vector

`std::vector` 是 C++ 标准库中的一个动态数组容器，提供了在运行时可变大小的数组。它是一个模板类，允许存储任意类型的元素。以下是对 `std::vector` 的详细解释：

1. 头文件包含

在使用 `std::vector` 之前，需要包含 `<vector>` 头文件。

```
#include <vector>
```

2. 创建和初始化

可以使用多种方式创建和初始化 `std::vector`。

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

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

    // 使用初始化列表创建 vector
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // 创建指定大小并初始化的 vector
    std::vector<int> sizedVector(5, 0);  // 大小为5，初始值为0

    // 复制另一个 vector
    std::vector<int> copiedVector(numbers);

    // 创建包含 n 个相同元素的 vector
    std::vector<int> repeatedVector(3, 7);  // 包含3个值为7的元
素

    // 通过迭代器初始化
    std::vector<int> iteratorVector(numbers.begin(),
numbers.end());
```

```
    // 输出 vector 内容
   for (auto it = numbers.begin(); it != numbers.end(); ++it)
{
        std::cout << *it << " ";
    }

    return 0;
}
```

3. 元素访问

通过索引访问元素或者使用迭代器。

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

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

    // 通过索引访问元素
    std::cout << "Element at index 2: " << numbers[2] <<
std::endl;

    // 使用迭代器访问元素
    for (auto it = numbers.begin(); it != numbers.end(); ++it)
{
        std::cout << *it << " ";
    }

    return 0;
}
```

4. 插入和删除元素

可以在 vector 的任意位置插入和删除元素。

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

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

```
    // 插入元素
    numbers.insert(numbers.begin() + 2, 10);  // 在索引2处插入值为10的元素

    // 删除元素
    numbers.erase(numbers.begin() + 3);  // 删除索引3处的元素

    // 输出修改后的 vector
    for (const auto& element : numbers) {
        std::cout << element << " ";
    }

    return 0;
}
```

5. 其他常用操作

`std::vector` 提供了许多其他常用的操作，比如获取 vector 大小、检查是否为空、清空等。

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

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

    // 获取 vector 大小
    std::cout << "Size of vector: " << numbers.size() << std::endl;

    // 检查 vector 是否为空
    std::cout << "Is vector empty? " << (numbers.empty() ? "Yes" : "No") << std::endl;

    // 清空 vector
    numbers.clear();

    return 0;
}
```

6. 动态调整大小

`std::vector` 允许在运行时动态调整大小。

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

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

    // 改变 vector 大小
    numbers.resize(8);  // 将 vector 大小调整为8，多余的元素使用默认值填充

    // 输出修改后的 vector
    for (const auto& element : numbers) {
        std::cout << element << " ";
    }

    return 0;
}
```

7. 性能注意事项

- `std::vector` 提供常数时间的随机访问，但在中间插入或删除元素可能会导致较大开销。
- 当需要在序列的中间频繁插入或删除元素时，可以考虑使用 `std::list` 或 `std::deque`。

`std::vector` 是 C++ 中非常常用的容器之一，它的灵活性和性能使其成为许多应用程序的首选。

pair

`std::pair` 是 C++ 标准库中的一个模板类，用于将两个值组合成一个对。它提供了一种简单的方法来存储和访问两个相关的值。以下是对 `std::pair` 的详细解释：

1. 头文件包含

在使用 `std::pair` 之前，需要包含 `<utility>` 头文件。

```
#include <utility>
```

2. 创建和初始化

可以使用多种方式创建和初始化 `std::pair`。

```
#include <utility>
#include <iostream>

int main() {
    // 创建一个默认构造的 pair
    std::pair<int, double> myPair1;

    // 使用构造函数初始化 pair
    std::pair<int, double> myPair2(42, 3.14);

    // 使用 make_pair 函数创建 pair
    auto myPair3 = std::make_pair(7, 2.718);

    // 创建一个包含其他 pair 的 pair
    std::pair<std::pair<int, int>, std::pair<double, double>>
nestedPair;
    nestedPair.first = std::make_pair(1, 2);
    nestedPair.second = std::make_pair(3.0, 4.0);

    // 输出 pair 的值
    std::cout << "myPair2: " << myPair2.first << ", " <<
myPair2.second << std::endl;
    std::cout << "myPair3: " << myPair3.first << ", " <<
myPair3.second << std::endl;

    return 0;
}
```

