贝
s.insert(0, s2)
删除从 pos 开始的 len 个字符。如果 len 省略，则删除 pos 开始的后面所有字符。返回一个指向 s 的引用。

s.erase(pos, len)
将 s 中的字符替换为 args 指定的字符。返回一个指向 s 的引用。

s.assign(args)
将 args 追加到 s 。返回一个指向 s 的引用。args 不能是单引号字符，若是单个字符则必须用双引号表示。如，可以是 s.append("A") 但不能是 s.append('A')

s.append(args)
将 s 中范围为 range 内的字符替换为 args 指定的字符。range 或者是一个下标或长度，或者是一对指向 s 的迭代器。返回一个指向 s 的引用。

s.replace(range, args)
// 从位置 3 开始，删除 6 个字符，并插入 "aaa".删除插入的字符数量不必相等
s.replace(3, 6, "aaa")
```

8、string 搜索操作

`std::string` 类的 `find` 方法用于查找子字符串或字符在字符串中的首次出现位置。它返回一个 `size_t` **类型的值**，代表匹配位置的索引。如果未找到，则返回 `std::string::npos`。

搜索操作返回指定字符出现的下标，如果未找到返回 npos

```
s.find(args)  // 查找 s 中 args 第一次出现的位置
s.rfind(args)  // 查找 s 中 args 最后一次出现的位置
```

如果找到了子字符串，则返回子字符串的起始位置（索引）。

如果未找到，返回 `std::string::npos`。

在 s 中查找 args 中任何一个字符 最早/最晚 出现的位置

```
s.find_first_of(args)  // 在 s 中查找 args 中任何一个字符最早出现的位置
s.find_last_of(args)  // 在 s 中查找 args 中任何一个字符最晚出现的位置
```

例如：

```
string s1 = "nice to meet you~";
cout << s1.find_first_of("mey") << endl; // 输出结果为 3，'e' 出现的最早
```

在 s 中查找 第一个/最后一个 不在 args 中的字符的位置

```
s.find_first_not_of(args)  // 查找 s 中 第一个不在 args 中的字符的位置
s.find_last_not_of(args)  // 查找 s 中 最后一个不在 args 中的字符的位置
```

例如：

```
string s1 = "nice to meet you~";
cout << s1.find_first_not_of("nop") << endl; // 输出结果为 1 ，'i' 不在 "nop" 里
```

9、string、char 型与数值的转换

1、将数值 val 转换为 string 。val 可以是任何算术类型（int、浮点型等）。

```
string s = to_string(val)
```

2、转换为整数并返回。返回类型分别是 int、long、unsigned long、long long、unsigned long long。b 表示转换所用的进制数，默认为10，即将字符串当作几进制的数转换，最终结果仍然是十进制的表示形式 。p 是 size_t 指针，用来保存 s 中第一个非数值字符的下标，默认为0，即函数不保存下标，该参数也可以是空指针，在这种情况下不使用。

```
stoi(s)
// 函数原型 int stoi (const string&  str, size_t* idx = 0, int base = 10);
stoi(s, p, b)
stol(s, p, b)
stoul(s, p, b)
stoll(s, p, b)
stoull(s, p, b)
// 例如
string s1 = "11";    // 初始化一个空字符串
int a1 = stoi(s1);
cout << a1 << endl; // 输出 11
int a2 = stoi(s1, nullptr, 8);
cout << a2 << endl; // 输出 9
```

```
int a3 = stoi(s1, nullptr, 2);
```

cout << a3 << endl; // 输出 3
3、转换为浮点数并返回。返回类型分别是 float、double、long double 。p 是 size_t 指针，用来保存 s 中第一个非数值字符的下标，默认为0，即函数不保存下标，该参数也可以是空指针，在这种情况下不使用。

```
stof(s)
stof(s, p)
stod(s, p)
stold(s, p)
```

4、char 型转数值。注意传入的参数是指针类型，即要对字符取地址

