**STL**

目录

[第一章：STL基础 1](#_Toc22876)

[第一 C++ STL是什么，有什么用？ 1](#_Toc14207)

[一 C++ STL是什么 1](#_Toc24066)

[二 学STL能干什么？ 1](#_Toc4389)

[第二C++ STL的发展历程是怎样的？ 2](#_Toc1894)

[第三C++ STL版本有哪些？ 3](#_Toc10905)

[第四 熟练使用STL标准库是每个C++程序员的必备技能！ 4](#_Toc20556)

[第五泛型是什么，C++泛型编程又是什么？ 5](#_Toc3519)

[第六 C++ STL基本组成（6大组件+13个头文件） 6](#_Toc29812)

[第七 如何衡量一个算法的执行效率？ 7](#_Toc8985)

[第二章STL序列式容器 10](#_Toc28434)

[第一C++ STL容器是什么？ 10](#_Toc17115)

[第二迭代器是什么，C++ STL迭代器（iterator）用法详解 11](#_Toc8722)

[一迭代器类别 11](#_Toc12200)

[二迭代器的定义方式 12](#_Toc5223)

[第三C++序列式容器（STL序列式容器）是什么 14](#_Toc20501)

[一容器中常见的函数成员 15](#_Toc26619)

[第四C++ array(STL array)容器用法详解 19](#_Toc25958)

[第五C++ STL array随机访问迭代器 21](#_Toc27644)

[一begin()/end() 和 cbegin()/cend() 22](#_Toc12584)

[二rbegin()/rend() 和 crbegin()/crend() 24](#_Toc5773)

[第六C++ STL array容器访问元素的几种方式 25](#_Toc18573)

[一访问array容器中单个元素 25](#_Toc18742)

[二访问array容器中多个元素 27](#_Toc2996)

[第七C++ array容器：普通数组的“升级版” 28](#_Toc7547)

[第八C++ STL vector容器详解 31](#_Toc10321)

[一创建vector容器的几种方式 31](#_Toc9091)

[二vector容器包含的成员函数 32](#_Toc2623)

[第九C++ STL vector容器迭代器用法详解 34](#_Toc2754)

[一vector容器迭代器的基本用法 35](#_Toc29194)

[二vector容器迭代器的独特之处 37](#_Toc11503)

[第十C++ STL vector容器访问元素的几种方式 39](#_Toc26745)

[一访问vector容器中单个元素 39](#_Toc30174)

[二 访问vector容器中多个元素 41](#_Toc16276)

[第十一 C++ vector容量（capacity）和大小（size）的区别 42](#_Toc14066)

[一 修改vector容器的容量和大小 44](#_Toc5316)

[二 vector容器容量和大小的数据类型 45](#_Toc11069)

[第十二 深度剖析C++ vector容器的底层实现机制 45](#_Toc9462)

[一 vector扩大容量的本质 46](#_Toc22924)

[第十三C++ STL vector添加元素（push\_back()和emplace\_back()）详解 47](#_Toc2803)

[一push\_back() 47](#_Toc11252)

[二emplace\_back() 47](#_Toc323)

[三 emplace\_back()和push\_back()的区别 48](#_Toc4244)

[第十四 C++ STL vector插入元素（insert()和emplace()）详解 49](#_Toc19538)

[一 insert() 49](#_Toc21937)

[二 emplace() 50](#_Toc22206)

[第十五C++ STL vector删除元素的几种方式 52](#_Toc17230)

[第十六 如何避免vector容器进行不必要的扩容？ 57](#_Toc6375)

[第十七vector swap()成员方法还可以这样用！ 59](#_Toc18160)

[一利用swap()方法去除vector多余容量 60](#_Toc28225)

[第十八切忌，vector<bool>不是存储bool类型元素的vector容器！ 62](#_Toc27195)

[一vector<bool>不是容器 62](#_Toc19960)

[二如何避免使用vector<bool> 63](#_Toc21362)

[第十九C++ STL deque容器（详解版） 64](#_Toc18912)

[一创建deque容器的几种方式 64](#_Toc18229)

[二deque容器可利用的成员函数 65](#_Toc3534)

[第二十 C++ STL deque容器迭代器用法详解 67](#_Toc7403)

[一deque容器迭代器的基本用法 68](#_Toc10953)

[二deque容器迭代器的使用注意事项 69](#_Toc25531)

[第二十一 深度剖析deque容器底层实现原理 70](#_Toc5304)

[一deque容器的存储结构 70](#_Toc9212)

[二deque容器迭代器的底层实现 71](#_Toc11886)

[三deque容器的底层实现 72](#_Toc18825)

[第二十二C++ STL deque容器访问元素（4种方法） 74](#_Toc24402)

[第二十三 C++ STL deque容器添加和删除元素方法完全攻略 76](#_Toc25952)

[第二十四C++ list（STL list）容器完全攻略（超级详细） 80](#_Toc18151)

[一 list容器的创建 81](#_Toc10603)

[二list容器可用的成员函数 82](#_Toc1542)

[第二十五C++ STL list迭代器及用法 83](#_Toc22378)

[第二十六C++ list容器底层存储结构 86](#_Toc4753)

[一 list容器迭代器的底层实现 87](#_Toc25684)

