容器

vector

(1) Vector的底层原理

- Vector底层是一个动态数组，由三个主要部分组成：`start`、`finish`和`end_of_storage`。这里的`start`、`finish`和`end_of_storage`是指针，而不是迭代器。`start`指向数组的开始，`finish`指向最后一个实际元素之后的位置，而`end_of_storage`指向分配的内存末尾。
- 当空间不足以容纳更多数据（如执行`vec.push_back(val)`时）时，`vector`会分配一个更大的内存块（通常为当前大小的1.5倍或2倍），然后将现有元素复制到新内存中，并释放旧内存。这是vector的内存增长机制。
- 使用`vec.clear()`清空数据时，虽然移除了所有元素，但内存空间并未被释放，只是将`finish`指针移动到`start`处。
- 任何导致内存重新分配的操作，如添加或移除元素导致的空间不足，都会使指向原始内存的所有指针、引用和迭代器失效。这是因为内存重新分配通常涉及到数据的移动，原有的内存地址不再有效。

(2) Vector中的reserve和resize的区别

- `reserve`直接扩充到确定的大小，减少多次开辟和释放空间的问题，优化push_back，提高效率，并减少数据拷贝。Reserve只保证vector的空间大小（capacity）至少达到指定大小n。reserve()只有一个参数。
- `resize`改变有效空间的大小，也可改变默认值。Capacity随之改变。resize()可以有多个参数。

(3) Vector中的size和capacity的区别

- `size`表示当前vector中元素的数量（finish - start）。
- `capacity`表示分配的内存中可容纳的元素数量（end_of_storage - start）。

(4) Vector的元素类型可以是引用吗?

- Vector要求连续的对象排列，引用并非对象且没有实际地址，所以vector的元素类型不能是引用。

(5) Vector迭代器失效的情况

- 插入元素导致内存重新分配，使指向原内存的迭代器失效。
- 删除容器中元素后，迭代器失效。`erase`方法返回下一个有效的迭代器，因此删除元素时需要`it = vec.erase(it);`。

(6) 正确释放Vector的内存(`clear()`, `swap()`, `shrink_to_fit()`)

- `vec.clear()`：清空内容，但不释放内存。
- `vector<int>().swap(vec)`：清空内容并释放内存，得到一个新的vector。
- `vec.shrink_to_fit()`：降低容器的capacity以匹配size。
- `vec.clear(); vec.shrink_to_fit();`：清空内容并释放内存。

(7) Vector扩容为什么要以1.5倍或者2倍扩容?

- 考虑堆空间浪费，增长倍数不宜过大。两种广泛使用的扩容方式：2倍和1.5倍。
- 2倍扩容可能导致之前分配的内存无法再使用，所以最好设置倍增长因子在(1, 2)之间。

(8) Vector的常用函数

```
vector<int> vec(10,100);          // 创建10个元素，每个元素值为100。
vec.resize(r,vector<int>(c,0)); // 二维数组初始化。
reverse(vec.begin(),vec.end()); // 元素翻转。
sort(vec.begin(),vec.end());     // 排序，默认升序。
vec.push_back(val);               // 尾部插入数字。
vec.size();                       // 获取向量大小。
find(vec.begin(),vec.end(),1);   // 查找元素。
iterator = vec.erase(iterator); // 删除元素。
```

list

(1) List的底层原理

- List的底层是一个**双向链表**。它由结点组成，每个结点存储数据。链表中的结点在内存中的地址不一定连续。
- 每次插入或删除元素时，List会配置或释放一个结点的空间。
- List不支持随机存取，也就是说，它不能像数组那样直接通过索引访问元素。如果应用场景需要频繁地插入和删除数据，而对随机存取的需求不高，List是一个好选择。

(2) List的常用函数

- `list.push_back(elem)`：在链表尾部添加一个元素。
- `list.pop_back()`：删除尾部的元素。
- `list.push_front(elem)`：在链表头部插入一个元素。
- `list.pop_front()`：删除头部的元素。
- `list.size()`：返回链表中实际数据的个数。
- `list.sort()`：对链表进行排序，默认是升序排序。
- `list.unique()`：移除数值相同的连续元素，只保留一个。
- `list.back()`：获取链表尾部元素的引用。
- `list.erase(iterator)`：删除指定位置的元素，参数是一个指向链表元素的迭代器。该函数返回删除元素后的下一个元素的迭代器。

(3) list的排序

`std::sort` 通常不能直接用于对 `std::list` 进行排序，原因在于 `std::sort` 要求随机访问迭代器，而 `std::list` 提供的是双向迭代器。由于 `list` 的底层实现是双向链表，所以它不支持快速随机访问，这正是 `std::sort` 所需的。