```
atoi(c)
// 函数原型 int atoi(const char *_Str)
atol(c)
atoll(c)
atof(c)
```

10、字符串反转

使用 头文件中的 reverse() 方法：

```
string s2 = "12345";    // 初始化一个字符串
reverse(s2.begin(), s2.end()); // 反转 string 定义的字符串 s2
cout << s2 << endl; // 输出 54321
```

11、提取字串

使用 string ss = s.substr(pos, n) 。从索引 pos 开始，提取连续的 n 个字符，包括 pos 位置的字符。函数原型：

```
inline std::__cxx11::string std::__cxx11::string::substr(std::size_t __pos,
std::size_t __n)
```

12.erase

`erase` 方法的工作原理：

`std::string::erase` 方法的签名通常是：

```
string& erase(size_t pos = 0, size_t len = npos);
```

- `pos`：起始删除位置的索引，默认值是 `0`，表示从字符串的开始删除。
- `len`：从 `pos` 开始删除的字符数，默认值是 `npos`，表示删除从 `pos` 到字符串末尾的所有字符。

  `erase(0, 0)` **不会删除任何字符**

atomic

C++ 中的 `std::atomic` 提供了一套高效的、线程安全的原子操作，允许开发者在不使用锁的情况下实现并发编程。以下是 C++ 标准库中关于 `std::atomic` 的操作大全，包括常见的函数和功能。

---

1. 初始化与基本操作

初始化

`std::atomic` 类型可以用于多种类型（如整型、布尔型、指针等），支持以下初始化方式：

```
std::atomic<int> atomic_int(0);  // 初始化为 0
std::atomic<bool> atomic_bool(false);  // 初始化为 false
std::atomic<int*> atomic_ptr(nullptr); // 初始化为空指针
```

存储与加载

- `store(value)`：以原子方式存储值。
- `load()`：以原子方式读取值。

示例：

```
std::atomic<int> atomic_value(10);
atomic_value.store(20);       // 设置值为 20
int val = atomic_value.load(); // 获取当前值
```

---

2. 原子操作方法

基本原子操作

- `fetch_add(value)`：原子地增加值。
- `fetch_sub(value)`：原子地减少值。
- `fetch_and(value)`：按位与。
- `fetch_or(value)`：按位或。
- `fetch_xor(value)`：按位异或。

示例：

```
std::atomic<int> counter(0);
int old_value = counter.fetch_add(1); // 增加 1，返回旧值
int new_value = counter.fetch_sub(2); // 减少 2，返回旧值
```

增强版操作

- `exchange(value)`：原子地设置新值，返回旧值。
- `compare_exchange_weak(expected, desired)`：CAS（Compare-And-Swap）操作。
- `compare_exchange_strong(expected, desired)`：强版本的 CAS。

示例：

```
std::atomic<int> atomic_val(10);

// exchange 操作
int old = atomic_val.exchange(20); // 设置为 20，返回旧值 10

// compare_exchange 操作
int expected = 10;
bool success = atomic_val.compare_exchange_weak(expected, 15);
// 如果 atomic_val 是 10，则修改为 15 并返回 true，否则返回 false
```

- `weak` 版本更高效，但可能会偶尔失败，即使满足条件（适合循环使用）。
- `strong` 版本总是准确，但可能稍慢。

---

3. 内存顺序控制

内存顺序模型

原子操作支持多种内存顺序模型，用于控制操作的可见性：

- `memory_order_relaxed`：不提供顺序保证。
- `memory_order_consume`：最弱的依赖排序。
- `memory_order_acquire`：保证操作前的读写有序。
- `memory_order_release`：保证操作后的读写有序。
- `memory_order_acq_rel`：结合了 acquire 和 release。
- `memory_order_seq_cst`（默认）：提供全局顺序一致性。

示例：