[二list容器的底层实现 88](#_Toc29140)

[第二十七C++ list（STL list）访问元素的几种方法 89](#_Toc23471)

[第二十八C++ STL list添加（插入）元素方法详解 90](#_Toc22268)

[一list insert()成员方法 91](#_Toc10922)

[二list splice()成员方法 92](#_Toc20047)

[第二十九empty()和size()都可以判断容器是否为空，谁更好？ 94](#_Toc30976)

[一深度剖析选用empty()的原因 94](#_Toc8964)

[第三十C++ STL list删除元素详解 95](#_Toc9701)

[第三十一 C++ STL forward\_list容器完全攻略 99](#_Toc23784)

[一 forward\_list容器的创建 100](#_Toc29083)

[二 forward\_list容器支持的成员函数 100](#_Toc23613)

[三 和使用forward\_list容器相关的函数 102](#_Toc24664)

[第三章 STL关联式容器 103](#_Toc3439)

[第一 C++ STL关联式容器是什么？ 103](#_Toc2465)

[一 C++ STL关联式容器是什么 103](#_Toc15253)

[二 C++ STL关联式容器种类 104](#_Toc24398)

[第二 C++ STL pair用法详解 105](#_Toc32727)

[第三 C++ STL map容器详解 108](#_Toc4555)

[一 创建C++ map容器的几种方法 109](#_Toc32052)

[二C++ map容器包含的成员方法 110](#_Toc9049)

[第四 C++ STL map容器迭代器用法详解 112](#_Toc21306)

[第五 C++ STL map获取键对应值的几种方法 117](#_Toc2649)

[第六 C++ STL map insert()插入数据的4种方式 120](#_Toc18343)

[第七 C++ map容器operator[]和insert()效率对比（深度剖析） 125](#_Toc743)

[一 向map容器中增添元素，insert()效率更高 126](#_Toc26140)

[二 更新map容器中的键值对，operator[]效率更高 127](#_Toc29260)

[第八 C++ STL map emplace()和emplace\_hint() 127](#_Toc3646)

[第九 C++ map容器3种插入键值对的方法，谁的效率更高？ 129](#_Toc4886)

[第十 C++ STL multimap容器用法完全攻略 131](#_Toc27636)

[一 创建C++ multimap容器的方法 131](#_Toc5845)

[二 C++ multimap容器包含的成员方法 133](#_Toc29362)

[第十一 C++ STL set容器完全攻略 135](#_Toc4810)

[一 创建C++ set容器的几种方法 136](#_Toc31721)

[二 C++ STL set容器包含的成员方法 138](#_Toc30477)

[第十二 C++ STL set容器迭代器用法详解 140](#_Toc27147)

[第十三 C++ STL set insert()方法详解 143](#_Toc3068)

[第十四 C++ STL set emplace()和emplace\_hint() 146](#_Toc31961)

[第十五C++ STL set删除数据：erase()和clear()方法 148](#_Toc4070)

[第十六 C++ STL multiset容器详解 150](#_Toc10433)

[一 创建C++ multiset容器的方法 150](#_Toc8021)

[二 C++ multiset容器提供的成员方法 152](#_Toc4767)

[第十七 如何自定义C++ STL关联式容器的排序规则？ 154](#_Toc9314)

[一  使用函数对象自定义排序规则 154](#_Toc6690)

[二 重载关系运算符实现自定义排序 156](#_Toc11771)

[第十八 如何修改关联式容器中键值对的键？ 158](#_Toc28822)

[第四章 C++ STL无序关联式容器 162](#_Toc22019)

[第一C++ STL无序容器种类 162](#_Toc26355)

[第二C++ STL unordered\_map容器用法详解 163](#_Toc21931)

[一创建C++ unordered\_map容器的方法 165](#_Toc13657)

[二C++ unordered\_map容器的成员方法 165](#_Toc8578)

[第三深度剖析C++无序容器的底层实现机制 168](#_Toc20314)

[第四C++ unordered\_map迭代器的用法 171](#_Toc29448)

[第五C++ STL unordered\_map获取元素的4种方法 173](#_Toc13261)

[第六C++ unordered\_map insert()用法精讲 176](#_Toc5060)

[第七C++ unordered\_map emplace()和emplace\_hint()方法 180](#_Toc23548)

[一unordered\_map emplace()方法 180](#_Toc7730)

[二unordered\_map emplace\_hint()方法 181](#_Toc19810)

[第八C++ STL unordered\_map删除元素：erase()和clear() 182](#_Toc19209)

[一unordered\_map erase()方法 182](#_Toc10850)

[二unordered\_map clear()方法 185](#_Toc21915)

[第九C++ STL unordered\_multimap容器精讲 186](#_Toc16789)

[一创建C++ unordered\_multimap容器的方法 187](#_Toc6247)

[二C++ unordered\_multimap容器的成员方法 188](#_Toc22689)

[第十C++ STL unordered\_set容器完全攻略 190](#_Toc19222)

[一创建C++ unordered\_set容器 191](#_Toc5818)

[二C++ unordered\_set容器的成员方法 192](#_Toc21879)

[第十一C++ STL unordered\_multiset容器详解 194](#_Toc21562)