相反，`std::list` 有自己的成员函数 `sort()` 用于排序。这个函数是专门为适应 `list` 的链表结构而设计的，能够有效地在链表上进行排序操作。`list::sort()` 通常使用的是**合并排序算法（merge sort）**，这种算法适合链式存储结构，因为它不依赖于随机访问。

总结一下主要区别：

1. **迭代器类型要求**：`std::sort` 需要随机访问迭代器，而 `list::sort()` 使用的是双向迭代器。

2. **适用性**：`std::sort` 适用于像 `vector`、`deque` 和数组这样提供随机访问的容器。`list::sort()` 专门设计用于 `list`。
3. **性能**：由于 `list::sort()` 是为 `list` 的数据结构量身定做的，因此它在对 `list` 进行排序时通常会比使用 `std::sort` 更有效率。

deque

（1）Deque的底层原理

- Deque（双端队列）是一个双向开口的连续线性空间。它允许在头尾两端进行元素的插入和删除操作，这些操作都具有理想的时间复杂度。
- Deque的内部实现通常是**一系列固定大小的数组**，这些数组通过特定的内部机制连接起来。这种结构使得它在两端的插入和删除操作都能保持高效，同时也能较好地支持随机访问。

（2）Vector、List与Deque的使用场景

- **Vector**：适用于元素数量相对固定、对象简单、随机访问频繁的场景。Vector在非尾部插入或删除数据时效率较低。一般情况下，优先选择Vector而不是Deque，因为Vector的迭代器结构比Deque简单。
- **List**：适用于对象较大、对象数量变化频繁、插入和删除操作频繁的场景，比如在写多读少的情况下。List不支持随机存取。
- **Deque**：当需要在序列的首尾两端进行插入或删除操作时，应选择Deque。

（3）Deque的常用函数

- `deque.push_back(elem)`：在尾部添加一个元素。
- `deque.pop_back()`：删除尾部元素。
- `deque.push_front(elem)`：在头部插入一个元素。
- `deque.pop_front()`：删除头部元素。
- `deque.size()`：返回容器中实际数据的个数。
- `deque.at(idx)`：返回索引`idx`所指的数据。如果`idx`越界，会抛出`out_of_range`异常。

priority_queue

（1）Priority Queue的底层原理

- `priority_queue` 是一个优先队列，其底层实现通常基于堆结构。在优先队列中，队首元素（即顶部元素）总是具有最高优先级的元素。
- 根据优先级的定义，优先队列可以是最大堆（默认情况，优先级最高的是最大元素）或最小堆。

（2）Priority Queue的常用函数

- 创建最小堆：`priority_queue<int, vector<int>, greater<int>> pq;`
- 创建最大堆：`priority_queue<int, vector<int>, less<int>> pq;`
- `pq.empty()`：如果队列为空，返回`true`。
- `pq.pop()`：删除队顶元素。
- `pq.push(val)`：向队列中添加一个元素。
- `pq.size()`：返回队列中的元素个数。
- `pq.top()`：返回优先级最高的元素。

Lambda函数自定义排序示例

假设我们想要一个优先队列，其中的元素按照自定义的规则排序。例如，如果我们想根据元素的绝对值来排列元素，我们可以这样做：

```
#include <iostream>
#include <queue>
#include <functional>  // For std::function
#include <cmath>       // For std::abs

int main() {
    // 使用lambda表达式创建一个自定义比较器
    auto compare = [](int a, int b) {
        return std::abs(a) > std::abs(b);
    };

    // 创建一个按照绝对值排序的优先队列
    std::priority_queue<int, std::vector<int>, decltype(compare)> pq(compare);

    // 添加一些元素
    pq.push(-5);
    pq.push(3);
    pq.push(-2);
    pq.push(6);

    // 打印并移除队列元素
    while (!pq.empty()) {
        std::cout << pq.top() << ' ';
        pq.pop();
    }

    return 0;
}
```

Map、Set、Multiset、Multimap

(1) Map、Set、Multiset、Multimap的底层原理

- 这些容器的底层实现都是基于**红黑树**，这是一种自平衡二叉搜索树。
- 红黑树的特性：
    1. 每个结点是红色或黑色。
    2. 根结点是黑色。
    3. 所有叶子结点（空结点）是黑色。
    4. 每个红色结点的两个子结点都是黑色（从每个叶子到根的所有路径上不能有两个连续的红色结点）。
    5. 从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点。
- 在STL的`map`容器中，`count`方法和`find`方法都可以用来判断一个键是否存在。`mp.count(key) > 0`表示键出现的次数（结果为0或1），而`mp.find(key) != mp.end()`则表示键存在。

(2) Map、Set、Multiset、Multimap的特点

- **Set** 和 **Multiset** 根据特定的排序准则自动对元素进行排序。Set中元素不重复，而Multiset允许重复元素。
- **Map** 和 **Multimap** 将键和值组成的对（pair）作为元素，根据键的排序准则自动排序。Map中键不重复，而Multimap允许重复键。
- Map和Set的增删查改操作的速度通常是O(log n)，这是相对高效的。

(3) 为什么Map和Set的插入删除效率高，且Iterator稳定?