```
std::atomic<int> atomic_data(0);
atomic_data.store(10, std::memory_order_relaxed);  // 放宽顺序限制
int value = atomic_data.load(std::memory_order_acquire); // 确保读取前所有写入可见
```

---

4. 常用类型的支持

整型操作

支持基本整型类型（`int`, `long`, `unsigned int`, 等）的原子操作：

```cpp
std::atomic<int> atomic_counter(0);
atomic_counter.fetch_add(1);  // 增加 1
atomic_counter.fetch_sub(1);  // 减少 1
atomic_counter.store(100);    // 设置值为 100
```

布尔操作

布尔类型的原子操作：

```cpp
std::atomic<bool> atomic_flag(false);
atomic_flag.store(true);  // 设置为 true
bool flag = atomic_flag.load();  // 获取值
```

指针操作

指针类型可以用 `std::atomic` 操作：

```cpp
std::atomic<int*> atomic_ptr(nullptr);
int value = 42;
atomic_ptr.store(&value);  // 设置指针
int* ptr = atomic_ptr.load();  // 获取指针
```

指针还支持指针算术：

```cpp
std::atomic<int*> atomic_ptr(&value);
atomic_ptr.fetch_add(1);  // 移动指针位置
atomic_ptr.fetch_sub(1);  // 逆向移动指针
```

---

5. 标志位操作

`std::atomic_flag`

`std::atomic_flag` 是最简单的锁原语，仅支持两个状态（设置和清除），无默认构造函数，必须初始化：

- `test_and_set()`：测试并设置标志，返回标志的旧值。
- `clear()`：清除标志。

示例：

```cpp
std::atomic_flag flag = ATOMIC_FLAG_INIT;
if (!flag.test_and_set()) {
    // 标志位为 false，现在已设置为 true
    std::cout << "Lock acquired\n";
}
flag.clear(); // 清除标志位
```

---

6. 高级功能

自旋锁

使用 `std::atomic_flag` 实现自旋锁：

```
std::atomic_flag lock = ATOMIC_FLAG_INIT;

void acquire_lock() {
    while (lock.test_and_set(std::memory_order_acquire)); // 自旋直到成功
}

void release_lock() {
    lock.clear(std::memory_order_release);
}
```

步骤解释：

- `lock.test_and_set(std::memory_order_acquire)`

  这行代码的作用是：

  - **测试**：首先检查 `lock` 的值。
  - **设置**：如果 `lock` 是 `false`，则将其设置为 `true`。
  - **返回值**：`test_and_set()` 返回 `lock` 之前的值（即之前的 `false` 或 `true`）。

自旋逻辑：

- `test_and_set()` 是一个原子操作，它会确保检查和更新 `lock` 的值是一个不可分割的操作。
- 如果 `lock` 原本是 `false`，则它会被设置为 `true`，并且 `test_and_set()` 会返回 `false`，表示成功获取锁。
- 如果 `lock` 已经是 `true`（意味着已经有其他线程获取了锁），`test_and_set()` 会返回 `true`，表示获取锁失败。此时，`while` 循环会继续执行，直到成功获取锁。
- `lock.clear(std::memory_order_release)` 清除 `lock` 的值，将其设置为 `false`，表示释放锁。
- `clear()` 是一个原子操作，确保 `lock` 的值更新是安全的，其他线程能够看到这一更新。

内存顺序：

- `std::memory_order_acquire` 表示对于该操作之前的所有操作（特别是对共享资源的操作）必须在该操作之后执行，从而确保锁的获取是可见的。这意味着在获取锁之前，线程会确保之前的操作（如更新数据）已经完成。
- `std::memory_order_release` 确保在释放锁之前，当前线程对共享数据的修改已经完成，并且这些修改对其他线程是可见的。

原子队列

通过 `std::atomic` 实现简单的无锁队列：

