

第 11 章

关联容器

内容

11.1 使用关联容器.....	374
11.2 关联容器概述.....	376
11.3 关联容器操作.....	381
11.4 无序容器.....	394
小结	397
术语表	397

关联容器和顺序容器有着根本的不同：关联容器中的元素是按关键字来保存和访问的。与之相对，顺序容器中的元素是按它们在容器中的位置来顺序保存和访问的。

虽然关联容器的很多行为与顺序容器相同，但其不同之处反映了关键字的作用。

420

关联容器支持高效的关键字查找和访问。两个主要的关联容器（associative-container）类型是 **map** 和 **set**。**map** 中的元素是一些关键字-值（key-value）对：关键字起到索引的作用，值则表示与索引相关联的数据。**set** 中每个元素只包含一个关键字；**set** 支持高效的关键字查询操作——检查一个给定关键字是否在 **set** 中。例如，在某些文本处理过程中，可以用一个 **set** 来保存想要忽略的单词。字典则是一个很好的使用 **map** 的例子：可以将单词作为关键字，将单词释义作为值。

标准库提供 8 个关联容器，如表 11.1 所示。这 8 个容器的不同体现在三个维度上：每个容器（1）或者是一个 **set**，或者是一个 **map**；（2）或者要求不重复的关键字，或者允许重复关键字；（3）按顺序保存元素，或无序保存。允许重复关键字的容器的名字中都包含单词 **multi**；不保持关键字按顺序存储的容器的名字都以单词 **unordered** 开头。因此一个 **unordered_multi_set** 是一个允许重复关键字，元素无序保存的集合，而一个 **set** 则是一个要求不重复关键字，有序存储的集合。无序容器使用哈希函数来组织元素，我们将在 11.4 节（第 394 页）中详细介绍有关哈希函数的更多内容。

类型 **map** 和 **multimap** 定义在头文件 **map** 中；**set** 和 **multiset** 定义在头文件 **set** 中；无序容器则定义在头文件 **unordered_map** 和 **unordered_set** 中。

表 11.1：关联容器类型

按关键字有序保存元素	
map	关联数组；保存关键字-值对
set	关键字即值，即只保存关键字的容器
multimap	关键字可重复出现的 map
multiset	关键字可重复出现的 set
无序集合	
unordered_map	用哈希函数组织的 map
unordered_set	用哈希函数组织的 set
unordered_multimap	哈希组织的 map ；关键字可以重复出现
unordered_multiset	哈希组织的 set ；关键字可以重复出现



11.1 使用关联容器

虽然大多数程序员都熟悉诸如 **vector** 和 **list** 这样的数据结构，但他们中很多人从未使用过关联数据结构。在学习标准库关联容器类型的详细内容之前，我们首先来看一个如何使用这类容器的例子，这对后续学习很有帮助。

map 是关键字-值对的集合。例如，可以将一个人的名字作为关键字，将其电话号码作为值。我们称这样的数据结构为“将名字映射到电话号码”。**map** 类型通常被称为**关联数组**（associative array）。关联数组与“正常”数组类似，不同之处在于其下标不必是整数。421 我们通过一个关键字而不是位置来查找值。给定一个名字到电话号码的 **map**，我们可以使用一个人的名字作为下标来获取此人的电话号码。

与之相对，**set** 就是关键字的简单集合。当只是想知道一个值是否存在时，**set** 是最有用的。例如，一个企业可以定义一个名为 **bad_checks** 的 **set** 来保存那些曾经开过空头支票的人的名字。在接受一张支票之前，可以查询 **bad_checks** 来检查顾客的名字是否在其中。

使用 map

一个经典的使用关联数组的例子是单词计数程序：

```
// 统计每个单词在输入中出现的次数
map<string, size_t> word_count; // string 到 size_t 的空 map
string word;
while (cin >> word)
    ++word_count[word];           // 提取 word 的计数器并将其加 1
for (const auto &w : word_count) // 对 map 中的每个元素
    // 打印结果
    cout << w.first << " occurs " << w.second
    << ((w.second > 1) ? " times" : " time") << endl;
```

此程序读取输入，报告每个单词出现多少次。

类似顺序容器，关联容器也是模板（参见 3.3 节，第 86 页）。为了定义一个 map，我们必须指定关键字和值的类型。在此程序中，map 保存的每个元素中，关键字是 string 类型，值是 size_t 类型（参见 3.5.2 节，第 103 页）。当对 word_count 进行下标操作时，我们使用一个 string 作为下标，获得与此 string 相关联的 size_t 类型的计数器。

while 循环每次从标准输入读取一个单词。它使用每个单词对 word_count 进行下标操作。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。不管元素是否是新创建的，我们将其值加 1。

一旦读取完所有输入，范围 for 语句（参见 3.2.3 节，第 81 页）就会遍历 map，打印每个单词和对应的计数器。当从 map 中提取一个元素时，会得到一个 pair 类型的对象，我们将在 11.2.3 节（第 379 页）介绍它。简单来说，pair 是一个模板类型，保存两个名为 first 和 second 的（公有）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。因此，输出语句的效果是打印每个单词及其关联的计数器。

如果我们对本节第一段中的文本（指英文版中的文本）运行这个程序，输出将会是：

```
Although occurs 1 time
Before occurs 1 time
an occurs 1 time
and occurs 1 time
...
...
```

使用 set

422

上一个示例程序的一个合理扩展是：忽略常见单词，如“the”、“and”、“or”等。我们可以使用 set 保存想忽略的单词，只对不在集合中的单词统计出现次数：

```
// 统计输入中每个单词出现的次数
map<string, size_t> word_count; // string 到 size_t 的空 map
set<string> exclude = {"The", "But", "And", "Or", "An", "A",
                       "the", "but", "and", "or", "an", "a"};
string word;
while (cin >> word)
    // 只统计不在 exclude 中的单词
    if (exclude.find(word) == exclude.end())
        ++word_count[word]; // 获取并递增 word 的计数器
```

与其他容器类似，`set` 也是模板。为了定义一个 `set`，必须指定其元素类型，本例中是 `string`。与顺序容器类似，可以对一个关联容器的元素进行列表初始化（参见 9.2.4 节，第 300 页）。集合 `exclude` 中保存了 12 个我们想忽略的单词。

此程序与前一个程序的重要不同是，在统计每个单词出现次数之前，我们检查单词是否在忽略集合中，这是在 `if` 语句中完成的：

```
// 只统计不在 exclude 中的单词
if (exclude.find(word) == exclude.end())
```

`find` 调用返回一个迭代器。如果给定关键字在 `set` 中，迭代器指向该关键字。否则，`find` 返回尾后迭代器。在此程序中，仅当 `word` 不在 `exclude` 中时我们才更新 `word` 的计数器。

如果用此程序处理与之前相同的输入，输出将会是：

```
Although occurs 1 time
Before occurs 1 time
are occurs 1 time
as occurs 1 time
...
...
```

11.1 节练习

练习 11.1：描述 `map` 和 `vector` 的不同。

练习 11.2：分别给出最适合使用 `list`、`vector`、`deque`、`map` 以及 `set` 的例子。

练习 11.3：编写你自己的单词计数程序。

练习 11.4：扩展你的程序，忽略大小写和标点。例如，“example.”、“example,”和“Example”应该递增相同的计数器。

423 >

11.2 关联容器概述

关联容器（有序的和无序的）都支持 9.2 节（第 294 页）中介绍的普通容器操作（列于表 9.2，第 295 页）。关联容器不支持顺序容器的位置相关的操作，例如 `push_front` 或 `push_back`。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。

除了与顺序容器相同的操作之外，关联容器还支持一些顺序容器不支持的操作（参见表 11.7，第 388 页）和类型别名（参见表 11.3，第 381 页）。此外，无序容器还提供一些用来调整哈希性能的操作，我们将在 11.4 节（第 394 页）中介绍。