- 因为它们存储的是结点而非仅仅是元素，不涉及内存拷贝和移动。
- 插入操作仅涉及结点指针的调整，结点本身的内存位置不变。因此，指向结点的迭代器（即Iterator）在插入操作后仍然有效。

(4) 为什么Map和Set没有像Vector那样的Reserve函数?

- 在Map和Set内部，存储的不再是单纯的元素，而是包含元素的结点。这意味着Map和Set使用的内存分配器（Alloc）并不是在声明时从参数中传入的Alloc，而是用于分配结点的。

(5) Map、Set、Multiset、Multimap的常用函数

```
auto it = map.begin();          // 返回指向容器起始位置的迭代器
auto it = map.end();            // 返回指向容器末尾位置的迭代器
bool empty = map.empty();       // 若容器为空，返回true，否则返回false
auto it = map.find(k);          // 查找键为k的元素，返回其迭代器
int size = map.size();          // 返回map中元素的数量
map.insert({int, string});      // 插入元素

for (auto itor = map.begin(); itor != map.end();) {
    if (itor->second == "target") {
        map.erase(itor++); // erase之后，将迭代器移到其后继
    } else {
        ++itor;
    }
}
```

unordered_map、unordered_set

(1) Unordered Map、Unordered Set的底层原理

- Unordered map和unordered set的底层实现是基于**哈希表**（防冗余设计）。
- 哈希表通过哈希函数将每个元素的键映射到一个数组索引（也称为哈希值或桶）。
- 哈希表的优点在于将数据的存储和查找时间大幅降低，理想情况下时间复杂度为O(1)；但代价是相对较高的内存消耗。
- 由于哈希函数可能会为不同的键产生相同的哈希值（即冲突），通常使用**拉链法**等策略来解决冲突，即在相同哈希值的位置存储一个链表。

(2) Unordered Map 与 Map的区别及使用场景

- **构造函数**：Unordered map需要哈希函数和等于函数，而map只需要比较函数（通常是小于函数）。
- **存储结构**：Unordered map采用哈希表存储，而map一般采用红黑树实现。

- **查找速度**：Unordered map的查找速度通常快于map，接近常数级别；而map的查找速度是O(log n)级别。
- **内存消耗**：Unordered map可能会消耗更多内存，特别是在对象数量众多时。
- **构造速度**：Unordered map的构造速度相对较慢。

总结：如果考虑查找效率，特别是在元素数量较多时，可以考虑使用unordered_map。但如果对内存使用有严格限制，要注意unordered_map可能导致较高的内存消耗。

（3）Unordered Map、Unordered Set的常用函数

- `unordered_map.begin()`：返回指向容器起始位置的迭代器。
- `unordered_map.end()`：返回指向容器末尾位置的迭代器。
- `unordered_map.cbegin()`：返回指向容器起始位置的常迭代器。
- `unordered_map.cend()`：返回指向容器末尾位置的常迭代器。
- `unordered_map.size()`：返回容器中有效元素的个数。
- `unordered_map.insert(key)`：插入元素。
- `unordered_map.find(key)`：查找元素，返回指向该元素的迭代器。
- `unordered_map.count(key)`：返回匹配给定键的元素个数。

迭代器

1. 迭代器的底层原理

- 迭代器是容器与算法之间的重要桥梁，允许在不了解容器内部原理的情况下遍历容器。
- 底层实现包括两个关键部分：萃取技术（traits）和模板偏特化。
- **萃取技术（traits）**：用于类型推导，允许根据不同类型执行不同处理流程。例如，对于`vector`，traits推导其迭代器为随机访问迭代器，而对于`list`则为双向迭代器。
- **模板偏特化**：用于推导参数。如果定义了多个类型，需要为这些自定义类型编写特化版本，否则只能判断它们是内置类型，不能确定具体类型。

```
template <typename T>
struct TraitsHelper {
     static const bool isPointer = false;
};
template <typename T>
struct TraitsHelper<T*> {
     static const bool isPointer = true;
};
```