```cpp
struct Node {
    int value;
    Node* next;
};

std::atomic<Node*> head(nullptr);

void push(int value) {
    Node* new_node = new Node{value, nullptr};
    new_node->next = head.load(std::memory_order_relaxed);
    while (!head.compare_exchange_weak(new_node->next, new_node));
}
```

---

7. 特殊原子类型

`std::atomic<T*>`

用于多线程中的指针操作，提供额外的功能如指针算术。

`std::atomic<std::shared_ptr<T>>`

在 C++20 中引入，支持原子操作管理共享指针。

---

8. 并发性能优化

- 使用 `memory_order_relaxed` 放宽顺序要求，提升性能。
- 使用 `fetch_add` 等批量操作，减少竞争。
- 在高并发场景中，结合 CAS 操作实现无锁数据结构。

总结

C++ 的 `std::atomic` 提供了多种灵活的原子操作，并且支持各种数据类型和内存顺序模型。熟练使用这些操作可以帮助开发者设计高效的无锁并发程序，同时减少锁的开销和死锁风险。

以下是C++中常用的正则表达式操作的详细指南，包括 `std::regex` 相关的类和方法：

---

regex

1. 常用正则表达式库组件

组件	功能
`std::regex`	用于定义和存储正则表达式模式。
`std::smatch`	用于存储匹配结果，适用于 `std::string`。
`std::cmatch`	用于存储匹配结果，适用于C风格字符串。

组件	功能
`std::regex_match`	判断整个字符串是否与正则表达式匹配。
`std::regex_search`	判断字符串中是否包含与正则表达式匹配的子字符串，并获取匹配位置。
`std::regex_replace`	用于将字符串中匹配的部分替换为指定的新字符串。

---

2. 常见正则表达式操作

2.1 定义正则表达式模式

```
#include <regex>
std::regex pattern1("[a-zA-Z]+"); // 匹配字母
std::regex pattern2("\\d{3}-\\d{2}-\\d{4}"); // 匹配类似社会安全号码的格式
```

2.2 判断是否完全匹配（`std::regex_match`）

适用于验证整个字符串是否与模式完全匹配。

```
#include <iostream>
#include <regex>

int main() {
    std::string str = "abc123";
    std::regex pattern("[a-z]+\\d+");

    if (std::regex_match(str, pattern)) {
        std::cout << "完全匹配" << std::endl;
    } else {
        std::cout << "不匹配" << std::endl;
    }
    return 0;
}
```

2.3 搜索子字符串（`std::regex_search`）

适用于查找字符串中是否包含某个正则模式匹配的部分。

```
#include <iostream>
#include <regex>

int main() {
    std::string str = "My number is 123-45-6789.";
    std::regex pattern("\\d{3}-\\d{2}-\\d{4}");
    std::smatch match;

    if (std::regex_search(str, match, pattern)) {
        std::cout << "找到匹配: " << match.str() << std::endl;
    } else {
        std::cout << "未找到匹配" << std::endl;
    }
```

```
    return 0;
}
```

2.4 替换匹配部分（`std::regex_replace`）

将字符串中匹配的部分替换为指定的新内容。

```
#include <iostream>
#include <regex>

int main() {
    std::string str = "abc123def456";
    std::regex pattern("\\d+"); // 匹配数字
    std::string result = std::regex_replace(str, pattern, "X");

    std::cout << "替换结果: " << result << std::endl; // 输出: abcXdefX
    return 0;
}
```

---

3. 高级用法

3.1 捕获组

使用捕获组提取匹配内容。

```
#include <iostream>
#include <regex>

int main() {
    std::string str = "Date: 2024-12-30";
    std::regex pattern("(\\d{4})-(\\d{2})-(\\d{2})");
    std::smatch match;

    if (std::regex_search(str, match, pattern)) {
        std::cout << "年份: " << match[1] << ", 月份: " << match[2] << ", 日期: "
<< match[3] << std::endl;
    }
    return 0;
}
```

3.2 标志（flags）

正则表达式支持多种标志：

- `std::regex_constants::icase`：忽略大小写。
- `std::regex_constants::nosubs`：禁用捕获组。
- `std::regex_constants::optimize`：优化匹配速度。
- `std::regex_constants::multiline`：多行模式。

