关联容器中的元素是按关键字来保存和访问的，支持高校的关键词查找和访问，顺序容器中的元素是按他们在容器中的位置来顺序保存和访问的。无序容器使用哈希函数来组织元素。multi：允许重复关键字，unordered：不保持关键字按顺序存储。

|  |  |
| --- | --- |
| 关联容器类型 | |
| map | 关联数组；保存关键字——值对 |
| set | 关键字即值，只保存关键字的容器，没有值 |
| multimap | 关键字可重复出现 |
| multiset |
| unordered\_map | 用哈希函数组织的无序 |
| unordered\_set |
| unordered\_multimap | 用哈希函数组织的无序，关键字可重复出现 |
| unordered\_multiset |

# 1.使用关联容器：

## 1.1使用map：

map<string,size\_t> word\_count;

string word;

while(cin>>word)

++word\_count[word];

//如果word还未在map中，下标运算符会创建一个新元素，其关键字为word，值为0，不管元素是否是新建的，我们将其值加1

for(const auto &w:word\_count)

cout<<w.first<<” occurs ”<<w.second<<((w.second > 1)?” times”:” time”);

//当从map中提取一个元素时，会得到一个pair类型的对象，pair是一个模板类型，保存两个名为first和second的数据成员，map所使用的pair用first成员保存关键字，用second成员保存对应的值。

## 1.2使用set：

set<string> exlude = {“The”,”But”,”And”,”Or”,”An”,”A”};//可以对一个关联容器进行列表初始化

if(exlude.find(word) == exlude.end())//找不到

# 2.关联容器概述：

管理容器都支持P294中介绍的普通容器操作，不支持顺序容器的位置相关的操作，如push\_back和push\_front，原因是关联容器中元素是根据关键字存储的，这些操作对关联容器没有意义，而且关联容器也不支持构造函数或插入操作这些接受一个元素值和一个数量值的操作。

关联容器的迭代器都是双向的！

## 2.1定义关联容器：

每个关联容器都定义了一个默认的构造函数，它创建一个指定类型的空容器。我们也可以将关联容器初始化为零一个同类型容器的拷贝，或是从一个值范围来初始化关联容器，只要这些值可以转化为容器所需类型就可以。

map<string,int> authors = {{“asdasd”,1},{“asd2”,156},{“3rwd”,96}};//三个映射

multiset<int> miset(v.begin(),v.end());

## 2.2关键字类型的要求：

关联容器对其关键字类型有一些限制，对于有序容器，关键字类型必须定义元素比较的方法。默认情况下，标准库使用关键字类型的<运算符来比较两个关键字。

比较操作必须在关键字类型上定义一个严格弱序，可以将严格弱序看作“小于等于”，必须定义具备以下基本性质：①两个关键字不能同时“小于等于”对方，②如果k1小于等于k2，且k2小于等于k3，那么k1必须小于等于k3，③如果存在两个关键字，任何一个都不小于等于另一个，那么我们称这两个关键字是等价的。如果k1等价于k2，且k2等价于k3，那么k1必须等价于k3。

自定义的操作类型必须在尖括号中紧跟着元素类型指出。为了使用自己定义的操作，在定义关联容器时我们必须提供两个类型：关键字类型以及比较操作类型。当定义此容器类型的对象时，需要提供想要使用的操作的指针。

multiset<Sales\_data,decltype(comp)\*> bookstore(comp);//使用decltype来指出自定义操作的类型，当用decltype来获得一个函数指针类型时，必须加上一个\*来指出我们要使用一个给定函数类型的指针。

## 2.3pair类型：

定义在头文件utility中，一个pair保存两个数据成员。类似容器，pair是一个用来生成特定类型的模板，两个类型不要求一样。默认下进行值初始化。

pair<vector<int>,size\_t> anno{{1,2,3,4,5},5};//提供初始化器

pair的数据成员是public的，两个成员分别命名为first和second，我们用普通的成员访问符号来访问。

|  |  |
| --- | --- |
| pair上的操作 | |
| pair<T1,T2> p; | p是一个pair，两个类型分别为T1和T2的成员进行了值初始化 |
| pair<T1,T2> p(v1,v2); | p是一个成员类型为T1和T2的pair，first和second成员分别用v1和v2进行初始化 |
| pair<T1,T2> p = {v1,v2}; |
| make\_pair(v1,v2); | 返回一个用v1和v2初始化的pair，类型从v1和v2推断出来 |
| p.first; | 返回数据成员 |
| p.second; |
| p1 relop p2; | 关系运算符（<、>、<=、>=）按字典序定义，关系运算利用元素<的运算符来实现。 |
| p1 == p2; | first和second成员分别相等时，两个pair相等。 |
| p1 != p2; |

pair<string,int> process(vector<string> &v)

