

# STL 中的容器类

主讲：陈笑沙

# 目录

- 12.1 容器类
- 12.2 线性结构容器
  - 12.2.1 vector
  - 12.2.2 deque
- 12.3 关联结构容器
  - 12.3.1 map
  - 12.3.2 set
  - 12.3.3 无序关联结构容器
- 12.4 总结

# 12.1 容器类

C++ 中有很多容器类

# 12.1 容器类

C++ 中有很多容器类

- `std::vector`

# 12.1 容器类

C++ 中有很多容器类

- `std::vector`
- `std::set`

# 12.1 容器类

C++ 中有很多容器类

- `std::vector`
- `std::set`
- `std::stack`

# 12.1 容器类

C++ 中有很多容器类

- `std::vector`
- `std::set`
- `std::stack`
- `std::queue`

# 12.1 容器类

C++ 中有很多容器类

- `std::vector`
- `std::set`
- `std::stack`
- `std::queue`
- ...



# 12.1 容器类

# 12.1 容器类

- 什么时候该用什么容器呢？

# 12.1 容器类

- 什么时候该用什么容器呢？
- 本质上是空间与时间的权衡

# 12.1 容器类

STL: Standard Template Library

全是模板类

# 12.1 容器类

主要包含：

# 12.1 容器类

主要包含：

- Containers

# 12.1 容器类

主要包含：

- Containers
- Iterators

# 12.1 容器类

主要包含：

- Containers
- Iterators
- Functors



# 12.1 容器类

主要包含：

- Containers
- Iterators
- Functors
- Algorithms

# 12.1 容器类

容器类可以分为：

# 12.1 容器类

容器类可以分为：

- 线性结构容器

# 12.1 容器类

容器类可以分为：

- 线性结构容器
- 关联结构容器

# 12.2 线性结构容器

## 12.2.1 VECTOR

`std::vector`

`#include <vector>`

# 12.2 线性结构容器

## 12.2.1 VECTOR

vector 是一个动态数组

```
1 std::vector<int> vec { 1, 2, 3, 4 };
2 vec.push_back(5);
3 vec.push_back(6);
4 vec[1] = 20;
5
6 for (size_t i = 0; i < vec.size(); i++) {
7     std::cout << vec[i] << " ";
8 }
```

# 12.2 线性结构容器

## 12.2.1 VECTOR

### 操作

### 代码

创建一个空的  
vector

```
std::vector<int> v;
```

创建一个有n个0  
的vector

```
std::vector<int>  
v(n);
```

创建一个有n个k  
的vector

```
std::vector<int> v(n,  
k);
```

# 12.2 线性结构容器

## 12.2.1 VECTOR

操作	代码
向后边添加k	<code>v.push_back(k);</code>
清空	<code>v.clear()</code>
判断是否为空	<code>v.empty()</code>



# 12.2 线性结构容器

## 12.2.1 VECTOR

### 操作

### 代码

获取第i个元素（无越界检查）

```
int k = v[i];
```

获取第i个元素（有越界检查）

```
int k =  
v.at(i);
```

替换第i个元素（无越界检查）

```
v.at(i) = k
```

## 操作

## 代码

---

替换第*i*个元素（有越界检查）

$v[i] = k$

# 12.2 线性结构容器

## 12.2.1 VECTOR

使用标准迭代方案：

```
1 for (size_t i = 0; i < vec.size(); i++) {  
2     std::cout << vec[i] << " ";  
3 }  
4  
5 for (const auto &elem : vec) {  
6     std::cout << elem << " ";  
7 }
```

# **12.2 线性结构容器**

## **12.2.1 VECTOR**

# 12.2 线性结构容器

## 12.2.1 VECTOR

- 尽量使用 `const auto& elem`, 而非 `auto elem`

# 12.2 线性结构容器

## 12.2.1 VECTOR

- 尽量使用 `const auto& elem`, 而非 `auto elem`
- 尽量使用 `at` 而非 `[]`

# 12.2 线性结构容器

## 12.2.1 VECTOR

零成本抽象：

# 12.2 线性结构容器

## 12.2.1 VECTOR

零成本抽象：

- 不必为没有使用的工具付出代价



# 12.2 线性结构容器

## 12.2.1 VECTOR

零成本抽象：

- 不必为没有使用的工具付出代价
- 使用工具后与纯手写的效率应当一致

# 12.2 线性结构容器

## 12.2.1 VECTOR

某些场景下，vector并不适用，例如：

```
1 void receivePrice(vector<double>& prices, double price)
2 {
3     prices.push_front(price); // 没有这个方法
4     if (prices.size() > 10000)
5         prices.pop_back();
6 }
```

# 12.2 线性结构容器

## 12.2.1 VECTOR

如何手动实现 `push_front`?