3. 元素访问

可以通过 `first` 和 `second` 成员访问 `std::pair` 的元素。

```
#include <utility>
#include <iostream>

int main() {
    std::pair<int, double> myPair(42, 3.14);

    // 访问 pair 的第一个元素
    int firstElement = myPair.first;

    // 访问 pair 的第二个元素
    double secondElement = myPair.second;

    // 输出元素的值
    std::cout << "First element: " << firstElement <<
std::endl;
    std::cout << "Second element: " << secondElement <<
std::endl;

    return 0;
}
```

4. 比较操作

`std::pair` 支持比较操作，可以使用关系运算符比较两个 pair。

```
#include <utility>
#include <iostream>

int main() {
    std::pair<int, double> pair1(42, 3.14);
    std::pair<int, double> pair2(7, 2.718);

    // 比较两个 pair
    if (pair1 == pair2) {
        std::cout << "pair1 is equal to pair2." << std::endl;
    } else {
        std::cout << "pair1 is not equal to pair2." <<
std::endl;
```

```
    }

    return 0;
}
```

5. 在容器中使用

`std::pair` 可以很方便地在容器中存储键值对。

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

int main() {
    // 使用 std::map 存储键值对
    std::map<std::string, int> myMap;
    myMap["one"] = 1;
    myMap["two"] = 2;

    // 使用 std::pair 在容器中存储键值对
    std::pair<std::string, int> keyValue("three", 3);
    myMap.insert(keyValue);

    // 输出键值对
    for (const auto& pair : myMap) {
        std::cout << pair.first << ": " << pair.second <<
std::endl;
    }

    return 0;
}
```

`std::pair` 是 C++ 中用于存储两个相关值的一个简便的工具。当需要将两个值组合在一起时，使用 `std::pair` 是一种方便和直观的方式。

queue

`std::queue` 是 C++ 标准库中的一个容器适配器，提供了队列（先进先出，FIFO）的操作。它基于其他底层容器（默认是 `std::deque`）实现。以下是对 `std::queue` 的详细解释：

1. 头文件包含

在使用 `std::queue` 之前，需要包含 `<queue>` 头文件。

```
#include <queue>
```

2. 创建和初始化

可以使用构造函数和赋值运算符初始化队列。

```
#include <queue>

int main() {
    // 默认构造一个空队列
    std::queue<int> myQueue1;

    // 使用已有的队列初始化新队列
    std::queue<int> myQueue2(myQueue1);

    // 使用赋值运算符
    std::queue<int> myQueue3 = myQueue2;

    return 0;
}
```

3. 元素操作

`std::queue` 提供了用于操作队列的成员函数，其中最常用的是 `push()`、`pop()` 和 `front()`。

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

int main() {
    std::queue<int> myQueue;

    // 向队尾添加元素
    myQueue.push(42);
    myQueue.push(7);
    myQueue.push(14);
```

```
    // 访问队首元素
    int frontElement = myQueue.front();
    std::cout << "Front element: " << frontElement <<
std::endl;

    // 弹出队首元素
    myQueue.pop();

    // 输出队列大小
    std::cout << "Queue size: " << myQueue.size() <<
std::endl;

    return 0;
}
```

4. 检查队列是否为空

可以使用 `empty()` 函数检查队列是否为空。

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

int main() {
    std::queue<int> myQueue;

    // 检查队列是否为空
    if (myQueue.empty()) {
        std::cout << "Queue is empty." << std::endl;
    } else {
        std::cout << "Queue is not empty." << std::endl;
    }

    return 0;
}
```

5. 底层容器选择

`std::queue` 的底层容器可以通过模板参数进行选择，默认是 `std::deque`。可以选择 `std::list` 或其他适当的容器。