```
#include <iostream>
#include <regex>
```

```cpp
int main() {
    std::string str = "Hello World";
    std::regex pattern("hello", std::regex_constants::icase); // 忽略大小写

    if (std::regex_search(str, pattern)) {
        std::cout << "找到匹配" << std::endl;
    } else {
        std::cout << "未找到匹配" << std::endl;
    }
    return 0;
}
```

3.3 多次匹配

使用迭代器获取字符串中所有匹配项。

```cpp
#include <iostream>
#include <regex>

int main() {
    std::string str = "abc123def456ghi789";
    std::regex pattern("\\d+");
    auto begin = std::sregex_iterator(str.begin(), str.end(), pattern);
    auto end = std::sregex_iterator();

    for (auto it = begin; it != end; ++it) {
        std::cout << "匹配: " << it->str() << std::endl;
    }
    return 0;
}
```

---

4. 注意事项

1. 特殊字符转义

   ：正则表达式中的特殊字符（如

   ```
   .*+?^${}()|[]\
   ```

   ）需要使用双斜杠转义。

   - 示例：`\\d` 表示匹配数字。

2. **性能问题**：正则表达式复杂度较高时，可能会影响性能，应避免滥用。
3. **正则标准库限制**：C++的正则支持不如Python等语言强大，在使用复杂正则表达式时可能需要适当拆解问题。

---

5. 常见正则表达式模式

正则表达式	匹配内容
`\\d+`	一个或多个数字
`[a-zA-Z]+`	一个或多个字母
`\\w+`	一个或多个单词字符（字母、数字、下划线）
`\\s+`	一个或多个空白字符
`^`	字符串的开头
`$`	字符串的结尾
`.*`	任意字符

基础正则表达式模式

编程语言中的字符串表示：如果你在 C++、Python 或其他编程语言中书写正则表达式，你通常需要在字符串中输入反斜杠时使用 `\\`，因为单个 `\` 会被解释为转义符号。为了正确表示反斜杠，你需要写 `\\` 来告诉编译器这是一个字面上的反斜杠。