关联容器的迭代器都是双向的（参见 10.5.1 节，第 365 页）。



11.2.1 定义关联容器

如前所示，当定义一个 `map` 时，必须既指明关键字类型又指明值类型；而定义一个 `set` 时，只需指明关键字类型，因为 `set` 中没有值。每个关联容器都定义了一个默认构

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另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。在新标准下，我们也可以对关联容器进行值初始化：

```
map<string, size_t> word_count; // 空容器
// 列表初始化
set<string> exclude = {"the", "but", "and", "or", "an", "a",
                        "The", "But", "And", "Or", "An", "A"};
// 三个元素；authors 将姓映射到名
map<string, string> authors = { {"Joyce", "James"}, 
                                 {"Austen", "Jane"}, 
                                 {"Dickens", "Charles"} };
```

与以往一样，初始化器必须能转换为容器中元素的类型。对于 `set`，元素类型就是关键字类型。

当初始化一个 `map` 时，必须提供关键字类型和值类型。我们将每个关键字-值对包围在花括号中：

```
{key, value}
```

来指出它们一起构成了 `map` 中的一个元素。在每个花括号中，关键字是第一个元素，值是第二个。因此，`authors` 将姓映射到名，初始化后它包含三个元素。

初始化 `multimap` 或 `multiset`

一个 `map` 或 `set` 中的关键字必须是唯一的，即，对于一个给定的关键字，只能有一个元素的关键字等于它。容器 `multimap` 和 `multiset` 没有此限制，它们都允许多个元素具有相同的关键字。例如，在我们用来统计单词数量的 `map` 中，每个单词只能有一个元素。另一方面，在一个词典中，一个特定单词则可具有多个与之关联的词义。 ◀424

下面的例子展示了具有唯一关键字的容器与允许重复关键字的容器之间的区别。首先，我们将创建一个名为 `ivec` 的保存 `int` 的 `vector`，它包含 20 个元素：0 到 9 每个整数有两个拷贝。我们将使用此 `vector` 初始化一个 `set` 和一个 `multiset`：

```
// 定义一个有 20 个元素的 vector，保存 0 到 9 每个整数的两个拷贝
vector<int> ivec;
for (vector<int>::size_type i = 0; i != 10; ++i) {
    ivec.push_back(i);
    ivec.push_back(i); // 每个数重复保存一次
}
// iset 包含来自 ivec 的不重复的元素；miset 包含所有 20 个元素
set<int> iset(ivec.cbegin(), ivec.cend());
multiset<int> miset(ivec.cbegin(), ivec.cend());
cout << ivec.size() << endl; // 打印出 20
cout << iset.size() << endl; // 打印出 10
cout << miset.size() << endl; // 打印出 20
```

即使我们用整个 `ivec` 容器来初始化 `iset`，它也只含有 10 个元素：对应 `ivec` 中每个不同的元素。另一方面，`miset` 有 20 个元素，与 `ivec` 中的元素数量一样多。

11.2.1 节练习

练习 11.5：解释 map 和 set 的区别。你如何选择使用哪个？

练习 11.6：解释 set 和 list 的区别。你如何选择使用哪个？

练习 11.7：定义一个 map，关键字是家庭的姓，值是一个 vector，保存家中孩子（们）的名。编写代码，实现添加新的家庭以及向已有家庭中添加新的孩子。

练习 11.8：编写一个程序，在一个 vector 而不是一个 set 中保存不重复的单词。使用 set 的优点是什么？



11.2.2 关键字类型的要求

关联容器对其关键字类型有一些限制。对于无序容器中关键字的要求，我们将在 11.4 节（第 396 页）中介绍。对于有序容器——map、multimap、set 以及 multiset，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。在集合类型中，关键字类型就是元素类型；在映射类型中，关键字类型是元素的第一部分的类型。因此，11.2 节（第 377 页）中 `word_count` 的关键字类型是 `string`。类似的，`exclude` 的关键字类型也是 `string`。
425>



传递给排序算法的可调用对象（参见 10.3.1 节，第 344 页）必须满足与关联容器中关键字一样的类型要求。

有序容器的关键字类型

可以向一个算法提供我们自己定义的比较操作（参见 10.3 节，第 344 页），与之类似，也可以提供自己定义的操作来代替关键字上的`<`运算符。所提供的操作必须在关键字类型上定义一个**严格弱序**（strict weak ordering）。可以将严格弱序看作“小于等于”，虽然实际定义的操作可能是一个复杂的函数。无论我们怎样定义比较函数，它必须具备如下基本性质：

- 两个关键字不能同时“小于等于”对方；如果 `k1` “小于等于” `k2`，那么 `k2` 绝不能“小于等于” `k1`。
- 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`。
- 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`。

如果两个关键字是等价的（即，任何一个都不“小于等于”另一个），那么容器将它们视作相等来处理。当用作 map 的关键字时，只能有一个元素与这两个关键字关联，我们可以用两者中任意一个来访问对应的值。



在实际编程中，重要的是，如果一个类型定义了“行为正常”的`<`运算符，则它可以用于关键字类型。

使用关键字类型的比较函数

用来组织一个容器中元素的操作的类型也是该容器类型的一部分。为了指定使用自定义的操作，必须在定义关联容器类型时提供此操作的类型。如前所述，用尖括号指出要定

义哪种类型的容器，自定义的操作类型必须在尖括号中紧跟着元素类型给出。

在尖括号中出现的每个类型，就仅仅是一个类型而已。当我们创建一个容器（对象）时，才会以构造函数参数的形式提供真正的比较操作（其类型必须与在尖括号中指定的类型相吻合）。

例如，我们不能直接定义一个 Sales_data 的 multiset，因为 Sales_data 没有 < 运算符。但是，可以用 10.3.1 节练习（第 345 页）中的 compareIsbn 函数来定义一个 multiset。此函数在 Sales_data 对象的 ISBN 成员上定义了一个严格弱序。函数 compareIsbn 应该像下面这样定义

```
bool compareIsbn(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() < rhs.isbn();
}
```

426

为了使用自己定义的操作，在定义 multiset 时我们必须提供两个类型：关键字类型 Sales_data，以及比较操作类型——应该是一种函数指针类型（参见 6.7 节，第 221 页），可以指向 compareIsbn。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。在本例中，我们提供一个指向 compareIsbn 的指针：

```
// bookstore 中多条记录可以有相同的 ISBN
// bookstore 中的元素以 ISBN 的顺序进行排列
multiset<Sales_data, decltype(compareIsbn)*>
bookstore(compareIsbn);
```

此处，我们使用 decltype 来指出自定义操作的类型。记住，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（参见 6.7 节，第 223 页）。用 compareIsbn 来初始化 bookstore 对象，这表示当我们向 bookstore 添加元素时，通过调用 compareIsbn 来为这些元素排序。即，bookstore 中的元素将按它们的 ISBN 成员的值排序。可以用 compareIsbn 代替 &compareIsbn 作为构造函数的参数，因为当我们使用一个函数的名字时，在需要的情况下它会自动转化为一个指针（参见 6.7 节，第 221 页）。当然，使用 &compareIsbn 的效果也是一样的。

11.2.2 节练习

练习 11.9： 定义一个 map，将单词与一个行号的 list 关联，list 中保存的是单词所出现的行号。

练习 11.10： 可以定义一个 vector<int>::iterator 到 int 的 map 吗？ list<int>::iterator 到 int 的 map 呢？对于两种情况，如果不能，解释为什么。

练习 11.11： 不使用 decltype 重新定义 bookstore。

11.2.3 pair 类型

在介绍关联容器操作之前，我们需要了解名为 **pair** 的标准库类型，它定义在头文件 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。当创建一个 pair 时，我们必须提供两个类型名，pair 的数据成员将具有对应的类型。两个类型不要求一样：

```

pair<string, string> anon;           // 保存两个 string
pair<string, size_t> word_count;    // 保存一个 string 和一个 size_t
pair<string, vector<int>> line;     // 保存 string 和 vector<int>