# 12.2 线性结构容器

## 12.2.2 DEQUE

`std::deque`

`#include <deque>`

# 12.2 线性结构容器

## 12.2.2 DEQUE

- `std::deque`
  - 是一个双端队列
  - 允许高效地在双端进行插入/删除操作

# 12.2 线性结构容器

## 12.2.2 DEQUE

```
1 void receivePrice(deque<double>& prices, double price)
2 {
3     prices.push_front(price); // Super fast
4     if (prices.size() > 10000)
5         prices.pop_back();
6 }
```

# 12.2 线性结构容器

## 12.2.2 DEQUE

deque与vector的接口完全相同，只是多了push\_front和pop\_front。

# 12.2 线性结构容器

## 12.2.2 DEQUE

deque 是如何实现的？



# 12.2 线性结构容器

## 12.2.2 DEQUE

deque 是如何实现的？

基本思路是多个小数组线性排列

## 12.3 关联结构容器

## 12.3 关联结构容器

关联结构容器通过键值管理元素

# 12.3 关联结构容器

## 12.3.1 MAP

`std::map`

`#include <map>`

# **12.3 关联结构容器**

## **12.3.1 MAP**

# 12.3 关联结构容器

## 12.3.1 MAP

- `std::map` 将键(Key)映射到值 (Value)

## 12.3 关联结构容器

### 12.3.1 MAP

- `std::map` 将键(Key)映射到值 (Value)
- `"ABC" → std::map<std::string, int> → 42`

# 12.3 关联结构容器

## 12.3.1 MAP

- `std::map` 将键(Key)映射到值(Value)
- `"ABC" → std::map<std::string, int> → 42`
- 可以认为是 python 的字典



# 12.3 关联结构容器

## 12.3.1 MAP

```
1 std::map<std::string, int> map {  
2     { "Chris", 2},  
3     { "CN", 42},  
4     { "Keith", 14 },  
5     { "Nick", 51 },  
6     { "Sean", 35 },  
7 };  
8  
9 int sean = map["Sean"]; // 35  
10 map["Chris"] = 31;
```

# 12.3 关联结构容器

## 12.3.1 MAP

操作	代码
创建一个空map	<code>std::map&lt;char, int&gt; m</code>
把键k映射为v	<code>m.insert(k, v);</code> <code>m[k] = v</code>
删除键k	<code>m.erase(k);</code>

# 12.3 关联结构容器

## 12.3.1 MAP

### 操作

### 代码

---

检测是否包含键k

```
if (m.count(k))
```

---

检测是否包含键  
k(C++20)

```
if  
(m.contains(k))
```

---

检测m是否为空

```
if (m.empty())
```

---

获取与更新

```
int i = m[k];  
m[k] = i;
```

# 12.3 关联结构容器

## 12.3.1 MAP

可以认为，`std::map<K, V>` 保存了一堆  
`std::pair<const K, V>`

# 12.3 关联结构容器

## 12.3.1 MAP

可以认为，`std::map<K, V>` 保存了一堆  
`std::pair<const K, V>`

为什么K是const?

# 12.3 关联结构容器

## 12.3.1 MAP

```
1 std::map<std::string, int> map;  
2  
3 for (auto kv : map) {  
4     // kv: std::pair<const std::string, int>  
5     std::string key = kv.first;  
6     int value = kv.second;  
7 }
```

# 12.3 关联结构容器

## 12.3.1 MAP

```
1 std::map<std::string, int> map;  
2  
3 for (const auto& [key, value] : map) {  
4     // key is const std::string&  
5     // value is const int&  
6 }
```