[一创建C++ unordered\_multiset容器 195](#_Toc1841)

[二C++ unordered\_multimap容器的成员方法 196](#_Toc32760)

[第十二如何自定义C++ STL无序容器的哈希函数和比较规则？ 199](#_Toc16974)

[一C++无序容器自定义哈希函数 199](#_Toc4159)

[二C++无序容器自定义比较规则 200](#_Toc22213)

[1) 重载==运算符 200](#_Toc25749)

[2) 以函数对象类的方式自定义比较规则 201](#_Toc32686)

[第十三C++ STL容器这么多，怎样选出最适合的？ 203](#_Toc26295)

[第五章 C++（STL）容器适配器 204](#_Toc14027)

[第一什么是适配器，C++ STL容器适配器详解 204](#_Toc27763)

[一STL容器适配器的种类 205](#_Toc31102)

[第二 C++ stack（STL stack）容器适配器用法详解 206](#_Toc10564)

[一stack容器适配器的创建 207](#_Toc12913)

[二stack容器适配器支持的成员函数 208](#_Toc13787)

[第三stack容器适配器实现计算器（含实现代码） 209](#_Toc21880)

[一 计算器程序的实现流程 212](#_Toc9475)

[第四C++ STL queue容器适配器详解 213](#_Toc26478)

[一queue容器适配器的创建 213](#_Toc6266)

[第五C++ queue容器适配器模拟超市结账环节 215](#_Toc14882)

[第六C++ STL priority\_queue容器适配器详解 220](#_Toc7604)

[一创建priority\_queue的几种方式 221](#_Toc9362)

[第七 priority\_queue容器适配器实现自定义排序 223](#_Toc6569)

[第八深度剖析priority\_queue容器的底层实现 227](#_Toc511)

[一priority\_queue底层的堆存储结构 227](#_Toc1051)

[第六章 C++ STL迭代器适配器完全攻略 230](#_Toc22177)

[第一C++ STL迭代器适配器是什么？ 230](#_Toc30194)

[一 C++ STL迭代器适配器种类 231](#_Toc12038)

[第二 C++ STL 反向迭代器适配器（reverse\_iterator）详解 232](#_Toc27557)

[一 C++ STL反向迭代器的创建 232](#_Toc14177)

[二C++ STL reverse\_iterator模板类中的成员 233](#_Toc25551)

[第三 C++ STL插入迭代器适配器（insert\_iterator） 235](#_Toc12883)

[一 C++ STL back\_insert\_iterator迭代器 236](#_Toc847)

[二 C++ STL front\_insert\_iterator迭代器 237](#_Toc29426)

[三 C++ STL insert\_iterator迭代器 238](#_Toc29620)

[第四 C++ STL流迭代器(istream\_iterator和ostream\_iterator) 239](#_Toc23581)

[一 C++ STL输入流迭代器（istream\_iterator） 240](#_Toc5572)

[二 C++ STL输出流迭代器（ostream\_iterator） 241](#_Toc14174)

[第五 C++ STL流缓冲区迭代器（streambuf\_iterator） 243](#_Toc12624)

[一 C++ STL输入流缓冲区迭代器（istreambuf\_iterator） 243](#_Toc2024)

[二 C++ STL输出流缓冲区迭代器（ostreambuf\_iterator） 244](#_Toc21701)

[第六 C++ STL move\_iterator移动迭代器用法详解 245](#_Toc25201)

[一 C++ STL move\_iterator的创建 247](#_Toc27153)

[第七 C++ STL advance()函数用法详解 249](#_Toc22779)

[一 C++ STL advance()函数 250](#_Toc29646)

[第八 C++ STL distance()函数用法详解 252](#_Toc16843)

[第九 如何将const\_iterator转换为iterator类型迭代器？ 252](#_Toc15590)

[第十 C++ STL begin()和end()函数用法 255](#_Toc6933)

[一begin()和end()参数为容器 255](#_Toc22007)

[二 begin()和end()参数为数组 256](#_Toc6451)

[第十一 C++ STL prev()和next()函数用法详解 257](#_Toc22344)

[一 C++ STL prev()函数 257](#_Toc7004)

[二 C++ STL next()函数 258](#_Toc21272)

[第七章 C++ STL常用算法（排序、合并、搜索和分区） 259](#_Toc25390)

[第一 C++ sort()排序函数用法详解 259](#_Toc891)

[一 C++ sort()排序函数 260](#_Toc3478)

[第二 C++ stable\_sort()用法详解 262](#_Toc22576)

[第三 C++ partial\_sort()函数详解 264](#_Toc32720)

[一 C++ partial\_sort()排序函数 264](#_Toc27665)

[二 C++ partial\_sort\_copy()排序函数 266](#_Toc20891)

[第四 C++ nth\_element()用法详解 267](#_Toc26376)

[第五 C++ is\_sorted()函数完全攻略 270](#_Toc17875)

[一 C++ is\_sorted()函数 270](#_Toc14513)

[二 C++ is\_sorted\_until()函数 271](#_Toc11530)

[第六 C++ STL标准库这么多排序函数，该如何选择？ 273](#_Toc14611)

[第七 自定义STL算法规则，应优先使用函数对象！ 274](#_Toc10726)