```

427 pair 的默认构造函数对数据成员进行值初始化（参见 3.3.1 节，第 88 页）。因此，anon 是一个包含两个空 string 的 pair，line 保存一个空 string 和一个空 vector。word_count 中的 size_t 成员值为 0，而 string 成员被初始化为空。

我们也可以为每个成员提供初始化器：

```
pair<string, string> author{"James", "Joyce"};
```

这条语句创建一个名为 author 的 pair，两个成员被初始化为"James"和"Joyce"。

与其他标准库类型不同，pair 的数据成员是 public 的（参见 7.2 节，第 240 页）。两个成员分别命名为 first 和 second。我们用普通的成员访问符号（参见 1.5.2 节，第 20 页）来访问它们，例如，在第 375 页的单词计数程序的输出语句中我们就是这么做的：

```

// 打印结果
cout << w.first << " occurs " << w.second
    << ((w.second > 1) ? " times" : " time") << endl;

```

此处，w 是指向 map 中某个元素的引用。map 的元素是 pair。在这条语句中，我们首先打印关键字——元素的 first 成员，接着打印计数器——second 成员。标准库只定义了有限的几个 pair 操作，表 11.2 列出了这些操作。

表 11.2: pair 上的操作

pair<T1, T2> p;	p 是一个 pair，两个类型分别为 T1 和 T2 的成员都进行了值初始化（参见 3.3.1 节，第 88 页）
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} ;	等价于 p (v1,v2)
make_pair(v1, v2)	返回一个用 v1 和 v2 初始化的 pair。pair 的类型从 v1 和 v2 的类型推断出来
p.first	返回 p 的名为 first 的（公有）数据成员
p.second	返回 p 的名为 second 的（公有）数据成员
p1 relop p2	关系运算符 (<、>、<=、>=) 按字典序定义：例如，当 p1.first < p2.first 或 !(p2.first < p1.first) && p1.second < p2.second 成立时，p1 < p2 为 true。关系运算利用元素的< 运算符来实现
p1 == p2	当 first 和 second 成员分别相等时，两个 pair 相等。相等性判断利用元素的==运算符实现
p1 != p2	

创建 pair 对象的函数

C++ 11 想象有一个函数需要返回一个 pair。在新标准下，我们可以对返回值进行列表初始化（参见 6.3.2 节，第 203 页）

428

```

pair<string, int>
process(vector<string> &v)
{
    // 处理 v
}

```

```

    if (!v.empty())
        return {v.back(), v.back().size()}; // 列表初始化
    else
        return pair<string, int>(); // 隐式构造返回值
}

```

若 `v` 不为空，我们返回一个由 `v` 中最后一个 `string` 及其大小组成的 `pair`。否则，隐式构造一个空 `pair`，并返回它。

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，必须显式构造返回值：

```

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

```

我们还可以用 `make_pair` 来生成 `pair` 对象，`pair` 的两个类型来自于 `make_pair` 的参数：

```

if (!v.empty())
    return make_pair(v.back(), v.back().size());

```

11.2.3 节练习

练习 11.12： 编写程序，读入 `string` 和 `int` 的序列，将每个 `string` 和 `int` 存入一个 `pair` 中，`pair` 保存在一个 `vector` 中。

练习 11.13： 在上一题的程序中，至少有三种创建 `pair` 的方法。编写此程序的三个版本，分别采用不同的方法创建 `pair`。解释你认为哪种形式最易于编写和理解，为什么？

练习 11.14： 扩展你在 11.2.1 节练习（第 378 页）中编写的孩子姓到名的 `map`，添加一个 `pair` 的 `vector`，保存孩子的名和生日。

11.3 关联容器操作

除了表 9.2（第 295 页）中列出的类型，关联容器还定义了表 11.3 中列出的类型。这些类型表示容器关键字和值的类型。

表 11.3：关联容器额外的类型别名

<code>key_type</code>	此容器类型的关键字类型
<code>mapped_type</code>	每个关键字关联的类型；只适用于 <code>map</code>
<code>value_type</code>	对于 <code>set</code> ，与 <code>key_type</code> 相同 对于 <code>map</code> ，为 <code>pair<const key_type, mapped_type></code>

对于 `set` 类型，`key_type` 和 `value_type` 是一样的；`set` 中保存的值就是关键字。在一个 `map` 中，元素是关键字-值对。即，每个元素是一个 `pair` 对象，包含一个关键字和一个关联的值。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>::value_type v1;      // v1 是一个 string
set<string>::key_type v2;        // v2 是一个 string
map<string, int>::value_type v3; // v3 是一个 pair<const string, int>
map<string, int>::key_type v4;   // v4 是一个 string
map<string, int>::mapped_type v5; // v5 是一个 int

```

与顺序容器一样（参见 9.2.2 节，第 297 页），我们使用作用域运算符来提取一个类型的成员——例如，`map<string, int>::key_type`。

只有 `map` 类型（`unordered_map`、`unordered_multimap`、`multimap` 和 `map`）才定义了 `mapped_type`。

11.3.1 关联容器迭代器

当解引用一个关联容器迭代器时，我们会得到一个类型为容器的 `value_type` 的值的引用。对 `map` 而言，`value_type` 是一个 `pair` 类型，其 `first` 成员保存 `const` 的关键字，`second` 成员保存值：

```
// 获得指向 word_count 中一个元素的迭代器
auto map_it = word_count.begin();
// *map_it 是指向一个 pair<const string, size_t>对象的引用
cout << map_it->first;           // 打印此元素的关键字
cout << " " << map_it->second;   // 打印此元素的值
map_it->first = "new key";       // 错误：关键字是 const 的
++map_it->second; // 正确：我们可以通过迭代器改变元素
```



必须记住，一个 `map` 的 `value_type` 是一个 `pair`，我们可以改变 `pair` 的值，但不能改变关键字成员的值。

set 的迭代器是 `const` 的

虽然 `set` 类型同时定义了 `iterator` 和 `const_iterator` 类型，但两种类型都只允许只读访问 `set` 中的元素。与不能改变一个 `map` 元素的关键字一样，一个 `set` 中的关键字也是 `const` 的。可以用一个 `set` 迭代器来读取元素的值，但不能修改：

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
set<int>::iterator set_it = iset.begin();
if (set_it != iset.end()) {
    *set_it = 42;           // 错误：set 中的关键字是只读的
    cout << *set_it << endl; // 正确：可以读关键字
}
```

430 遍历关联容器

`map` 和 `set` 类型都支持表 9.2（第 295 页）中的 `begin` 和 `end` 操作。与往常一样，我们可以用这些函数获取迭代器，然后用迭代器来遍历容器。例如，我们可以编写一个循环来打印第 375 页中单词计数程序的结果，如下所示：

```
// 获得一个指向首元素的迭代器
auto map_it = word_count.cbegin();
// 比较当前迭代器和尾后迭代器
while (map_it != word_count.cend()) {
    // 解引用迭代器，打印关键字-值对
    cout << map_it->first << " occurs "
        << map_it->second << " times" << endl;
    ++map_it; // 递增迭代器，移动到下一个元素
}
```

`while` 的循环条件和循环中的迭代器递增操作看起来很像我们之前编写的打印一个 `vector`

或一个 `string` 的程序。我们首先初始化迭代器 `map_it`, 让它指向 `word_count` 中的首元素。只要迭代器不等于 `end`, 就打印当前元素并递增迭代器。输出语句解引用 `map_it` 来获得 `pair` 的成员, 否则与我们之前的程序一样。