# 12.3 关联结构容器

## 12.3.1 MAP

map 是如何实现的？



# 12.3 关联结构容器

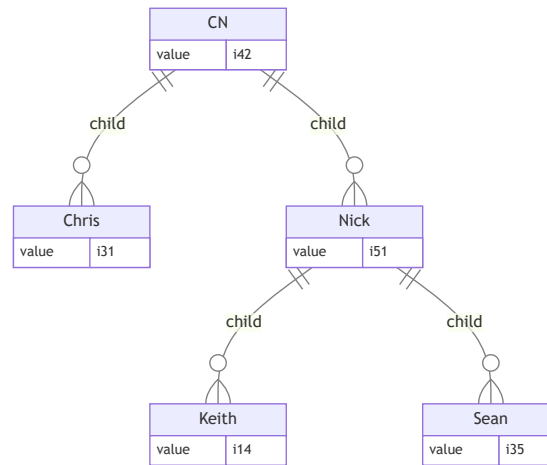
## 12.3.1 MAP

map 是如何实现的？

底层是用一棵**红黑树**实现的。

# 12.3 关联结构容器

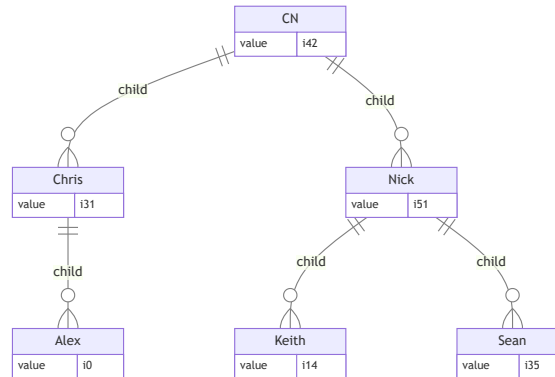
## 12.3.1 MAP



# 12.3 关联结构容器

## 12.3.1 MAP

那么 `map["Alex"]` 呢?



# 12.3 关联结构容器

## 12.3.1 MAP

`std::map<K, V>` 需要 K 支持 `operator<`

# 12.3 关联结构容器

## 12.3.1 MAP


`std::map<K, V>` 需要 K 支持 `operator<`

- `std::map<int, int> map1;`

# 12.3 关联结构容器

## 12.3.1 MAP


`std::map<K, V>` 需要 K 支持 `operator<`

- `std::map<int, int> map1;`
  -  OKAY - `int` has `operator<`

# 12.3 关联结构容器

## 12.3.1 MAP



`std::map<K, V>` 需要 K 支持 `operator<`

- `std::map<int, int> map1;`
  -  OKAY - `int` has `operator<`
- `std::map<std::ifstream, int> map2;`

# 12.3 关联结构容器

## 12.3.1 MAP

`std::map<K, V>` 需要 K 支持 `operator<`

- `std::map<int, int> map1;`
  -  OKAY - `int` has `operator<`
- `std::map<std::ifstream, int> map2;`
  -  ERROR - `std::ifstream` has no `operator<`



# 12.3 关联结构容器

## 12.3.2 SET

`std::set`

`#include <set>`

# 12.3 关联结构容器

## 12.3.2 SET

`set`

包含了一系列不同的元素（参考数学上的集合）

# 12.3 关联结构容器

## 12.3.2 SET

### 操作

### 代码

创建一个空的 set

```
std::set<char> s;
```

添加一个元素 k

```
s.insert(k);
```

删除元素 k

```
s.erase(k);
```

检查 k 是否在集合中

```
if (s.count(k))
```

检查 k 是否在集合中  
(c++20)

```
if  
(s.contains(k))
```

# 操作

# 代码

---

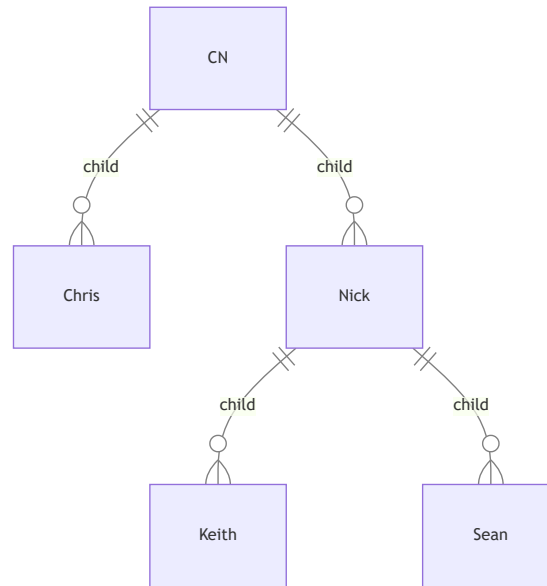
检查 s 是否为空

```
if(s.empty())
```

# 12.3 关联结构容器

## 12.3.2 SET

`std::set` 实际上是没有值，只有键的 `std::map`



## 12.3 关联结构容器

### 12.3.3 无序关联结构容器

有时候不需要排序，map 和 set 有更高效的选择。

## 12.3 关联结构容器

### 12.3.3 无序关联结构容器

`std::unordered_map` and `std::unordered_set`

```
#include <unordered_map>
```

```
#include <unordered_set>
```

