跳转STL主目录

6. map

6.1 介绍

6.1.1 简介

映射类似于函数的对应关系，每个 x 对应一个 y ，而 map 是每个键对应一个值。这和python的字典类型非常相似。

容器中的每个存储对为一个 键值对 ，包含两个元素（键和值）。

```
// 头文件
#include<map>
```

6.1.2 初始化

```
//初始化定义
map<string, string> mp;
map<string, int> mp;
map<int, node> mp;//node是结构体类型
```

> map特性：map会按照键的顺序从小到大自动排序，键的类型必须可以 比较大小

6.2 函数方法

6.2.1 函数方法

代码	含义	
mp.find(key)	返回键为key的映射的迭代器 注意：用find函数来定位数据出现位置，它返回一个迭代器。当数据存在时，返回数据所在位置的迭代器，数据不存在时，返回*mp.end*()	C
mp.erase(it)	删除迭代器对应的键和值	C
mp.erase(key)	根据映射的键删除键和值	C
mp.erase(first,last)	删除左闭右开区间迭代器对应的键和值	C
mp.size()	返回映射的对数	
mp.clear()	清空map中的所有元素	
mp.insert()	插入元素，插入时要构造键值对	C
mp.empty()	如果map为空，返回true，否则返回false	
mp.begin()	返回指向map第一个元素的迭代器（地址）	
mp.end()	返回指向map尾部的迭代器（最后一个元素的**下一个**地址）	
mp.rbegin()	返回指向map最后一个元素的迭代器（地址）	
mp.rend()	返回指向map第一个元素前面(上一个）的逆向迭代器（地址）	

代码	含义	
mp.count(key)	查看元素是否存在，因为map中键是唯一的，所以存在返回1，不存在返回0	C
mp.lower_bound()	返回一个迭代器，指向键值>= **key**的第一个元素	
mp.upper_bound()	返回一个迭代器，指向键值> key的第一个元素	

6.2.2 注意情况

下面说明部分函数方法的注意点

- 注意点一：
  查找元素是否存在时，可以使用 ① `mp.find()` ② `mp.count()` ③ `mp[key]`
  但是第三种情况，如果不存在对应的 `key` 时，会自动创建一个键值对（产生一个额外的键值对空间）
  所以为了不增加额外的空间负担，最好使用前两种方法。

6.2.3 迭代器进行正反向遍历

- `mp.begin()` 和 `mp.end()` 用法：

用于正向遍历map

```
map<int,int> mp;
mp[1] = 2;
mp[2] = 3;
mp[3] = 4;
auto it = mp.begin();
while(it != mp.end()) {
        cout << it->first << " " << it->second << "\n";
        it ++;
}
```

结果：

```
1 2
2 3
3 4
```

- `mp.rbegin()` 和 `mp.rend()`

用于逆向遍历map

```
map<int,int> mp;
mp[1] = 2;
mp[2] = 3;
mp[3] = 4;
auto it = mp.rbegin();
while(it != mp.rend()) {
        cout << it->first << " " << it->second << "\n";
        it ++;
}
```

结果：

```
3 4
2 3
1 2
```

6.2.4 二分查找

二分查找 `lower_bound()` `upper_bound()`

> map的二分查找以第一个元素（即键为准），对**键**进行二分查找
> 返回值为map迭代器类型

```cpp
#include<bits/stdc++.h>
using namespace std;

int main() {
        map<int, int> m{{1, 2}, {2, 2}, {1, 2}, {8, 2}, {6, 2}};//有序
        map<int, int>::iterator it1 = m.lower_bound(2);
        cout << it1->first << "\n";//it1->first=2
        map<int, int>::iterator it2 = m.upper_bound(2);
        cout << it2->first << "\n";//it2->first=6
        return 0;
}