本程序的输出是按字典序排列的。当使用一个迭代器遍历一个 `map`、`multimap`、`set` 或 `multiset` 时, 迭代器按关键字升序遍历元素。

关联容器和算法

我们通常不对关联容器使用泛型算法 (参见第 10 章)。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, 因为这类算法需要向元素写入值, 而 `set` 类型中的元素是 `const` 的, `map` 中的元素是 `pair`, 其第一个成员是 `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 (快速) 查找, 因此对其使用泛型搜索算法几乎总是个坏主意。例如, 我们将在 11.3.5 节 (第 388 页) 中看到, 关联容器定义了一个名为 `find` 的成员, 它通过一个给定的关键字直接获取元素。我们可以用泛型 `find` 算法来查找一个元素, 但此算法会进行顺序搜索。使用关联容器定义的专用的 `find` 成员会比调用泛型 `find` 快得多。

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。例如, 可以用泛型 `copy` 算法将元素从一个关联容器拷贝到另一个序列。类似的, 可以调用 `inserter` 将一个插入器绑定 (参见 10.4.1 节, 第 358 页) 到一个关联容器。通过使用 `inserter`, 我们可以将关联容器当作一个目的位置来调用另一个算法。

11.3.1 节练习

< 431

练习 11.15: 对一个 `int` 到 `vector<int>` 的 `map`, 其 `mapped_type`、`key_type` 和 `value_type` 分别是什么?

练习 11.16: 使用一个 `map` 迭代器编写一个表达式, 将一个值赋予一个元素。

练习 11.17: 假定 `c` 是一个 `string` 的 `multiset`, `v` 是一个 `string` 的 `vector`, 解释下面的调用。指出每个调用是否合法:

```
copy(v.begin(), v.end(), inserter(c, c.end()));
copy(v.begin(), v.end(), back_inserter(c));
copy(c.begin(), c.end(), inserter(v, v.end()));
copy(c.begin(), c.end(), back_inserter(v));
```

练习 11.18: 写出第 382 页循环中 `map_it` 的类型, 不要使用 `auto` 或 `decltype`。

练习 11.19: 定义一个变量, 通过对 11.2.2 节 (第 378 页) 中的名为 `bookstore` 的 `multiset` 调用 `begin()` 来初始化这个变量。写出变量的类型, 不要使用 `auto` 或 `decltype`。

11.3.2 添加元素

关联容器的 `insert` 成员 (见表 11.4, 第 384 页) 向容器中添加一个元素或一个元素范围。由于 `map` 和 `set` (以及对应的无序类型) 包含不重复的关键字, 因此插入一个已

存在的元素对容器没有任何影响：

```
vector<int> ivec = {2,4,6,8,2,4,6,8};           // ivec 有 8 个元素
set<int> set2;
set2.insert(ivec.cbegin(), ivec.cend());         // set2 有 4 个元素
set2.insert({1,3,5,7,1,3,5,7});                  // set2 现在有 8 个元素
```

`insert` 有两个版本，分别接受一对迭代器，或是一个初始化器列表，这两个版本的行为类似对应的构造函数（参见 11.2.1 节，第 376 页）——对于一个给定的关键字，只有第一个带此关键字的元素才被插入到容器中。

向 map 添加元素

对一个 `map` 进行 `insert` 操作时，必须记住元素类型是 `pair`。通常，对于想要插入的数据，并没有一个现成的 `pair` 对象。可以在 `insert` 的参数列表中创建一个 `pair`：

```
// 向 word_count 插入 word 的 4 种方法
word_count.insert({word, 1});
word_count.insert(make_pair(word, 1));
word_count.insert(pair<string, size_t>(word, 1));
word_count.insert(map<string, size_t>::value_type(word, 1));
```

如我们所见，在新标准下，创建一个 `pair` 最简单的方法是在参数列表中使用花括号初始化。也可以调用 `make_pair` 或显式构造 `pair`。最后一个 `insert` 调用中的参数：

```
map<string, size_t>::value_type(s, 1)
```

构造一个恰当的 `pair` 类型，并构造该类型的一个新对象，插入到 `map` 中。

表 11.4：关联容器 `insert` 操作

<code>c.insert(v)</code>	<code>v</code> 是 <code>value_type</code> 类型的对象； <code>args</code> 用来构造一个元素
<code>c.emplace(args)</code>	对于 <code>map</code> 和 <code>set</code> ，只有当元素的关键字不在 <code>c</code> 中时才插入（或构造）元素。函数返回一个 <code>pair</code> ，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 <code>bool</code> 值。 对于 <code>multimap</code> 和 <code>multiset</code> ，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
<code>c.insert(b, e)</code>	<code>b</code> 和 <code>e</code> 是迭代器，表示一个 <code>c::value_type</code> 类型值的范围； <code>i1</code> 是这种值的花括号列表。函数返回 <code>void</code>
<code>c.insert(i1)</code>	对于 <code>map</code> 和 <code>set</code> ，只插入关键字不在 <code>c</code> 中的元素。对于 <code>multimap</code> 和 <code>multiset</code> ，则会插入范围中的每个元素
<code>c.insert(p, v)</code>	类似 <code>insert(v)</code> （或 <code>emplace(args)</code> ），但将迭代器 <code>p</code> 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
<code>c.emplace(p, args)</code>	

检测 `insert` 的返回值

`insert`（或 `emplace`）返回的值依赖于容器类型和参数。对于不包含重复关键字的容器，添加单一元素的 `insert` 和 `emplace` 版本返回一个 `pair`，告诉我们插入操作是否成功。`pair` 的 `first` 成员是一个迭代器，指向具有给定关键字的元素；`second` 成员是一个 `bool` 值，指出元素是插入成功还是已经存在于容器中。如果关键字已在容器中，则 `insert` 什么也不做，且返回值中的 `bool` 部分为 `false`。如果关键字不存在，元

素被插入容器中，且 bool 值为 true。

作为一个例子，我们用 insert 重写单词计数程序：

```
// 统计每个单词在输入中出现次数的一种更烦琐的方法
map<string, size_t> word_count; // 从 string 到 size_t 的空 map
string word;
while (cin >> word) {
    // 插入一个元素，关键字等于 word，值为 1;
    // 若 word 已在 word_count 中，insert 什么也不做
    auto ret = word_count.insert({word, 1});
    if (!ret.second) // word 已在 word_count 中
        ++ret.first->second; // 递增计数器
}
```

对于每个 word，我们尝试将其插入到容器中，对应的值为 1。若 word 已在 map 中，则什么都不做，特别是与 word 相关联的计数器的值不变。若 word 还未在 map 中，则此 433 string 对象被添加到 map 中，且其计数器的值被置为 1。

if 语句检查返回值的 bool 部分，若为 false，则表明插入操作未发生。在此情况下，word 已存在于 word_count 中，因此必须递增此元素所关联的计数器。

展开递增语句

在这个版本的单词计数程序中，递增计数器的语句很难理解。通过添加一些括号来反映出运算符的优先级（参见 4.1.2 节，第 121 页），会使表达式更容易理解一些：

```
++((ret.first)->second); // 等价的表达式
```

下面我们一步一步来解释此表达式：

ret 保存 insert 返回的值，是一个 pair。

ret.first 是 pair 的第一个成员，是一个 map 迭代器，指向具有给定关键字的元素。

ret.first-> 解引用此迭代器，提取 map 中的元素，元素也是一个 pair。

ret.first->second map 中元素的值部分。

++ret.first->second 递增此值。

再回到原来完整的递增语句，它提取匹配关键字 word 的元素的迭代器，并递增与我们试图插入的关键字相关联的计数器。

如果读者使用的是旧版本的编译器，或者是在阅读新标准推出之前编写的代码，ret 的声明和初始化可能复杂些：

```
pair<map<string, size_t>::iterator, bool> ret =
    word_count.insert(make_pair(word, 1));
```