```
std::regex r("\\d+"); // 匹配一个或多个数字
```

模式	描述	示例匹配内容
`\d`	匹配一个数字字符，相当于 `[0-9]`	`1`, `7`, `0`
`\D`	匹配一个非数字字符	`a`, `#`, `z`
`\w`	匹配一个单词字符，相当于 `[a-zA-Z0-9_]`	`a`, `Z`, `1`, `_`
`\W`	匹配一个非单词字符	`@`, `#`, `!`
`\s`	匹配一个空白字符（空格、制表符、换行符等）	`` ` ``, `\t`, `\n` ``
`\S`	匹配一个非空白字符	`a`, `1`, `#`
`.`	匹配除换行符 `\n` 以外的任意单个字符	`a`, `1`, `@`
`^`	匹配字符串的开始	`^Hello` 匹配 `Hello world`
`$`	匹配字符串的结束	`world$` 匹配 `Hello world`
`[abc]`	匹配括号内任意一个字符	`a`, `b`, `c`
`[^abc]`	匹配括号内未列出的字符	`d`, `1`, `@`
`[a-z]`	匹配小写字母范围	`a`, `z`, `m`

模式	描述	示例匹配内容
`[A-Z]`	匹配大写字母范围	`A`, `Z`, `M`
`[0-9]`	匹配数字范围	`0`, `9`, `5`
`\b`	匹配单词边界	`\bcat\b` 匹配 `cat` 但不匹配 `catalog`
`\B`	匹配非单词边界	`\Bcat\B` 匹配 `catalog`

常见正则表达式示例

功能描述	正则表达式模式	匹配示例
验证邮箱	`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`	`example@mail.com`
验证电话号码（仅数字）	`^\d{10}$`	`1234567890`
匹配日期（格式：YYYY-MM-DD）	`^\d{4}-\d{2}-\d{2}$`	`2023-12-30`
匹配 IP 地址	`^((25[0-5]	2[0-4]\d
验证 URL	`^https?:\/\/[^\s/$.?#].[^\s]*$`	`https://www.example.com`
匹配 HTML 标签	`<[^>]*>`	`<div>`, `<span class="x">`
匹配重复单词	`\b(\w+)\s+\1\b`	`word word`
匹配十六进制颜色代码	`^#?([a-fA-F0-9]{6} | [a-fA-F0-9]{3})$`	
验证强密码（至少8个字符，含大小写字母、数字、特殊字符）	`^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$`	`Abcd1234@`

1. `\*`：零次或多次

- 匹配前面的元素零次或多次。
- **示例**：`a*` 匹配零个或多个字符 `a`，包括空字符串。

2. `+`：一次或多次

- 匹配前面的元素一次或多次。
- **示例**：`a+` 匹配一个或多个字符 `a`。

3. `?`：零次或一次

- 匹配前面的元素零次或一次。
- **示例**：`a?` 匹配零个或一个字符 `a`，包括空字符串。

4. `{n}`：精确匹配 n 次

- 匹配前面的元素恰好 n 次。
- **示例**：`a{3}` 匹配连续的三个字符 `a`，即 `"aaa"`。

5. `{n,}`：至少匹配 n 次

- 匹配前面的元素至少 n 次。
- **示例**：`a{2,}` 匹配两个或更多的字符 `a`，如 `"aa"`, `"aaa"`。

6. `{n,m}`：匹配 n 次到 m 次

- 匹配前面的元素至少 n 次，但不超过 m 次。
- **示例**：`a{2,4}` 匹配两个到四个字符 `a`，即 `"aa"`, `"aaa"`, `"aaaa"`。

7. `\*?`、`+?`、`??`：懒惰匹配（非贪婪匹配）

- 这些量词会尽可能少地匹配字符。
- 示例
  :
  - `a*?` 匹配零个或多个字符 `a`，但尽可能少地匹配。
  - `a+?` 匹配一个或多个字符 `a`，尽可能少地匹配。

8. `{n,m}?`：懒惰匹配，匹配 n 到 m 次

- 匹配前面的元素至少 n 次，但不超过 m 次，并尽可能少地匹配。
- **示例**：`a{2,4}?` 匹配两个到四个字符 `a`，尽可能少地匹配。

总结:

- `*`：零次或多次
- `+`：一次或多次
- `?`：零次或一次
- `{n}`：精确匹配 n 次
- `{n,}`：至少匹配 n 次
- `{n,m}`：匹配 n 次到 m 次
- `*?`、`+?`、`??`：懒惰匹配（非贪婪匹配）

总结

C++中的`std::regex`功能强大，适合字符串匹配、提取和替换任务。通过灵活运用正则表达式模式和方法，可以高效地完成各种字符串操作。如果正则表达式复杂，建议使用工具（如Regex101）辅助调试。

erase不同

1. `std::string::erase`

`std::string::erase`用于删除字符串中的字符。它有几种重载形式：

语法：

```
std::string& erase(size_t pos = 0, size_t len = npos);
```

- `pos`：要删除的起始位置（从 0 开始）。
- `len`：要删除的字符数量。如果不指定，默认删除从`pos`到字符串末尾的所有字符。

例子：

```
std::string s = "Hello, World!";
s.erase(5, 7);  // 删除从位置 5 开始的 7 个字符，结果是 "Hello"
```

2. `std::vector::erase`

`std::vector::erase`用于删除`std::vector`中的元素。它的重载形式有两种，支持删除单个元素或一段范围内的元素：

语法：

```
iterator erase(iterator pos);
iterator erase(iterator first, iterator last);
```

- `pos`：指向要删除元素的迭代器。
- `first, last`：指向要删除的元素范围的迭代器。

例子：