```

6.3 添加元素

```cpp
//先声明
map<string, string> mp;
```

- **方式一:**

```cpp
mp["学习"] = "看书";
mp["玩耍"] = "打游戏";
```

- **方式二：插入元素构造键值对**

```cpp
mp.insert(make_pair("vegetable","蔬菜"));
```

- **方式三:**

```cpp
mp.insert(pair<string,string>("fruit","水果"));
```

- **方式四:**

```cpp
mp.insert({"hahaha","wawawa"});
```

6.4 访问元素

6.4.1 下标访问

(大部分情况用于访问单个元素)

```
mp["菜哇菜"] = "强哇强";
cout << mp["菜哇菜"] << "\n";//只是简写的一个例子，程序并不完整
```

6.4.2 遍历访问

- 方式一：迭代器访问

```
map<string,string>::iterator it;
for(it = mp.begin(); it != mp.end(); it++) {
        //      键                  值
        // it是结构体指针访问所以要用 -> 访问
        cout << it->first << " " << it->second << "\n";
        //*it是结构体变量 访问要用 . 访问
        //cout<<(*it).first<<" "<<(*it).second;
}
```

- 方式二：智能指针访问

```
for(auto i : mp)
cout << i.first << " " << i.second << endl;//键，值
```

- 方式三：对指定单个元素访问

```
map<char,int>::iterator it = mp.find('a');
cout << it -> first << " " <<  it->second << "\n";
```

- 方式四：c++17特性才具有

```
for(auto [x, y] : mp)
        cout << x << " " << y << "\n";
//x,y对应键和值
```

6.5 与unordered_map的比较

这里就不单开一个大目录讲unordered_map了，直接在map里面讲了。

6.5.1 内部实现原理

map：内部用**红黑树**实现，具有**自动排序**（按键从小到大）功能。

unordered_map：内部用**哈希表**实现，内部元素无序杂乱。

6.5.2 效率比较

map：

- 优点：内部用 红黑树 实现，内部元素具有有序性，查询删除等操作复杂度为$O(logN)$

- 缺点：占用空间，红黑树里每个节点需要保存父子节点和红黑性质等信息，空间占用较大。

unordered_map：

- 优点：内部用哈希表实现，查找速度非常快（适用于大量的查询操作）。
- 缺点：建立哈希表比较耗时。

> 两者方法函数基本一样，差别不大。
> 注意：
>
> - 随着内部元素越来越多，两种容器的插入删除查询操作的时间都会逐渐变大，效率逐渐变低。
> - 使用 `[]` 查找元素时，如果元素不存在，两种容器**都是**创建一个空的元素；如果存在，会正常索引对应的值。所以如果查询过多的不存在的元素值，容器内部会创建大量的空的键值对，后续查询创建删除效率会**大大降低**。
> - 查询容器内部元素的最优方法是：先判断存在与否，再索引对应值（适用于这两种容器）
>
> ```
> // 以 map 为例
> map<int, int> mp;
> int x = 999999999;
> if(mp.count(x)) // 此处判断是否存在x这个键
>     cout << mp[x] << "\n";   // 只有存在才会索引对应的值，避免不存在x时多余空元素的创建
> ```

另外：

> 还有一种映射： `multimap`
> 键可以重复，即一个键对应多个值，如要了解，可以自行搜索。

6.5.3 自定义hash函数

由于unordered_map中的元素需要具备hash特性，如果语言没有自带hash特性的话，需要我们自定义hash函数，以下举一个 `pair<int, int>` 的hash函数定义的例子，hash函数看自己怎么定义了（只要能实现hash功能就行）。

```
// 使用 lambda 表达式来定义哈希函数
auto hash_pair = [](const std::pair<int, int>& p) -> std::size_t {
    static hash<long long> hash_ll;
    return hash_ll(p.first + (static_cast<long long>(p.second) << 32));
};

// 使用 lambda 表达式作为哈希函数定义 unordered_map, 10为桶的数量
std::unordered_map<std::pair<int, int>, int, decltype(hash_pair)> my_map(10, hash_pair);
```