[第八 C++ merge()和inplace\_merge()函数用法（详解版） 276](#_Toc26646)

[一 C++ merge()函数 276](#_Toc19772)

[二 C++ inplace\_merge()函数 278](#_Toc2457)

[第九 C++ find()函数用法详解 278](#_Toc28743)

[第十 能用STL算法，绝不自己实现！ 280](#_Toc2906)

[一 为什么STL算法效率更高 281](#_Toc9590)

[第十一 STL算法和容器中的成员方法同名时，该如何选择？ 282](#_Toc19220)

[第十二 C++ find\_if()和find\_if\_not()函数用法详解 283](#_Toc7316)

[一 C++ find\_if()函数 284](#_Toc18037)

[二 C++ find\_if\_not()函数 285](#_Toc32099)

[第十三 C++ find\_end()函数详解 286](#_Toc19419)

[第十四 C++ find\_first\_of()函数完全攻略 289](#_Toc5087)

[第十五 C++ adjacent\_find()函数用法详解 291](#_Toc14849)

[第十六 C++ search()函数用法完全攻略 293](#_Toc17345)

[第十七 C++ search\_n()函数用法 295](#_Toc3581)

[第十八 C++ partition()和stable\_partition()函数详解 298](#_Toc9472)

[一 C++ stable\_partition()函数 299](#_Toc14912)

[第十九 C++ partition\_copy()函数详解 300](#_Toc31066)

[第二十 C++ partition\_point()函数 303](#_Toc22128)

[第二十一 C++ lower\_bound()函数用法详解 305](#_Toc27439)

[一 C++ lower\_bound()函数 305](#_Toc30559)

[第二十二 C++ upper\_bound()函数 307](#_Toc18486)

[第二十三 C++ equel\_range()函数详解 308](#_Toc14187)

[第二十四 C++ binary\_search()函数详解 310](#_Toc16082)

[第二十五 C++(STL) all\_of、any\_of及none\_of算法详解 312](#_Toc31484)

[第二十六 C++ equal(STL equal)比较算法详解 313](#_Toc6927)

[第二十七 C++ mismatch(STL mismatch)算法详解 316](#_Toc6672)

[第二十八 C++（STL） lexicographical\_compare字符串排序算法详解 319](#_Toc10387)

[第二十九 C++ next\_permutation(STL next\_permutation)算法详解 320](#_Toc7911)

[第三十 C++ prev\_permutation(STL prev\_permutation)算法详解 322](#_Toc18693)

[第三十一 C++ is\_permutation（STL is\_permutation）算法详解 323](#_Toc2894)

[第三十二 C++ copy\_n(STL copy\_n)算法详解 324](#_Toc2742)

[第三十三 C++ copy\_if(STL copy\_if)算法详解 324](#_Toc24869)

[第三十四 C++ copy\_backward(STL copy\_backward)算法详解 325](#_Toc10643)

[第三十五 C++ reverse\_copy(STL reverse\_copy)算法详解 327](#_Toc20486)

[第三十六 C++ unique(STL unique)算法详解 328](#_Toc29865)

[第三十七 C++ rotate(STL rotate)算法详解 329](#_Toc14651)

[第三十八 C++ rotate\_copy（STL rotate\_copy）算法详解 330](#_Toc9410)

[第三十九 C++ move(STL move)函数使用详解 331](#_Toc1132)

[第四十 C++ swap\_ranges(STL swap\_ranges)函数使用详解 332](#_Toc21725)

[第四十一 C++ remove、remove\_copy、remove\_if和remove\_copy\_if函数使用详解 332](#_Toc21161)

[第四十二 C++ fill和fill\_n函数用法详解 334](#_Toc5020)

[第四十三 C++（STL）generate和generate\_n函数用法详解 335](#_Toc19121)

[第四十四 C++ transform(STL transform)函数用法详解 336](#_Toc31957)

[第四十五 C++ replace,replace\_if和replace\_copy函数用法详解 339](#_Toc20660)

**第一章：STL基础**

**第一 C++ STL是什么，有什么用？**

**一 C++ STL是什么**

在已有 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 尤其是 C++ 模板的基础上，从本节开始，我们开始系统地学习 [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 标准模板库，首先来了解什么是 STL，以及学习 STL 有什么用？

STL，英文全称 s[tan](http://c.biancheng.net/ref/tan.html" \t "http://c.biancheng.net/view/_blank)dard template library，中文可译为标准模板库或者泛型库，其包含有大量的模板类和模板函数，是 C++ 提供的一个基础模板的集合，用于完成诸如输入/输出、数学计算等功能。

STL 最初由惠普实验室开发，于 1998 年被定为国际标准，正式成为 C++ 程序库的重要组成部分。值得一提的是，如今 STL 已完全被内置到支持 C++ 的编译器中，无需额外安装，这可能也是 STL 被广泛使用的原因之一。

从根本上说，STL 是一些容器、算法和其他一些组件的集合，所有容器和算法都是总结了几十年来算法和[数据结构](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)的研究成果，汇集了许多计算机专家学者经验的基础上实现的，因此可以说，STL 基本上达到了各种存储方法和相关算法的高度优化。