应该容易看出这条语句定义了一个 pair，其第二个类型为 bool 类型。第一个类型理解起来有点儿困难，它是一个在 map<string, size_t>类型上定义的 iterator 类型。

向 multiset 或 multimap 添加元素

我们的单词计数程序依赖于这样一个事实：一个给定的关键字只能出现一次。这样，任意给定的单词只有一个关联的计数器。我们有时希望能添加具有相同关键字的多个元素。例如，可能想建立作者到他所著书籍题目的映射。在此情况下，每个作者可能有多个

条目，因此我们应该使用 multimap 而不是 map。由于一个 multi 容器中的关键字不必唯一，在这些类型上调用 insert 总会插入一个元素：

434 >

```
multimap<string, string> authors;
// 插入第一个元素，关键字为 Barth, John
authors.insert({"Barth, John", "Sot-Weed Factor"});
// 正确：添加第二个元素，关键字也是 Barth, John
authors.insert({"Barth, John", "Lost in the Funhouse"});
```

对允许重复关键字的容器，接受单个元素的 insert 操作返回一个指向新元素的迭代器。这里无须返回一个 bool 值，因为 insert 总是向这类容器中加入一个新元素。

11.3.2 节练习

练习 11.20：重写 11.1 节练习（第 376 页）的单词计数程序，使用 insert 替代下标操作。你认为哪个程序更容易编写和阅读？解释原因。

练习 11.21：假定 word_count 是一个 string 到 size_t 的 map，word 是一个 string，解释下面循环的作用：

```
while (cin >> word)
    ++word_count.insert({word, 0}).first->second;
```

练习 11.22：给定一个 map<string, vector<int>>，对此容器的插入一个元素的 insert 版本，写出其参数类型和返回类型。

练习 11.23：11.2.1 节练习（第 378 页）中的 map 以孩子的姓为关键字，保存他们的名的 vector，用 multimap 重写此 map。

11.3.3 删除元素

关联容器定义了三个版本的 erase，如表 11.5 所示。与顺序容器一样，我们可以传递给 erase 一个迭代器或一个迭代器对来删除一个元素或者一个元素范围。这两个版本的 erase 与对应的顺序容器的操作非常相似：指定的元素被删除，函数返回 void。

关联容器提供一个额外的 erase 操作，它接受一个 key_type 参数。此版本删除所有匹配给定关键字的元素（如果存在的话），返回实际删除的元素的数量。我们可以用此版本在打印结果之前从 word_count 中删除一个特定的单词：

```
// 删除一个关键字，返回删除的元素数量
if (word_count.erase(removal_word))
    cout << "ok: " << removal_word << " removed\n";
else cout << "oops: " << removal_word << " not found!\n";
```

对于保存不重复关键字的容器，erase 的返回值总是 0 或 1。若返回值为 0，则表明想要删除的元素并不在容器中

435 > 对允许重复关键字的容器，删除元素的数量可能大于 1：

```
auto cnt = authors.erase("Barth, John");
```

如果 authors 是我们在 11.3.2 节（第 386 页）中创建的 multimap，则 cnt 的值为 2。

表 11.5：从关联容器删除元素

<code>c.erase(k)</code>	从 c 中删除每个关键字为 k 的元素。返回一个 <code>size_type</code> 值，指出删除的元素的数量
<code>c.erase(p)</code>	从 c 中删除迭代器 p 指定的元素。p 必须指向 c 中一个真实元素，不能等于 <code>c.end()</code> 。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 <code>c.end()</code>
<code>c.erase(b, e)</code>	删除迭代器对 b 和 e 所表示的范围中的元素。返回 e

11.3.4 map 的下标操作



`map` 和 `unordered_map` 容器提供了下标运算符和一个对应的 `at` 函数（参见 9.3.2 节，第 311 页），如表 11.6 所示。`set` 类型不支持下标，因为 `set` 中没有与关键字相关联的“值”。元素本身就是关键字，因此“获取与一个关键字相关联的值”的操作就没有意义了。我们不能对一个 `multimap` 或一个 `unordered_multimap` 进行下标操作，因为这些容器中可能有多个值与一个关键字相关联。

类似我们用过的其他下标运算符，`map` 下标运算符接受一个索引（即，一个关键字），获取与此关键字相关联的值。但是，与其他下标运算符不同的是，如果关键字并不在 `map` 中，会为它创建一个元素并插入到 `map` 中，关联值将进行值初始化（参见 3.3.1 节，第 88 页）。

例如，如果我们编写如下代码

```
map <string, size_t> word_count; // empty map
// 插入一个关键字为 Anna 的元素，关联值进行值初始化；然后将 1 赋予它
word_count["Anna"] = 1;
```

将会执行如下操作：

- 在 `word_count` 中搜索关键字为 `Anna` 的元素，未找到。
- 将一个新的关键字-值对插入到 `word_count` 中。关键字是一个 `const string`，保存 `Anna`。值进行值初始化，在本例中意味着值为 0。
- 提取出新插入的元素，并将值 1 赋予它。

由于下标运算符可能插入一个新元素，我们只可以对非 `const` 的 `map` 使用下标操作。

436



对一个 `map` 使用下标操作，其行为与数组或 `vector` 上的下标操作很不相同：使用一个不在容器中的关键字作为下标，会添加一个具有此关键字的元素到 `map` 中。

表 11.6：`map` 和 `unordered_map` 的下标操作

<code>c[k]</code>	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
<code>c.at(k)</code>	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 <code>out_of_range</code> 异常（参见 5.6 节，第 173 页）

使用下标操作的返回值

`map` 的下标运算符与我们用过的其他下标运算符的另一个不同之处是其返回类型。通

常情况下，解引用一个迭代器所返回的类型与下标运算符返回的类型是一样的。但对 map 则不然：当对一个 map 进行下标操作时，会获得一个 mapped_type 对象；但当解引用一个 map 迭代器时，会得到一个 value_type 对象（参见 11.3 节，第 381 页）。

与其他下标运算符相同的是，map 的下标运算符返回一个左值（参见 4.1.1 节，第 121 页）。由于返回的是一个左值，所以我们既可以读也可以写元素：

```
cout << word_count["Anna"];           // 用 Anna 作为下标提取元素；会打印出 1
++word_count["Anna"];                // 提取元素，将其增 1
cout << word_count["Anna"];           // 提取元素并打印它；会打印出 2
```



与 vector 与 string 不同，map 的下标运算符返回的类型与解引用 map 迭代器得到的类型不同。

如果关键字还未在 map 中，下标运算符会添加一个新元素，这一特性允许我们编写出异常简洁的程序，例如单词计数程序中的循环（参见 11.1 节，第 375 页）。另一方面，有时只是想知道一个元素是否已在 map 中，但在不存在时并不想添加元素。在这种情况下，就不能使用下标运算符。

11.3.4 节练习

练习 11.24：下面的程序完成什么功能？

```
map<int, int> m;
m[0] = 1;
```

练习 11.25：对比下面程序与上一题程序

```
vector<int> v;
v[0] = 1;
```

练习 11.26：可以用什么类型来对一个 map 进行下标操作？下标运算符返回的类型是什么？请给出一个具体例子——即，定义一个 map，然后写出一个可以用来对 map 进行下标操作的类型以及下标运算符将会返回的类型。

11.3.5 访问元素

关联容器提供多种查找一个指定元素的方法，如表 11.7 所示。应该使用哪个操作依赖于我们要解决什么问题。如果我们所关心的只不过是一个特定元素是否已在容器中，可能 find 是最佳选择。对于不允许重复关键字的容器，可能使用 find 还是 count 没什么区别。但对于允许重复关键字的容器，count 还会做更多的工作：如果元素在容器中，它还会统计有多少个元素有相同的关键字。如果不需要计数，最好使用 find：

```
set<int> iset = {0,1,2,3,4,5,6,7,8,9};
iset.find(1);    // 返回一个迭代器，指向 key == 1 的元素
iset.find(11);   // 返回一个迭代器，其值等于 iset.end()
iset.count(1);   // 返回 1
iset.count(11);  // 返回 0
```