## **12.3 关联结构容器**

### **12.3.3 无序关联结构容器**



## 12.3 关联结构容器

### 12.3.3 无序关联结构容器

- `std::unordered_map` 是 `map` 的优化版本

## 12.3 关联结构容器

### 12.3.3 无序关联结构容器

- `std::unordered_map` 是 `map` 的优化版本
- 接口基本相同

## 12.3 关联结构容器

### 12.3.3 无序关联结构容器

```
1 std::unordered_map<std::string, int> map {  
2     { "Chris", 2 },  
3     { "Nick", 51 },  
4     { "Sean", 35 },  
5 };  
6 int sean = map["Sean"]; // 35  
7 map["Chris"] = 31;
```

## 12.3 关联结构容器

### 12.3.3 无序关联结构容器

`unordered_map` 以哈希值对 `std::pair` 进行保存

`std::unordered_map<K, V>` 需要 `K` 有一个哈希函数

# 12.3 关联结构容器

## 12.3.3 无序关联结构容器

```
1 // unordered_map
2 template<
3     class Key,
4     class T,
5     class Hash = std::hash<Key>,
6     class KeyEqual = std::equal_to<Key>,
7     class Allocator = std::allocator<std::pair<const Key, T>>
8 > class unordered_map;
```

# 12.3 关联结构容器

## 12.3.3 无序关联结构容器

```
1 // unordered_map
2 template<
3     class Key,
4     class T,
5     class Hash = std::hash<Key>,
6     class KeyEqual = std::equal_to<Key>,
7     class Allocator = std::allocator<std::pair<const Key, T>>
8 > class unordered_map;
```

## **12.3 关联结构容器**

### **12.3.3 无序关联结构容器**

## 12.3 关联结构容器


### 12.3.3 无序关联结构容器

- `std::map<int, int> map1;`




## 12.3 关联结构容器

### 12.3.3 无序关联结构容器

- `std::map<int, int> map1;`
  -  OKAY - `int` is hashable



## 12.3 关联结构容器

### 12.3.3 无序关联结构容器

- `std::map<int, int> map1;`
  -  OKAY - `int` is hashable
- `std::map<std::ifstream, int> map2;`

## 12.3 关联结构容器

### 12.3.3 无序关联结构容器

- `std::map<int, int> map1;`
  -  OKAY - `int` is hashable
- `std::map<std::ifstream, int> map2;`
  -  ERROR - `std::ifstream` is not hashable

## 12.3 关联结构容器

### 12.3.3 无序关联结构容器

`unordered_set` 就是一个没有值、只有键的  
`unordered_map`

## 12.3 关联结构容器

## 12.3 关联结构容器

- `unordered_map` 通常比 `map` 快

## 12.3 关联结构容器

- `unordered_map` 通常比 `map` 快
- 但是 `unordered_map` 需要更多内存

## 12.3 关联结构容器

- `unordered_map` 通常比 `map` 快
- 但是 `unordered_map` 需要更多内存
- 如果键的类型不支持比较，只能选择 `unordered_map`



## 12.3 关联结构容器

- `unordered_map` 通常比 `map` 快
- 但是 `unordered_map` 需要更多内存
- 如果键的类型不支持比较，只能选择 `unordered_map`
- 如果不清楚用哪个，则可以先试试 `unordered_map`

## 12.4 总结

	第 i 个元素	查找	插入	删除
std::vector	非常快	慢	最后快 其他慢	最后快(pop_back) 其他慢
std::deque	快	慢	两端快 其他慢	两端快 其他慢
std::set	慢	快	快	快
std::map	慢	快	快	快
std::unordered_set	不支持	非常快	非常快	非常快
std::unordered_map	不支持	非常快	非常快	非常快

## 12.4 总结

一个例子：

Day 11 - Advent of Code 2024

有一串神奇的石头，每个石头上有一个数字。盯着它们时，石头不会发生变化，但是每次眨眼睛，这些石头就会发生一些神奇的变化：

- 标着 0 的石头会变成数字 1
- 如果数字个数为偶数，则分成两个石头，每个石头分别是一半的数字（前面的0消失）：1234变成12和34，1104变成11和4（而不是04）
- 如果以上两条都不符合，则数字变为原来的2024倍。

开始时石头为：[3028, 78, 973951, 5146801, 5, 0, 23533, 857]

眨25次眼后，有多少个石头？

眨75次眼后，有多少个石头？

# 12.4 总结

一个例子：

Day 11 - Advent of Code 2024

`example/lec12/magicstone`