{

if(!v.empty())

return {v.back,v.back().size();}

else return pair<string,int> ();

}

在较早的C++版本中，不允许用花括号包围的初始化器来返回pair这种类型的对象，必须显示构造返回值。

if(!v.empty())//版本一

return pair<string,int>(v.back(),v.back().size());

if(!v.empty())//版本二

return make\_pair(v.back(),v.back().size());

# 3.关联容器操作：

还有一些表示容器关键字和值的类型：

|  |  |
| --- | --- |
| 关联容器额外的类型别名 | |
| key\_type | 此容器类型的关键字类型。 |
| mapped\_type | 每个关键字关联的类型；只适用于map及相关类型 |
| value\_type | 对于set，与key\_type相同，set中保存的值就是关键字  对于map，为pair<const key\_type,mapped\_type>，由于我们不能改变一个元素的关键字，因此这些pair的关键字部分是const的。 |

set<string>::key\_type v1;//v1是一个string

map<string,int>::value\_type v2;//v2是一个pair<const string,int>

## 3.1关联容器迭代器：

auto map\_it = word.begin();

cout<<map\_it->first;

cout<<” ”<<map\_it->second;

map\_it->first = “new key”;//错误，关键字是const的！

++map\_it->second;

set的迭代器是const的，遍历map和set和之前的一样。我们通常不对关联容器使用泛型算法，关键字是const这一特性意味着不能将关联容器传递给修改或重排容器元素的算法，set类型中的元素是const的，map类型中的元素是pair，第一个成员是const的。关联容器可用于只读取元素的算法。但是，很多这类算法都要搜索序列。由于关联容器中的元素不能通过它们的关键字进行查找，因此对其使用泛型搜索算法几乎是个坏主意。

在实际编程中，如果我们真要对一个关联容器使用算法，要么是将它当作一个源序列，要么当作一个目的位置。

## 3.2添加元素：

|  |  |
| --- | --- |
| 关联容器insert操作 | |
| c.insert(v); | v是value\_type类型的对象；args用来构造一个元素。  对于map和set，只有当元素的关键字不在c中时才插入（或构造）元素。函数返回一个pair，包含一个迭代器，指向具有指定关键字的元素，以及一个指示插入是否成功的bool值。如果关键字已经在容器中，则insert什么事情也不做，且返回值中的bool部分为false。  对于multimap和multiset，总会插入（或构造）给定元素，并返回一个指向新元素的迭代器，无需bool。 |
| c.emplace(args); |
| c.insert(b,e); | b和e是迭代器，表示一个c::value\_type类型值的范围，函数返回void。 |
| c.insert(il); | il是这种值的花括号列表，函数返回void。 |
| c.insert(p,v); | 将迭代器p作为一个提示，指出从哪里开始搜索新元素应该存储的位置。返回一个迭代器，指向具有给定关键字的元素。 |
| c.emplace(p,args); |

map<string,size\_t> word\_count;

string word;

while(cin>>word)

{

//等价于pair<map<string,size\_t>::iterator,bool> ret = ……

auto ret = word\_count.insert({word,1});

if(!ret.second) ++ret.first->second;

}

## 3.3删除元素：

|  |  |
| --- | --- |
| 从关联容器删除元素 | |
| c.erase(k); | 删除关键字为k的元素，返回一个size\_type值，指出删除的元素的数量 |
| c.erase(p); | 删除迭代器p指定的元素，p必须指向c中一个真实元素，不能等于c.end()，返回一个指向p之后元素的迭代器，若p指向c中的尾元素，则返回c.end(); |
| c.erase(b,e); | 删除迭代器对b和e所表示的范围中的元素，返回e |

## 3.4map的下标操作：

|  |  |
| --- | --- |
| 非const的map和unordered\_map的下标操作 | |
| c[k] | 返回关键字为k的元素，如果k不在c中，添加一个关键字为k的元素，对其进行值初始化。 |
| c.at(k) | 访问关键字为k的元素，带参数检查，若k不在c中，抛出一个out\_of\_range异常。 |

set类型不支持下标，因为set中没有与关键字相关联的值；multimap和unordered\_multimap不支持下标操作，因为这些容器中可能多个值与一个关键字相关联。

当对一个map进行下表操作时，会获得一个,apped\_type对象；但当解引用一个map迭代器时，会得到一个value\_type对象。

## 3.5访问元素：

|  |  |  |
| --- | --- | --- |
| 在一个关联容器中查找元素的操作 | | |
| 第一种方法 | c.find(k) | 返回一个迭代器，指向第一个关键字为k的元素，若k不在容器中，则返回尾后迭代器。 |
| c.count(k) | 返回关键字等于k的元素数量，对于不允许重复关键字的容器，返回值永远是0或1。 |
| 第二种方法 | c.lower\_bound(k) | 返回一个迭代器，指向第一个关键字不小于k的元素（不支持无序容器） |
| c.upper\_bound(k) | 返回一个迭代器，指向第一个关键字大于k的元素（不支持无序容器） |
| 第三种方法 | c.equal\_range(k) | 返回一个迭代器pair，表示关键字等于k的元素的范围。若k存在，则第一个迭代器指向第一个与关键字匹配的元素，第二个迭代器指向最后一个匹配元素之后的位置；若k不存在，pair的两个成员均等于可以插入的位置（即c.end()） |