表 11.7：在一个关联容器中查找元素的操作

lower_bound 和 upper_bound 不适用于无序容器。

下标和 at 操作只适用于非 const 的 map 和 unordered_map。

续表

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 的元素的数量。对于不允许重复关键字的容器，返回值永远是 0 或 1
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
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
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 不存在，pair 的两个成员均等于 c.end()

对 map 使用 find 代替下标操作

对 map 和 unordered_map 类型，下标运算符提供了最简单的提取元素的方法。但是，如我们所见，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。这种行为是否正确完全依赖于我们的预期是什么。例如，单词计数程序依赖于这样一个特性：使用一个不存在的关键字作为下标，会插入一个新元素，其关键字为给定关键字，其值为 0。也就是说，下标操作的行为符合我们的预期。

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。在这种情况下，应该使用 find：

```
if (word_count.find("foobar") == word_count.end())
    cout << "foobar is not in the map" << endl;
```

在 multimap 或 multiset 中查找元素

在一个不允许重复关键字的关联容器中查找一个元素是一件很简单的事情——元素要么在容器中，要么不在。但对于允许重复关键字的容器来说，过程就更为复杂：在容器中可能有很多元素具有给定的关键字。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。

例如，给定一个从作者到著作题目的映射，我们可能想打印一个特定作者的所有著作。◀ 438可以用三种不同方法来解决这个问题。最直观的方法是使用 find 和 count：

```
string search_item("Alain de Botton");           // 要查找的作者
auto entries = authors.count(search_item);         // 元素的数量
auto iter = authors.find(search_item);             // 此作者的第一本书
// 用一个循环查找此作者的所有著作
while(entries) {
    cout << iter->second << endl;                  // 打印每个题目
    ++iter;                                         // 前进到下一本书
    --entries;                                       // 记录已经打印了多少本书
}
```

首先调用 count 确定此作者共有多少本著作，并调用 find 获得一个迭代器，指向第一个关键字为此作者的元素。for 循环的迭代次数依赖于 count 的返回值。特别是，如果 count 返回 0，则循环一次也不执行。



当我们遍历一个 multimap 或 multiset 时，保证可以得到序列中所有具有给定关键字的元素。

一种不同的，面向迭代器的解决方法

我们还可以用 `lower_bound` 和 `upper_bound` 来解决此问题。这两个操作都接受一个关键字，返回一个迭代器。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` 返回的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。如果元素不在 `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。因此，用相同的关键字调用 `lower_bound` 和 `upper_bound` 会得到一个迭代器范围(参见 9.2.1 节，第 296 页)，表示所有具有该关键字的元素的范围。

439 >

当然，这两个操作返回的迭代器可能是容器的尾后迭代器。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，且大于容器中任何关键字，则 `lower_bound` 返回的也是尾后迭代器。



`lower_bound` 返回的迭代器可能指向一个具有给定关键字的元素，但也可能不指向。如果关键字不在容器中，则 `lower_bound` 会返回关键字的第一个安全插入点——不影响容器中元素顺序的插入位置。

使用这两个操作，我们可以重写前面的程序：

```
// authors 和 search_item 的定义，与前面的程序一样
// beg 和 end 表示对应此作者的元素的范围
for (auto beg = authors.lower_bound(search_item),
        end = authors.upper_bound(search_item);
     beg != end; ++beg)
    cout << beg->second << endl; // 打印每个题目
```

此程序与使用 `count` 和 `find` 的版本完成相同的工作，但更直接。对 `lower_bound` 的调用将 `beg` 定位到第一个与 `search_item` 匹配的元素（如果存在的话）。如果容器中没有这样的元素，`beg` 将指向第一个关键字大于 `search_item` 的元素，有可能是尾后迭代器。`upper_bound` 调用将 `end` 指向最后一个匹配指定关键字的元素之后的元素。这两个操作并不报告关键字是否存在，重要的是它们的返回值可作为一个迭代器范围（参见 9.2.1 节，第 296 页）。

如果没有元素与给定关键字匹配，则 `lower_bound` 和 `upper_bound` 会返回相等的迭代器——都指向给定关键字的插入点，能保持容器中元素顺序的插入位置。

假定有多个元素与给定关键字匹配，`beg` 将指向其中第一个元素。我们可以通过递增 `beg` 来遍历这些元素。`end` 中的迭代器会指出何时完成遍历——当 `beg` 等于 `end` 时，就表明已经遍历了所有匹配给定关键字的元素了。

由于这两个迭代器构成一个范围，我们可以用一个 `for` 循环来遍历这个范围。循环可能执行零次，如果存在给定作者的话，就会执行多次，打印出该作者的所有项。如果给定作者不存在，`beg` 和 `end` 相等，循环就一次也不会执行。否则，我们知道递增 `beg` 最终会使它到达 `end`，在此过程中我们就会打印出与此作者关联的每条记录。



如果 `lower_bound` 和 `upper_bound` 返回相同的迭代器，则给定关键字不在容器中。

equal_range 函数

解决此问题的最后一一种方法是三种方法中最直接的：不必再调用 `upper_bound` 和

lower_bound，直接调用 equal_range 即可。此函数接受一个关键字，返回一个迭代器 pair。若关键字存在，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。若未找到匹配元素，则两个迭代器都指向关键字可以插入的位置。

可以用 equal_range 来再次修改我们的程序：

```
// authors 和 search_item 的定义，与前面的程序一样
// pos 保存迭代器对，表示与关键字匹配的元素范围
for (auto pos = authors.equal_range(search_item);
     pos.first != pos.second; ++pos.first)
    cout << pos.first->second << endl; // 打印每个题目
```

此程序本质上与前一个使用 upper_bound 和 lower_bound 的程序是一样的。不同之处就是，没有用局部变量 beg 和 end 来保存元素范围，而是使用了 equal_range 返回的 pair。此 pair 的 first 成员保存的迭代器与 lower_bound 返回的迭代器是一样的，second 保存的迭代器与 upper_bound 的返回值是一样的。因此，在此程序中，pos.first 等价于 beg，pos.second 等价于 end。

11.3.5 节练习

练习 11.27：对于什么问题你会使用 count 来解决？什么时候你又会选择 find 呢？

练习 11.28：对一个 string 到 int 的 vector 的 map，定义并初始化一个变量来保存在其上调用 find 所返回的结果。

练习 11.29：如果给定的关键字不在容器中，upper_bound、lower_bound 和 equal_range 分别会返回什么？

练习 11.30：对于本节最后一个程序中的输出表达式，解释运算对象 pos.first->second 的含义。

练习 11.31：编写程序，定义一个作者及其作品的 multimap。使用 find 在 multimap 中查找一个元素并用 erase 删除它。确保你的程序在元素不在 map 中时也能正常运行。

练习 11.32：使用上一题定义的 multimap 编写一个程序，按字典序打印作者列表和他们的作品。

11.3.6 一个单词转换的 map

我们将以一个程序结束本节的内容，它将展示 map 的创建、搜索以及遍历。这个程序的功能是这样的：给定一个 string，将它转换为另一个 string。程序的输入是两个文件。第一个文件保存的是一些规则，用来转换第二个文件中的文本。每条规则由两部分组成：一个可能出现在输入文件中的单词和一个用来替换它的短语。表达的含义是，每当第一个单词出现在输入中时，我们就将它替换为对应的短语。第二个输入文件包含要转换的文本。

如果单词转换文件的内容如下所示：

```
brb be right back
k okay?
y why
r are
```

```

u you
pic picture
thk thanks!
18r later