```
std::vector<int> vec = {1, 2, 3, 4, 5};
vec.erase(vec.begin() + 2);  // 删除索引为 2 的元素，结果是 {1, 2, 4, 5}

std::vector<int> vec2 = {1, 2, 3, 4, 5};
vec2.erase(vec2.begin() + 1, vec2.begin() + 4);  // 删除索引从 1 到 3 的元素，结果是
{1, 5}
```

堆与堆排序

一、基础知识
堆结构是一个用数组实现的完全二叉树结构，分为大根堆和小根堆两种，其性质分别为：
(1)大根堆：将数组还原成完全二叉树结构后，每个节点的值都不大于其父节点的值；
(2)小根堆：将数组还原成完全二叉树结构后，每个节点的值都不小于其父节点的值。
本文假设堆中实际记录元素的数组从下标为0的位置开始存储堆中的数据。完全二叉树中的父子关系可以

转换为数组的下标变换关系，比如对于节点i(数组中对应nums[i]位置的元素)，其左孩子节点应为数组中的nums[2*i* + 1]，*其右孩子节点应为数组中的nums[2*(i + 1)]，父节点应为数组中的nums[(i - 1) / 2]。

二、堆结构实现
1.接口定义
堆结构需要实现两个接口：
(1)push：将一个新元素插入堆结构中，并维护堆结构(保证修改后仍满足堆的性质)
(2)pop：(大根堆)弹出并返回堆中最大的元素，并维护堆结构。(小根堆)弹出并返回堆中最小的元素，并维护堆结构。

分析这两个函数实现的具体细节

push
先将插入的元素放在内部数组的末端，实现数据的插入。为了维护大根堆结构，首先将新插入元素与其父节点比较大小，若比父节点大则交换这两个节点，循环与父节点比较的过程，两个停止条件：
(1)新插入的节点移动到根节点则停止；
(2)新插入的节点移动到某个位置，其值不再大于此时的父节点值则停止。
push函数中维护堆结构的部分调用heapInsert函数实现。最后，更新堆中元素个数。

pop
当前数组中首元素即最大元素，记录该值，选取当前数组末尾元素替换首元素，并收缩堆元素的范围。
由于补位的末尾元素可能破坏堆的结构，需要对该元素进行判断，判断的方法如下：
(1)若补位元素的值大于其左子节点和右子节点中的较大值，满足堆结构，判断结束；
(2)若补位元素无左子节点和右子节点，满足堆结构，判断结束；
(3)除上述两种情况，该元素至少存在左子节点，将补位元素与当前的左子节点和右子节点(若存在)中的较大值进行比较，记录较大值所在的下标，交换较大值节点与该补位节点；
循环判断上述过程至满足结束判断条件。

2.代码实现
以下以大根堆为例，介绍大根堆的实现方法：

```
#include <vector>
#include <iostream>
using namespace std;

template <class T>
class heap
{
public:
    heap() : m_size(0) {}
    ~heap() = default;

    // 插入元素
    void push(T elem)
    {
        nums.
        nums.push_back(elem);
        heapInsert(m_size);
        m_size++;
    }
    // 弹出堆顶
    T pop()
    {
        T res = nums[0];
        nums[0] = nums[m_size - 1];
```

```
        --m_size;
        nums.pop_back();//移除最后一个元素
        heapify(0);
        return res;
    }

private:

    vector<T> nums;
    int m_size;

public:
    void heapInsert(int index)
    {
        int parent = (index - 1) / 2;
        while (parent != index && nums[parent] < nums[index])
        {
            swap(nums[index], nums[parent]);
            index = parent;
            parent = (index - 1) / 2;
        }
    }


    void heapify(int index)
    {
        int largest = index;          // 假设当前节点是最大值
    int left = 2 * index + 1;     // 左子节点
    int right = 2 * index + 2;    // 右子节点