注意，这里提到的容器，本质上就是封装有数据结构的模板类，例如 list、vector、set、map 等，有关这些容器的具体用法，后续章节会做详细介绍。

**二 学STL能干什么？**

为了让读者清楚地了解 STL 是什么，使用 STL 编程有哪些优势，这里举一个使用 STL 的例子。  
以 C++ 定义数组的操作为例，在 C++ 中如果定义一个数组，可以采用如下方式：

int a[n];

这种定义数组的方法需要事先确定好数组的长度，即 n 必须为常量，这意味着，如果在实际应用中无法确定数组长度，则一般会将数组长度设为可能的最大值，但这极有可能导致存储空间的浪费。

所以除此之外，还可以采用在堆空间中动态申请内存的方法，此时长度可以是变量：

int \*p = new int[n];

这种定义方式可根据变量 n 动态申请内存，不会出现存储空间浪费的问题。但是，如果程序执行过程中出现空间不足的情况时，则需要加大存储空间，此时需要进行如下操作：

**（1）新申请一个较大的内存空间，即执行int \* temp = new int[m];**

**（2）将原内存空间的数据全部复制到新申请的内存空间中，即执行memecpy(temp, p, sizeof(int)\*n);**

**（3）将原来的堆空间释放，即执行delete [] p; p = temp;**

而完成相同的操作，如果采用 STL 标准库，则会简单很多，因为大多数操作细节将不需要程序员关心。下面是使用向量模板类 vector 实现以上功能的示例：

vector <int> a; //定义 a 数组，当前数组长度为 0，但和普通数组不同的是，此数组 a 可以根据存储数据的数量自动变长。

//向数组 a 中添加 10 个元素

for (int i = 0; i < 10; i++)

a.push\_back(i)

//还可以手动调整数组 a 的大小

a.resize(100);

a[90] = 100;

//还可以直接删除数组 a 中所有的元素，此时 a 的长度变为 0

a.clear();

//重新调整 a 的大小为 20，并存储 20 个 -1 元素。

a.resize(20, -1)

对比以上两种使用数组的方式不难看出，使用 STL 可以更加方便灵活地处理数据。所以，大家只需要系统地学习 STL，便可以集中精力去实现程序的功能，而无需再纠结某些细节如何用代码实现。

**第二C++ STL的发展历程是怎样的？**

Alexander Stepanov（后被誉为 [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 标准模板库之父，后简称 Stepanov），1950 年出生与前苏联的莫斯科，他曾在莫斯科大学研究数学，此后一直致力于计算机语言和泛型库研究。  
 在 20 世纪 70 年代，Stepanov 开始考虑，在保证效率的前提下，是否能将算法从诸多具体应用之中抽象出来？为了验证自己的思想，他和纽约州立大学教授 Deepak Kapur 以及伦塞里尔技术学院教授 David Musser 共同开发了一种叫做 Tecton 的语言，尽管这次尝试没有取得实用性的成果，但却给了 Stepanov 很大的启示。  
 在随后的几年中，他又和 David Musser 等人先后用 Schema 语言（一种 Lisp 语言的变种）和 Ada 语言建立了一些大型程序库。Stepanov 逐渐意识到，在当时的面向对象程序设计思想中存在一些问题，比如抽象数据类型概念所存在的缺陷，他希望通过对软件领域中各组成部分的分类，逐渐形成一种软件设计的概念性框架。  
 1987 年，在贝尔实验室工作的 Stepanov 开始首次采用 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 语言进行泛型软件库的研究。由于当时的 C++ 语言还没有引入模板的编程技术，泛型库只能是通过 C++ 的继承机制来开发，代码表达起来非常笨拙。  
 但尽管如此，Stepanov 还是开发出了一个庞大的算法库。与此同时，在与 Andrew Koenig（前 ISO C++ 标准化委员会主席）和 Bjarne Stroustrup（C++ 语言的创始人）等顶级大师们的共事过程中，Stepanov 开始注意到 C/C++ 语言在实现其泛型思想方面所具有的潜在优势。  
 就拿 C/C++ 中的[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)而言，它的灵活与高效运用使后来的 STL 在实现泛型化的同时更是保持了高效率。另外，在 STL 中占据极其重要地位的迭代器概念便是源自于 C/C++ 中原生指针的一般化推广。  
 1988 年，Stepanov 开始进入惠普的 Palo Alto 实验室工作，在随后的 4 年中，他从事的是有关磁盘驱动器方面的工作。直到 1992 年，由于参加并主持了实验室主任 Bill Worley 所建立的一个有关算法的研究项目，才使他重新回到了泛型化算法的研究工作上来。  
 项目自建立之后，参与者从最初的 8 人逐渐减少，最后只剩下 Stepanov 和 Meng Lee 两个人。经过长时间的努力，最终完成了一个包含有大量[数据结构](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)和算法部件的庞大运行库（HP 版本的 C++ STL），这便是现在 STL 的雏形。  
 1993 年，当时在贝尔实验室的 Andrew Koenig 看到了 Stepanov 的研究成果，在他的鼓励与帮助下，Stepanov 于 1993 年 9 月在圣何塞为 ANSI/ISO C++ 标准委员会做了一个题为“The Science of C++ Programming” 的演讲，向委员们讲述了其观念。然后又于 1994 年 3 月，在圣迭戈会议上向委员会提交了一份建议书，以期将 STL 通用库纳入 C++ 标准。  
 尽管这一建议十分庞大，以至于降低了被通过的可能性，但其所包含的新思想吸引了许多人的注意力。随后在众人的帮助之下，包括 Bjame Stroustrup 在内，Stepanov 又对 STL 进行了改进，同时加入了一个封装内存模式信息的抽象模块，也就是现在 STL 中的 allocator（内存分配器），它使 STL 的大部分实现都可以独立于具体的内存模式，从而独立于具体平台。  
 最终在 1994 年的滑铁卢会议上，委员们通过了提案，决定将 STL 正式纳入 C++ 标准化进程之中，随后 STL 便被放进了会议的工作文件中。自此，STL 终于成为 C++ 家族中的重要一员。  
 此后，随者 C++ 标准的不断改进，STL 也在不断地做着相应的演化。直至 1998 年，ANSI/ISO C++ 标准正式定案，STL 始终是 C++ 标准库不可或缺的重要组成部分。