```

我们希望转换的文本为

```

where r u
y dont u send me a pic
k thk 18r

```

则程序应该生成这样的输出：

```

where are you
why dont you send me a picture
okay? thanks! later

```

单词转换程序

我们的程序将使用三个函数。函数 `word_transform` 管理整个过程。它接受两个 `ifstream` 参数：第一个参数应绑定到单词转换文件，第二个参数应绑定到我们要转换的文本文件。函数 `buildMap` 会读取转换规则文件，并创建一个 `map`，用于保存每个单词到其转换内容的映射。函数 `transform` 接受一个 `string`，如果存在转换规则，返回转换后的内容。

我们首先定义 `word_transform` 函数。最重要的部分是调用 `buildMap` 和 `transform`：

```

void word_transform(ifstream &map_file, ifstream &input)
{
    auto trans_map = buildMap(map_file); // 保存转换规则
    string text; // 保存输入中的每一行
    while (getline(input, text)) { // 读取一行输入
        istringstream stream(text); // 读取每个单词
        string word;
        bool firstword = true; // 控制是否打印空格
        while (stream >> word) {
            if (firstword)
                firstword = false;
            else
                cout << " "; // 在单词间打印一个空格
            // transform 返回它的第一个参数或其转换之后的形式
            cout << transform(word, trans_map); // 打印输出
        }
        cout << endl; // 完成一行的转换
    }
}

```

442 函数首先调用 `buildMap` 来生成单词转换 `map`，我们将它保存在 `trans_map` 中。函数的剩余部分处理输入文件。`while` 循环用 `getline` 一行一行地读取输入文件。这样做的目的是使得输出中的换行位置能和输入文件中一样。为了从每行中获取单词，我们使用了一个嵌套的 `while` 循环，它用一个 `istringstream`（参见 8.3 节，第 287 页）来处理当前行中的每个单词。

在输出过程中，内层 `while` 循环使用一个 `bool` 变量 `firstword` 来确定是否打印

一个空格。它通过调用 `transform` 来获得要打印的单词。`transform` 的返回值或者是 `word` 中原来的 `string`, 或者是 `trans_map` 中指出的对应的转换内容。

建立转换映射

函数 `buildMap` 读入给定文件, 建立起转换映射。

```
map<string, string> buildMap(ifstream &map_file)
{
    map<string, string> trans_map; // 保存转换规则
    string key; // 要转换的单词
    string value; // 替换后的内容
    // 读取第一个单词存入 key 中, 行中剩余内容存入 value
    while (map_file >> key && getline(map_file, value))
        if (value.size() > 1) // 检查是否有转换规则
            trans_map[key] = value.substr(1); // 跳过前导空格
        else
            throw runtime_error("no rule for " + key);
    return trans_map;
}
```

`map_file` 中的每一行对应一条规则。每条规则由一个单词和一个短语组成, 短语可能包含多个单词。我们用`>>`读取要转换的单词, 存入 `key` 中, 并调用 `getline` 读取这一行中的剩余内容存入 `value`。由于 `getline` 不会跳过前导空格 (参见 3.2.2 节, 第 78 页), 需要我们来跳过单词和它的转换内容之间的空格。在保存转换规则之前, 检查是否获得了一个以上的字符。如果是, 调用 `substr` (参见 9.5.1 节, 第 321 页) 来跳过分隔单词及其转换短语之间的前导空格, 并将得到的子字符串存入 `trans_map`。

注意, 我们使用下标运算符来添加关键字-值对。我们隐含地忽略了一个单词在转换文件中出现多次的情况。如果真的有单词出现多次, 循环会将最后一个对应短语存入 `trans_map`。当 `while` 循环结束后, `trans_map` 中将保存着用来转换输入文本的规则。

生成转换文本

函数 `transform` 进行实际的转换工作。其参数是需要转换的 `string` 的引用和转换规则 `map`。如果给定 `string` 在 `map` 中, `transform` 返回相应的短语。否则, `transform` 直接返回原 `string`:

```
const string &
transform(const string &s, const map<string, string> &m)
{
    // 实际的转换工作; 此部分是程序的核心
    auto map_it = m.find(s);
    // 如果单词在转换规则 map 中
    if (map_it != m.cend())
        return map_it->second; // 使用替换短语
    else
        return s; // 否则返回原 string
}
```

< 443

函数首先调用 `find` 来确定给定 `string` 是否在 `map` 中。如果存在, 则 `find` 返回一个指向对应元素的迭代器。否则, `find` 返回尾后迭代器。如果元素存在, 我们解引用迭代器, 获得一个保存关键字和值的 `pair` (参见 11.3 节, 第 381 页), 然后返回成员 `second`, 即

用来替代 s 的内容。

11.3.6 节练习

练习 11.33: 实现你自己版本的单词转换程序。

练习 11.34: 如果你将 transform 函数中的 find 替换为下标运算符，会发生什么情况？

练习 11.35: 在 buildMap 中，如果进行如下改写，会有什么效果？

```
trans_map[key] = value.substr(1);
改为 trans_map.insert({key, value.substr(1)})
```

练习 11.36: 我们的程序并没有检查输入文件的合法性。特别是，它假定转换规则文件中的规则都是有意义的。如果文件中的某一行包含一个关键字、一个空格，然后就结束了，会发生什么？预测程序的行为并进行验证，再与你的程序进行比较。



11.4 无序容器

新标准定义了 4 个无序关联容器（unordered associative container）。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（hash function）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，维护元素的序代价非常高昂，此时无序容器也很有用。

虽然理论上哈希技术能获得更好的平均性能，但在实际中想要达到很好的效果还需要进行一些性能测试和调优工作。因此，使用无序容器通常更为简单（通常也会有更好的性能）。

444



如果关键字类型固有就是无序的，或者性能测试发现问题可以用哈希技术解决，就可以使用无序容器。

使用无序容器

除了哈希管理操作之外，无序容器还提供了与有序容器相同的操作（find、insert 等）。这意味着我们曾用于 map 和 set 的操作也能用于 unordered_map 和 unordered_set。类似的，无序容器也有允许重复关键字的版本。

因此，通常可以用一个无序容器替换对应的有序容器，反之亦然。但是，由于元素未按顺序存储，一个使用无序容器的程序的输出（通常）会与使用有序容器的版本不同。

例如，可以用 unordered_map 重写最初的单词计数程序（参见 11.1 节，第 375 页）：

```
// 统计出现次数，但单词不会按字典序排列
unordered_map<string, size_t> word_count;
string word;
while (cin >> word)
    ++word_count[word]; // 提取并递增 word 的计数器
for (const auto &w : word_count) // 对 map 中的每个元素
    // 打印结果
    cout << w.first << " occurs " << w.second
        << ((w.second > 1) ? " times" : " time") << endl;
```

此程序与原程序的唯一区别是 `word_count` 的类型。如果在相同的输入数据上运行此版本，会得到这样的输出：

```
containers occurs 1 time
use occurs 1 time
can occurs 1 time
examples occurs 1 time
...
```

对于每个单词，我们将得到相同的计数结果。但单词不太可能按字典序输出。

管理桶

无序容器在存储上组织为一组桶，每个桶保存零个或多个元素。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，所有具有相同关键字的元素也都会在同一个桶中。因此，无序容器的性能依赖于哈希函数的质量和桶的数量和大小。

对于相同的参数，哈希函数必须总是产生相同的结果。理想情况下，哈希函数还能将每个特定的值映射到唯一的桶。但是，将不同关键字的元素映射到相同的桶也是允许的。当一个桶保存多个元素时，需要顺序搜索这些元素来查找我们想要的那个。计算一个元素的哈希值和在桶中搜索通常都是很快的操作。但是，如果一个桶中保存了很多元素，那么查找一个特定元素就需要大量比较操作。