    // 如果左子节点比当前节点大
    if (left < m_size && nums[left] > nums[largest])
        largest = left;

    // 如果右子节点比当前最大值大
    if (right < m_size && nums[right] > nums[largest])
        largest = right;

    // 如果最大值不是当前节点，交换并递归
    if (largest != index)
    {
        swap(nums[index], nums[largest]);
        heapify(largest);
    }
    }
    // 打印验证函数
    void print()
    {
        for (int i = 0; i < m_size; ++i)
        {
            cout << nums[i] << " ";
        }
        cout << endl;
    }
};
```

```
int main(){
    heap<int> h;
    for(int i = 10 ; i>=0 ; i--){
        int a = rand() % 100;
        h.push(a);
    }
    h.print();
    for(int i = 10 ; i>=0 ; i--){
        cout<<h.pop()<<" ";
    }
}
```

插入操作的时间复杂度为O(logN)，返回并删除最大元素的时间复杂度为O(logN)。

三、堆排序
方法一：利用priority_queue结构实现堆排序
可以借助C++中的priority_queue容器实现堆排序，默认为大根堆，具体代码如下：

```
#include <queue>
using namespace std;

void heapSort(vector<int>& nums){
    priority_queue<int, vector<int>> q;
    for(int i = 0; i < nums.size(); ++i){
        q.push(nums[i]);
    }
    for(int i = nums.size() - 1; i >= 0; --i){
        nums[i] = q.top(); q.pop();
    }
}
```

时间复杂度为O(NlogN)，该方法中使用额外容器实现堆结构，空间复杂度为O(N)。

方法二：借助堆结构性质实现空间复杂度为O(1)的堆排序

```
// 调整以 index 为根节点的子堆，n 是堆的大小
void heapify(vector<int>& nums, int index, int n) {
    int left = index * 2 + 1; // 左子节点索引
    int right = left + 1;     // 右子节点索引
    int largest = index;      // 假设当前节点为最大值

    // 如果左子节点存在且比当前最大值大，更新最大值索引
    if (left < n && nums[left] > nums[largest]) {
        largest = left;
    }
    // 如果右子节点存在且比当前最大值大，更新最大值索引
    if (right < n && nums[right] > nums[largest]) {
        largest = right;
    }

    // 如果最大值不是当前节点，交换并递归调整
    if (largest != index) {
        swap(nums[index], nums[largest]);
        heapify(nums, largest, n); // 递归调整
    }
```

```
}

// 堆排序
void heapSort(vector<int>& nums) {
    int n = nums.size();

    // 建堆，从最后一个非叶子节点开始调整
    for (int i = n / 2 - 1; i >= 0; --i) {
        heapify(nums, i, n);
    }

    // 排序，将堆顶元素移到数组末尾，并调整剩余堆
    for (int i = n - 1; i > 0; --i) {
        swap(nums[0], nums[i]);    // 堆顶和末尾元素交换
        heapify(nums, 0, i);       // 调整剩余堆
    }
}
```

自定义排序的几种方式

```
struct CustomCompare {
    bool operator()(int a, int b) const {
        return a > b;  // 降序
    }
};
```

```
template <typename T>
struct CustomCompare {
    bool operator()(const T& a, const T& b) const {
        return a > b;  // 降序
    }
};
```

```
// 使用 lambda 表达式自定义排序
    auto compare = [](int a, int b) { return a > b; };
    std::set<int, decltype(compare)> mySet(compare);
```

为什么需要 `decltype`？

STL 的容器（如 `std::set`）要求指定比较器的类型作为模板参数。在使用 lambda 表达式时，lambda 是一种匿名类型，无法直接以名字引用，因此需要 `decltype` 来推导其类型。

替代方案是 `std::function`

```
std::function<bool(int, int)> compare = [](int a, int b) { return a > b; };
std::set<int, std::function<bool(int, int)>> mySet(compare);
```

reserve和reverse不同