**第三C++ STL版本有哪些？**

自 1998 年 ANSI/ISO [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 标准正式定案，C++ [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 规范版本正式通过以后，由于其实开源的，各个 C++ 编译器厂商在此标准的基础上，实现了满足自己需求的 C++ STL 泛型库，主要包括 HP STL、SGI STL、STLport、PJ STL、Rouge Wave STL 等。

**（1）HP STL**

HP STL 是 Alexandar Stepanov（STL 标准模板库之父，文章后续简称 Stepanov）在惠普 Palo Alto 实验室工作时，与 Meng Lee 合作完成的。HP STL 是开放源码的，即任何人都可以免费使用、复制、修改、发布和销售该软件以及相关文档，但前提是必须在相关文档中，加入 HP STL 版本信息和授权信息。  
 HP STL 是 C++ STL 的第一个实现版本，其它版本的 C++ STL 一般是以 HP STL 为蓝本实现出来的。不过，现在已经很少直接使用此版本的 STL 了。

**（2）SGI STL**

Stepanov 在离开 HP 之后，就加入到了 SGI 公司，并和 Matt Austern 等人开发了 SGI STL。严格意义上来说，它是 HP STL 的一个继承版本。和 HP STL 一样，SGI STL 也是开源的，其源代码的可读性可非常好，并且任何人都可以修改和销售它。  
 注意，和 STL 官方版本来说，SGI STL 只能算是一个“民间”版本，因此并不是所有支持 C++ 的编译器都支持使用 SGI STL 模板库，唯一能确定的是，[GCC](http://c.biancheng.net/gcc/" \t "http://c.biancheng.net/view/_blank)（Linux 下的 C++ 编译器）是支持的，所以 SGI STL 在 Linux 平台上的性能非常出色。

**（3）STLport**

为了使 SGI STL 的基本代码都适用于 VC++ 和 C++ Builder 等多种编译器，俄国人 Boris Fomitchev 建立了一个 free 项目来开发 STLport，此版本 STL 是开放源码的。

**（4）PJ STL**

PJ STL（全称为 P.J. Plauger STL）是由 P.J.Plauger（美国人，1965 年毕业于普林斯顿大学，物理专业学士）参照 HP STL 实现出来的，也是 HP STL 的一个继承版本，因此该头文件中不仅含有 HP STL 的相关授权信息，同时还有 P.J.Plauger 本人的版权信息。

其实 PJ STL 是 P.J.Plauger 公司的产品，尽管该公司当时只有 3 个人。

PJ STL 被 Visual C++ 编译器所采用，但和 PH STL、SGI STL 不同的是，PJ STL 并不是开源。

**（5）Rouge Wave STL**

该版本的 STL 是由 Rouge Wave 公司开发的，也是继承 HP STL 的一个版本，它也不是开源的。  
  
 Rouge Wave STL 用于 Borland C++ Builder 编译器中，我们可以在 C++ Builder 的 Inculde 子目录中找到该 STL 的所有头文件。  
 值得一提的是，尽管 Rouge Wave STL 的性能不是很好，但 C++ Builder 对 C++ 语言标准的支持还算不错，所以在一定程度上使 Rouge Wave STL 的表现得以改善。  
 遗憾的是，由于 Rouge Wave STL 长期没有更新且不完全符合标准，因此 Rouge Wave STL 在 6.0 版本时改用了 STLport 版本（之后的版本也都采用了 STLport），不过考虑到和之前版本的兼容，6.0 版本中依旧保留了 Rouge Wave STL。

Rouge Wave 公司在 C++ 程序库领域应该说是鼎鼎大名，对 C++ 标准化的过程出力甚多。不过 Rouge Wave STL 版本不仅更新频率慢，费用还高，基于这两个原因，Borland 在 6.0 版本决定弃用 Rouge Wave STL 而改用 STLport。