使用下标操作有一个严重的副作用：如果关键字还未在map中，下标操作会插入一个具有给定关键字的元素，但有时我们只是想知道一个给定关键字是否在map中，而不想改变map，这样就不能使用下标运算符来检查一个元素是否存在，因为如果关键字不存在的话，下标运算符会插入一个新元素，这种情况下应该使用find。

如果一个multimap或multiset中有多个元素具有给定关键字，则这些元素在容器中会相邻存储。

如果关键字在容器中，lower\_bound返回的迭代器将指向第一个具有给定关键字的元素，而upper\_bound返回迭代器则指向最后一个匹配给定关键字的元素之后的位置；如果元素不再multimap中，则lower\_bound和upper\_bound会返回相等的迭代器——指向一个不影响排序的关键字插入位置。如果我们查找的元素具有容器中最大的关键字，则此关键字的upper\_bound返回尾后迭代器；如果关键字不存在，且大一容器中任何关键字，则lower\_bound返回的也是尾后迭代器。如果容器中没有这样的元素，返回迭代器将指向第一个关键字大于搜索对象的元素。这两个操作不报告关键字是否存在，重要的是它们返回值可作为一个迭代器范围。

# 4.无序容器：

定义了4个无序容器，这些容器不是使用比较运算符来组织元素，而是使用一个哈希函数和关键字类型的==运算符。在关键字类型的元素没有明显的序关系的情况下，无序容器是非常有用的。在某些应用中，维护元素的徐代价非常高昂，此时无序容器也很有用。无序容器也有允许重复关键字的版本。

无序容器在存储上组织为一组桶，每个桶保存0个或多个元素。无序容器使用一个哈希函数将元素映射到桶。为了访问一个元素，容器首先计算元素的哈希值，它指出应该搜索哪个桶。容器将具有一个特定哈希值的所有元素都保存在相同的桶中。如果容器允许重复关键字，所具有相同关键字的元素也都会在同一个桶中，因此无序容器的性能以来于哈希函数的质量和桶的数量和大小。

|  |  |  |
| --- | --- | --- |
| 无序容器管理操作 | | |
| 桶接口 | c.bucket\_count() | 正在使用的桶的数目 |
| c.max\_bucket\_nt() | 容器能容纳的最多桶的数量 |
| c.bucket\_size(n) | 第n个桶中有多少个元素 |
| c.bucket(k) | 关键字为k的元素在哪个桶中 |
| 桶迭代 | local\_iterator | 可用来访问桶中元素的迭代器类型 |
| const\_local\_iterator | 桶迭代器的const 版本 |
| c.begin(n),c.end(n) | 桶n的首元素迭代器和尾后迭代器 |
| c.cbegin(n),c.cend(n) | 同上 |
| 哈希策略 | c.load\_factor() | 每个桶的平均元素数量，返回float值 |
| c.max\_load\_factor() | c试图维护的平均桶大小，返回float值。c会在需要时添加新的桶，以使得load\_factor<=max\_load\_factor |
| c.rehash(n) | 重组存储，使得bucket\_count>=n且bucket\_count>size/max\_load\_factor |
| c.reserve(n) | 重组存储，使得c可以保存n个元素且不必rehash |

无序容器用一个hash<key\_type>类型的对象来生成每个元素的哈希值。标准库为内置类型（包括指针）提供了hash模板，还为一些标准库类型（包括string和智能指针类型定义了hash）。因此我们可以直接定义关键字是内置类型、string、智能指针类型的无序容器。但是我们不能直接定义关键字类型为自定义类类型的无序容器，如果没有定义==和哈希值计算函数，它不能直接使用哈希模板，而必须提供我们自己的hash模板版本。

//为了能将类类型Sales\_data用作关键字，我们需要提供函数来替代==运算符和哈希值计算函数

size\_t hasher(const Sales\_data &sd)

{ return hash<string>()(sd.isbn());}

bool eqOp(const Sales\_data &lhs, const Sales\_data &rhs)

{return lhs.isbn() == rhs.isbn();}

using SD\_multiset = unordered\_multiset<Sales\_data,decltype(hash)\*,decltype(eqOp)\*>;

SD\_multiset bookstore(42,hasher,eqOp);