```cpp
#include <queue>
#include <list>

int main() {
    // 使用 std::list 作为底层容器
    std::queue<int, std::list<int>> myQueue;

    return 0;
}
```

6. 性能考虑

- `std::queue` 提供了常数时间的插入和删除操作。
- 如果需要快速随机访问元素，但仍希望保持队列特性，可以考虑使用 `std::deque` 作为底层容器。

`std::queue` 是一个简单而有用的容器适配器，适合在需要遵循先进先出原则的情况下使用。

stack

`std::stack` 是 C++ 标准库中的一个容器适配器，提供了堆栈（后进先出，LIFO）的操作。它基于其他底层容器（默认是 `std::deque`）实现。以下是对 `std::stack` 的详细解释：

1. 头文件包含

在使用 `std::stack` 之前，需要包含 `<stack>` 头文件。

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

2. 创建和初始化

可以使用构造函数和赋值运算符初始化堆栈。

```cpp
#include <stack>

int main() {
    // 默认构造一个空堆栈
    std::stack<int> myStack1;
```

```
    // 使用已有的堆栈初始化新堆栈
    std::stack<int> myStack2(myStack1);

    // 使用赋值运算符
    std::stack<int> myStack3 = myStack2;

    return 0;
}
```

3. 元素操作

`std::stack` 提供了用于操作堆栈的成员函数，其中最常用的是 `push()`、`pop()` 和 `top()`。

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

int main() {
    std::stack<int> myStack;

    // 向堆栈顶部添加元素
    myStack.push(42);
    myStack.push(7);
    myStack.push(14);

    // 访问堆栈顶部元素
    int topElement = myStack.top();
    std::cout << "Top element: " << topElement << std::endl;

    // 弹出堆栈顶部元素
    myStack.pop();

    // 输出堆栈大小
    std::cout << "Stack size: " << myStack.size() <<
std::endl;

    return 0;
}
```

4. 检查堆栈是否为空

可以使用 `empty()` 函数检查堆栈是否为空。

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

int main() {
    std::stack<int> myStack;

    // 检查堆栈是否为空
    if (myStack.empty()) {
        std::cout << "Stack is empty." << std::endl;
    } else {
        std::cout << "Stack is not empty." << std::endl;
    }

    return 0;
}
```

5. 底层容器选择

`std::stack` 的底层容器可以通过模板参数进行选择，默认是 `std::deque`。可以选择 `std::list` 或其他适当的容器。

```
#include <stack>
#include <list>

int main() {
    // 使用 std::list 作为底层容器
    std::stack<int, std::list<int>> myStack;

    return 0;
}
```

6. 性能考虑

- `std::stack` 提供了常数时间的插入和删除操作。
- 如果需要快速随机访问元素，但仍希望保持堆栈特性，可以考虑使用 `std::deque` 作为底层容器。

`std::stack` 是一个简单而有用的容器适配器，适合在需要遵循后进先出原则的情况下使用。

set

`std::set` 是 C++ 标准库中的一个关联容器，实现了红黑树（Red-Black Tree）数据结构。它提供了一种有序的、不允许重复元素的集合。以下是对 `std::set` 的详细解释：

1. 头文件包含

在使用 `std::set` 之前，需要包含 `<set>` 头文件。

```
#include <set>
```

2. 创建和初始化

可以使用多种方式创建和初始化 `std::set`。

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

int main() {
    // 默认构造一个空集合
    std::set<int> mySet1;

    // 使用已有的集合初始化新集合
    std::set<int> mySet2(mySet1);

    // 使用初始化列表创建集合
    std::set<int> mySet3 = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    // 创建指定范围的集合
    int arr[] = {1, 2, 3, 4, 5};
    std::set<int> mySet4(arr, arr + 5);

    // 输出集合内容
    for (const auto& element : mySet3) {
        std::cout << element << " ";
    }
```

```
    return 0;
}
```

3. 插入和删除元素

`std::set` 提供了插入和删除元素的成员函数，包括 `insert()`、`erase()` 和 `clear()`。

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

int main() {
    std::set<int> mySet;

    // 插入元素
    mySet.insert(42);
    mySet.insert(7);
    mySet.insert(14);

    // 删除元素
    mySet.erase(7);

    // 清空集合
    mySet.clear();

    return 0;
}
```

4. 查找元素

可以使用 `find()` 函数查找元素。

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

int main() {
    std::set<int> mySet = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    // 查找元素
    auto it = mySet.find(4);
    if (it != mySet.end()) {
        std::cout << "Element found: " << *it << std::endl;
```

```
    } else {
        std::cout << "Element not found." << std::endl;
    }

    return 0;
}
```