**第四 熟练使用STL标准库是每个C++程序员的必备技能！**

C++ 标准程序库发展至今，几乎所有内容都被设计为了模板的形式，STL 已经成为 C++ 程序库的重要组成部分。可以这么说，如果 C++ 不支持 STL 标准模板库，就无法使用程序库。那么，C++ 为什么要引入 STL 呢？  
 在大多数人看来，计算机既神秘有能干，但在程序员的眼中，计算机又蠢又笨，唯一的优点就是运算速度比人快，不给指令什么都干不了，就是给指令，计算机也不能灵活运用。  
比如说，在 C++ 中，同样一个加法，不同的数据类型，要给出不同的运行代码：

#include<iostream>

using namespace *std*;

//处理整形之间的加法

int addInt(int m, int n) {

return m + n;

}

//处理浮点类型值之间的加法

double addDouble(double i, double j) {

return i + j;

}

//......

int main()

{

*cout* << addInt(1, 2) << *endl* << addDouble(1.2, 2.1);

return 0;

}

**运行结果为：**

**3  
3.3**

像这样，对于每一种数据类型，我们都必须给计算机设计一个单独的函数，实在太繁琐了。由此可以感受到，计算机并不具备人类的基本思维，处理问题不灵活。  
 为了让计算机不断接近人类的认知能力，科学家们想了很多办法，比如使用面向对象开发技术，通过类的封装和函数重载，可以部分解决上面的问题：

#include<iostream>

using namespace *std*;

class calc {

public:

//处理整形之间的加法

int add(int m, int n) {

return m + n;

}

//处理浮点类型值之间的加法

double add(double i, double j) {

return i + j;

}

//......

};

int main()

{

calc a;

*cout* << a.add(1, 2) << *endl* << a.add(1.2, 2.1);

return 0;

}

创建这样的类之后，当通过类对象调用 add 方法时，就无需考虑参数的具体数据类型了。但从某种程序上来说，这也仅是让计算机聪明了一点点。  
 为了让程序更加智能、人性化，经过科学家们持续的努力，C++ 引入了模板这个功能。模板可以认为是针对一个或多个尚未明确的类型而编写的一个个函数，是 C++ 的一个新特性。  
 通过引入模板，C++ 引申出了泛型编程技术。简单的理解泛型编程，即使用该技术编写的代码，可以支持多种数据类型。也就是说，通过泛型编程，能编写出可重复利用的程序代码，并且其运行效率和针对某特定数据类型而设计的代码相同。由此可见，C++ 很需要泛型这种新的编程模式，可以减轻编程的工作量，增强代码的重用性。

在 C++ 支持模板功能，引入了泛型编程思想的基础上，C++ 程序员们想编写出很多通用的针对不同数据类型的算法，其中 STL 脱颖而出成为 C++ 标准，并被引入 C++ 标准程序库。  
 STL 是一个具有高度可用性、高效的模板库，该库包含了诸多在计算机科学领域中常用的基础数据结构和算法，掌握了 STL 标准，很多功能就无需自己费心费力的去实现了（不用重复的造轮子），直接拿来用即可。  
 总的来说，STL 模板库是 C++ 标准程序库的重要组成部分，为 C++ 程序员提供了大量的可扩展的程序框架，高度实现了代码的可重用性，并且它是内置的，不需要额外安装，使用非常方便。

**第五泛型是什么，C++泛型编程又是什么？**

在计算机程序设计领域，为了避免因数据类型的不同，而被迫重复编写大量相同业务逻辑的代码，人们发展的泛型及泛型编程技术。  
 那么，什么是泛型呢？本节就带领读者深度剖析一下这个问题。  
 所以泛型，实质上就是不使用具体数据类型（例如 int、double、float 等），而是使用一种通用类型来进行程序设计的方法，该方法可以大规模的减少程序代码的编写量，让程序员可以集中精力用于业务逻辑的实现。  
 为了更好地说明使用具体数据类型有多么麻烦，这里先举个例子，假设客户需要一个函数，功能是返回两个 int 类型数据中较大的那个，很多读者自然而然会编写如下代码：

int maxt(int x, int y) {

return (x > y) ? x : y;

}

可是没过几天，该用户又提出需要编写一个返回两个 double 类型数据中较大的那个，于是我们需要之前的代码进行修改：

double maxt(double x, double y) {

return (x > y) ? x : y;

}

之后，该用户又提出需要再编写一个能返回两个 char 类型数据中较大的那个...。可以看到，只是因为数据类型不同，就迫使我们不得不把具有相同功能的代码写了若干遍，这样的实现方法简直令人崩溃。  
 为了解决类似的问题，有聪明的人将代码修改成如下的样子：

**T maxt(T x, T y) {**

**return (x > y) ? x : y;**

**}**

如此，当用户需要某个数据类型的 maxt 函数时，我们只需要把其中的 T 替换成用户需要的实际数据类型就行了。  
 那么，代码中的 T 是什么呢？很明显，这是一个占位符，更确切的说是一个类型占位符。也就是说，将来在 T 这个位置上的是一个真实、具体的数据类型，至于到底是哪个类型，完全取决于用户的需求。  
 当然，如果硬要给 T 这种类型占位符也叫做一种数据类型，提供这种想法的发明者称它为泛型（generic type），而使用这种类型占位符的编程方式就被称为泛型编程。  
 值得一提的是，既然泛型并不是真实的数据类型，那么使用泛型编写的代码也就不是真正的程序实体，只能算是一个程序实体的样板。故此，通常形象的将这种使用了泛型的代码称为模板，由模板生成实际代码的过程称为模板的具体实现。