无序容器提供了一组管理桶的函数，如表 11.8 所示。这些成员函数允许我们查询容器的状态以及在必要时强制容器进行重组。

<445

表 11.8：无序容器管理操作

桶接口	
<code>c.bucket_count()</code>	正在使用的桶的数目
<code>c.max_bucket_count()</code>	容器能容纳的最多的桶的数量
<code>c.bucket_size(n)</code>	第 n 个桶中有多少个元素
<code>c.bucket(k)</code>	关键字为 k 的元素在哪个桶中
桶迭代	
<code>local_iterator</code>	可以用来访问桶中元素的迭代器类型
<code>const_local_iterator</code>	桶迭代器的 <code>const</code> 版本
<code>c.begin(n), c.end(n)</code>	桶 n 的首元素迭代器和尾后迭代器
<code>c.cbegin(n), c.cend(n)</code>	与前两个函数类似，但返回 <code>const_local_iterator</code>
哈希策略	
<code>c.load_factor()</code>	每个桶的平均元素数量，返回 <code>float</code> 值
<code>c.max_load_factor()</code>	c 试图维护的平均桶大小，返回 <code>float</code> 值。c 会在需要时添加新的桶，以使得 <code>load_factor<=max_load_factor</code>
<code>c.rehash(n)</code>	重组存储，使得 <code>bucket_count>=n</code> 且 <code>bucket_count>size/max_load_factor</code>
<code>c.reserve(n)</code>	重组存储，使得 c 可以保存 n 个元素且不必 rehash

无序容器对关键字类型的要求

默认情况下，无序容器使用关键字类型的`==`运算符来比较元素，它们还使用一个`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`和我们将要在第 12 章介绍的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（包括指针类型）、`string`还是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。与容器不同，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。我们将在 16.5 节（第 626 页）中介绍如何做到这一点。

我们不使用默认的`hash`，而是使用另一种方法，类似于为有序容器重载关键字类型的默认比较操作（参见 11.2.2 节，第 378 页）。为了能将`Sale_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。我们从定义这些重载函数开始：

```
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();
}
```

我们的`hasher`函数使用一个标准库`hash`类型对象来计算 ISBN 成员的哈希值，该`hash`类型建立在`string`类型之上。类似的，`eqOp` 函数通过比较 ISBN 号来比较两个`Sales_data`。

我们使用这些函数来定义一个`unordered_multiset`

```
using SD_multiset = unordered_multiset<Sales_data,
                                         decltype(hasher)*, decltype(eqOp)*>;
// 参数是桶大小、哈希函数指针和相等性判断运算符指针
SD_multiset bookstore(42, hasher, eqOp);
```

为了简化`bookstore`的定义，首先为`unordered_multiset`定义了一个类型别名（参见 2.5.1 节，第 60 页），此集合的哈希和相等性判断操作与`hasher`和`eqOp` 函数有着相同类型。通过使用这种类型，在定义`bookstore`时可以将我们希望它使用的函数的指针传递给它。

如果我们的类定义了`==`运算符，则可以只重载哈希函数：

```
// 使用 FooHash 生成哈希值；Foo 必须有==运算符
unordered_set<Foo, decltype(FooHash)*> fooSet(10, FooHash);
```

11.4 节练习

练习 11.37：一个无序容器与其有序版本相比有何优势？有序版本有何优势？

练习 11.38：用`unordered_map`重写单词计数程序（参见 11.1 节，第 375 页）和单词转换程序（参见 11.3.6 节，第 391 页）。

小结

447

关联容器支持通过关键字高效查找和提取元素。对关键字的使用将关联容器和顺序容器区分开来，顺序容器中是通过位置访问元素的。

标准库定义了 8 个关联容器，每个容器

- 是一个 `map` 或是一个 `set`。`map` 保存关键字-值对；`set` 只保存关键字。
- 要求关键字唯一或不要求。
- 保持关键字有序或不保证有序。

有序容器使用比较函数来比较关键字，从而将元素按顺序存储。默认情况下，比较操作是采用关键字类型的`<`运算符。无序容器使用关键字类型的`==`运算符和一个 `hash<key_type>`类型的对象来组织元素。

允许重复关键字的容器的名字中都包含 `multi`；而使用哈希技术的容器的名字都以 `unordered` 开头。例如，`set` 是一个有序集合，其中每个关键字只可以出现一次；`unordered_multiset` 则是一个无序的关键字集合，其中关键字可以出现多次。

关联容器和顺序容器有很多共同的元素。但是，关联容器定义了一些新操作，并对一些和顺序容器和关联容器都支持的操作重新定义了含义或返回类型。操作的不同反映出关联容器使用关键字的特点。

有序容器的迭代器通过关键字有序访问容器中的元素。无论在有序容器中还是在无序容器中，具有相同关键字的元素都是相邻存储的。

术语表

关联数组（associative array） 元素通过关键字而不是位置来索引的数组。我们称这样的数组将一个关键字映射到其关联的值。

关联容器（associative container） 类型，保存对象的集合，支持通过关键字的高效查找。

hash 特殊的标准库模板，无序容器用它来管理元素的位置。

哈希函数（hash function） 将给定类型的值映射到整形 (`size_t`) 值的函数。相等的值必须映射到相同的整数；不相等的值应尽可能映射到不同整数。

key_type 关联容器定义的类型，用来保存和提取值的关键字的类型。对于一个 `map`，`key_type` 是用来索引 `map` 的类型。对于 `set`, `key_type` 和 `value_type` 是一样的。

map 关联容器类型，定义了一个关联数组。类似 `vector`, `map` 是一个类模板。但是，一个 `map` 要用两个类型来定义：关键字的类型和关联的值的类型。在一个 `map` 中，一个给定关键字只能出现一次。每个关键字关联一个特定的值。解引用一个 `map` 迭代器会生成一个 `pair`，它保存一个 `const` 关键字及其关联的值。

mapped_type 映射类型定义的类型，就是映射中关键字关联的值的类型。

multimap 关联容器类型，类似 `map`，不同之处在于，在一个 `multimap` 中，一个给定的关键字可以出现多次。`multimap` 不支持下标操作。

multiset 保存关键字的关联容器类型。在一个 `multiset` 中，一个给定关键字可以出现多次。

448

pair 类型，保存名为 `first` 和 `second` 的 `public` 数据成员。`pair` 类型是模板类型，接受两个类型参数，作为其成员的类型。

set 保存关键字的关联容器。在一个 `set` 中，一个给定的关键字只能出现一次。

严格弱序 (strict weak ordering) 关联容器所使用的关键字间的关系。在一个严格弱序中，可以比较任意两个值并确定哪个更小。若任何一个都不小于另一个，则认为两个值相等。

无序容器 (unordered container) 关联容器，用哈希技术而不是比较操作来存储和访问元素。这类容器的性能依赖于哈希函数的质量。

unordered_map 保存关键字-值对的容器，不允许重复关键字。

unordered_multimap 保存关键字-值对的容器，允许重复关键字。

unordered_multiset 保存关键字的容器，

允许重复关键字。

unordered_set 保存关键字的容器，不允许重复关键字。

value_type 容器中元素的类型。对于 `set` 和 `multiset`, `value_type` 和 `key_type` 是一样的。对于 `map` 和 `multimap`, 此类型是一个 `pair`, 其 `first` 成员类型为 `const key_type` , `second` 成员类型为 `mapped_type`。

***运算符** 解引用运算符。当应用于 `map`、`set`、`multimap` 或 `multiset` 的迭代器时，会生成一个 `value_type` 值。注意，对 `map` 和 `multimap`, `value_type` 是一个 `pair`。

[]运算符 下标运算符。只能用于 `map` 和 `unordered_map` 类型的非 `const` 对象。对于映射类型，`[]` 接受一个索引，必须是一个 `key_type` 值（或者是能转换为 `key_type` 的类型）。生成一个 `mapped_type` 值。