5. 访问元素

`std::set` 中的元素是有序的，可以通过迭代器访问元素。

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

int main() {
    std::set<int> mySet = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    // 使用迭代器访问元素
    for (auto it = mySet.begin(); it != mySet.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

6. 检查元素是否存在

可以使用 `count()` 或 `find()` 函数检查元素是否存在。

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

int main() {
    std::set<int> mySet = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    // 使用 count 函数检查元素是否存在
    if (mySet.count(4) > 0) {
        std::cout << "Element exists in the set." <<
std::endl;
    } else {
        std::cout << "Element does not exist in the set." <<
std::endl;
```

```
    }

    return 0;
}
```

7. 性能考虑

- `std::set` 提供了对数时间的插入、删除和查找操作。
- 由于底层使用红黑树实现，元素是有序的，因此插入和删除元素相对于其他容器可能稍慢。

8. 注意事项

- `std::set` 不允许重复的元素，如果需要允许重复，可以使用 `std::multiset`。

`std::set` 是一个非常有用的容器，特别适用于需要保持元素有序并且不允许重复的情况。

map

`std::map` 是 C++ 标准库中的一个关联容器，实现了红黑树（Red-Black Tree）数据结构。它提供了一种有序的、基于键值对的关联集合。以下是对 `std::map` 的详细解释：

1. 头文件包含

在使用 `std::map` 之前，需要包含 `<map>` 头文件。

```
#include <map>
```

2. 创建和初始化

可以使用多种方式创建和初始化 `std::map`。

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

int main() {
    // 默认构造一个空映射
```

```
    std::map<int, std::string> myMap1;

    // 使用已有的映射初始化新映射
    std::map<int, std::string> myMap2(myMap1);

    // 使用初始化列表创建映射
    std::map<int, std::string> myMap3 = {{1, "One"}, {2,
"Two"}, {3, "Three"}};

    // 创建指定范围的映射
    std::pair<int, std::string> arr[] = {{4, "Four"}, {5,
"Five"}};
    std::map<int, std::string> myMap4(arr, arr + 2);

    // 输出映射内容
    for (const auto& pair : myMap3) {
        std::cout << pair.first << ": " << pair.second <<
std::endl;
    }

    return 0;
}
```

3. 插入和删除元素

`std::map` 提供了插入和删除元素的成员函数，包括 `insert()`、`erase()` 和 `clear()`。

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

int main() {
    std::map<int, std::string> myMap;

    // 插入元素
    myMap.insert(std::make_pair(1, "One"));
    myMap[2] = "Two";
    myMap[3] = "Three";

    // 删除元素
    myMap.erase(2);
```

```
    // 清空映射
    myMap.clear();

    return 0;
}
```

4. 查找元素

可以使用 `find()` 函数查找元素。

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

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

    // 查找元素
    auto it = myMap.find(2);
    if (it != myMap.end()) {
        std::cout << "Element found: " << it->second <<
std::endl;
    } else {
        std::cout << "Element not found." << std::endl;
    }

    return 0;
}
```

5. 访问元素

`std::map` 中的元素是有序的，可以通过迭代器访问元素。

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

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

    // 使用迭代器访问元素
    for (auto it = myMap.begin(); it != myMap.end(); ++it) {
        std::cout << it->first << ": " << it->second <<
std::endl;
    }

    return 0;
}
```

6. 检查元素是否存在

可以使用 `count()` 或 `find()` 函数检查元素是否存在。

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

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

    // 使用 count 函数检查元素是否存在
    if (myMap.count(2) > 0) {
        std::cout << "Element exists in the map." <<
std::endl;
    } else {
        std::cout << "Element does not exist in the map." <<
std::endl;
    }

    return 0;
}
```

7. 性能考虑

- `std::map` 提供了对数时间的插入、删除和查找操作。
- 由于底层使用红黑树实现，元素是有序的，因此插入和删除元素相对于其他容器可能稍慢。

8. 注意事项

- `std::map` 中的键是唯一的，如果需要允许重复的键，可以使用 `std::multimap`。
- C++11 引入了 `std::unordered_map`，提供了基于哈希表的映射，对于查找操作可能更加高效。

`std::map` 是一个非常有用的容器，特别适用于需要保持键值对有序并且不允许重复的情况。