2. 理解traits的例子

- 需要在T为`int`时，`Compute`方法参数和返回类型都为`int`；当T为`float`时，参数为`float`，返回类型为`int`。

```
template <typename T>
class Test {
public:
     TraitsHelper<T>::ret_type Compute(TraitsHelper<T>::par_type d);
private:
     T mData;
};
```

```cpp
template <typename T>
struct TraitsHelper {
     typedef T ret_type;
     typedef T par_type;
};

template <>
struct TraitsHelper<int> {
     typedef int ret_type;
     typedef int par_type;
};

template <>
struct TraitsHelper<float> {
     typedef float ret_type;
     typedef int par_type;
};
```

3. 迭代器的种类

- **输入迭代器**：只读迭代器，每个位置只读取一次。
- **输出迭代器**：只写迭代器，每个位置只写入一次。
- **前向迭代器**：结合输入和输出迭代器能力，可重复读写同一位置，只能向前移动。
- **双向迭代器**：类似前向迭代器，可向前和向后移动。
- **随机访问迭代器**：具有双向迭代器全部功能，提供迭代器算术，如跳跃至任意位置。

4. 迭代器失效的问题

- **插入操作**：
  - 对于`vector`和`string`，如果容器内存被重新分配，则所有迭代器、指针、引用失效。如果未重新分配，插入点之前的迭代器有效，之后的失效。
  - 对于`deque`，除首尾外的插入点会使所有迭代器、指针、引用失效；首尾插入时，迭代器失效，但引用和指针有效。
  - 对于`list`和`forward_list`，所有迭代器、指针和引用都保持有效。
- **删除操作**：
  - 对于`vector`和`string`，删除点之前的迭代器、指针、引用有效；末尾迭代器总是失效。
  - 对于`deque`，除首尾外的删除点会使所有迭代器、指针、引用失效；首尾删除时，末尾迭代器失效，其他迭代器、指针、引用有效。
  - 对于`list`和`forward_list`，所有迭代器、指针和引用保持有效。
  - 对于关联容器如`map`，被删除元素的迭代器失效，不应再使用。

5. 为什么有指针还要迭代器

1. **抽象级别**：迭代器提供了一个更高层次的抽象。与指针相比，迭代器隐藏了容器的内部实现细节。例如，你可以使用相同的方式（通过迭代器）来遍历`vector`、`list`或`map`，尽管它们内部实现完全不同。
2. **通用性**：迭代器为不同类型的容器提供了统一的访问方式。这使得算法可以独立于容器类型工作，增加了代码的复用性。例如，标准库算法`std::sort`可以用于任何提供随机访问迭代器的容器，如`vector`和`deque`。
3. **安全性**：与裸指针相比，某些类型的迭代器提供了额外的安全性和错误检查。例如，STL迭代器可以进行边界检查，而普通指针则不能。

4. **功能丰富**：迭代器不仅仅是访问元素的手段，它还定义了容器和元素之间的关系。例如，双向迭代器和随机访问迭代器支持比单纯的指针更复杂的操作，如双向遍历和随机访问。
5. **与容器的紧密结合**：迭代器通常是容器的一部分，设计为与其容器紧密协作。这意味着当容器的状态改变时（如元素添加或删除），迭代器可以相应地调整自己的行为。

STL容器的线程安全性

1. 线程安全的情况

- **多个读取者**：当多个线程仅读取同一容器的内容时，操作是线程安全的。在这种情况下，容器的内容不会被更改，因此并发读取不会导致问题。重要的是要确保在读取期间没有线程在写入或修改容器。
- **不同容器的多写入者**：多线程可以安全地同时写入不同的容器。因为每个容器是独立的，线程间的操作不会相互干扰。

2. 线程不安全的情况

- **同一容器的读写或多写操作**：当多个线程试图同时读取和写入同一个容器，或者有多个线程同时写入时，容器的行为是线程不安全的。这可能导致数据竞争和非预期的结果。
- **成员函数调用期间的锁定**：为了确保线程安全，在调用容器的任何成员函数期间，应该锁定该容器。这是因为大多数STL容器的成员函数不是线程安全的。
- **迭代器的生存期内的锁定**：在使用容器返回的迭代器（如通过`begin`或`end`方法获得）期间，也应锁定容器。迭代器可能会失效，特别是在容器被修改时。
- **算法执行期间的锁定**：在对容器进行操作的任何算法执行期间，也应锁定该容器。这是因为算法可能在容器上进行多步操作，而在这些操作期间容器的状态可能会因其他线程的干扰而改变。