**注意，类型占位符的替换工作，不需要人为操控，可以完全交由计算机来完成，更准确的说，是交由编译器在编译阶段来完成模板的具体实现。**

总之一句话，泛型也是一种数据类型，只不过它是一种用来代替所有类型的“通用类型”。在 C++ 中，用以支持泛型应用的就是标准模板库 STL，它提供了 C++ 泛型设计常用的类模板和函数模板。

**第六 C++ STL基本组成（6大组件+13个头文件）**

通常认为，[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 是由容器、算法、迭代器、函数对象、适配器、内存分配器这 6 部分构成，其中后面 4 部分是为前 2 部分服务的，它们各自的含义如表 1 所示。

表 1 STL 组成结构

|  |  |
| --- | --- |
| STL的组成 | 含义 |
| 容器 | 一些封装[数据结构](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)的模板类，例如 vector 向量容器、list 列表容器等。 |
| 算法 | STL 提供了非常多（大约 100 个）的数据结构算法，它们都被设计成一个个的模板函数，这些算法在 std 命名空间中定义，其中大部分算法都包含在头文件 <algorithm> 中，少部分位于头文件 <numeric> 中。 |
| 迭代器 | 在 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) STL 中，对容器中数据的读和写，是通过迭代器完成的，扮演着容器和算法之间的胶合剂。 |
| 函数对象 | 如果一个类将 () 运算符重载为成员函数，这个类就称为函数对象类，这个类的对象就是函数对象（又称仿函数）。 |
| 适配器 | 可以使一个类的接口（模板的参数）适配成用户指定的形式，从而让原本不能在一起工作的两个类工作在一起。值得一提的是，容器、迭代器和函数都有适配器。 |
| 内存分配器 | 为容器类模板提供自定义的内存申请和释放功能，由于往往只有高级用户才有改变内存分配策略的需求，因此内存分配器对于一般用户来说，并不常用。 |

另外，在惠普实验室最初发行的版本中，STL 被组织成 48 个头文件；但在 C++ 标准中，它们被重新组织为 13 个头文件，如表 2 所示。

表 2 C++ STL头文件

|  |  |  |  |
| --- | --- | --- | --- |
| <iterator> | <functional> | <vector> | <deque> |
| <list> | <queue> | <stack> | <set> |
| <map> | <algorithm> | <numeric> | <memory> |
| <utility> |  |  |  |

按照 C++ 标准库的规定，所有标准头文件都不再有扩展名。以 <vector> 为例，此为无扩展名的形式，而 <vector.h> 为有扩展名的形式。  
 但是，或许是为了向下兼容，或许是为了内部组织规划，某些 STL 版本同时存储具备扩展名和无扩展名的两份文件（例如 Visual C++ 支持的 Dinkumware 版本同时具备 <vector.h> 和 <vector>）；甚至有些 STL 版本同时拥有 3 种形式的头文件（例如 SGI 版本同时拥有 <vector>、<vector.h> 和 <stl\_vector.h>）；但也有个别的 STL 版本只存在包含扩展名的头文件（例如 C++ Builder 的 RaugeWare 版本只有 <vector.h>）。

**建议读者养成良好的习惯，遵照 C++ 规范，使用无扩展名的头文件**。

**第七 如何衡量一个算法的执行效率？**

学习 C++ 标准库，特别是 STL，经常需要考量算法和成员函数的效能（也就是运行效率，又称复杂度），因此每个学习 STL 的读者都需要掌握一种衡量算法（或成员函数）复杂度的方法，目前最常用的方法称为大 O 表示法**（注意，不是数字 0，而是字母 O）**。  
 使用大 O 表示法衡量某个算法的复杂度，其实就是将该算法的运行时间用输入量为 n 的函数表示出来。这里的输入量 n 在 STL 中通常指的是算法操作的元素个数。  
 举个例子，当算法运行时间随元素个数成线性增长时（即如果元素个数呈倍数增长，运行时间也呈倍数增长），该算法的复杂度用 O(n) 来表示；反之，如果算法的运行时间和输入量 n 无关，则该算法的复杂度就用 O(1) 来表示。

**表 1 列出了常见的算法复杂度种类，以及使用大 O 表示法表示的复杂度。**

表 1 常见的算法复杂度表示

|  |  |  |
| --- | --- | --- |
| 算法复杂度种类 | 含义 | 大 O 表示法 |
| 常数阶 | 算法运行时间和所操作的元素个数无关 | O(1) |
| 对数阶 | 算法运行时间随所操作的元素个数呈对数增长 | O(log(n)) |
| 线性阶 | 算法运行时间随所操作的元素个数呈线性增长 | O(n) |
| 指数阶（m次方，m为数字） | 算法运行时间随所操作的元素个数呈 m 次方增长  O(nm) | 常见的有 O(n2)、O(n3) 等 |

值得注意的是，大 O 表示法并不关心算法运行所消耗的具体时间，换句话说，对于那些影响算法运行效率较小的因素，使用大 O 表示法表示时会直接将其忽略。例如，某个算法运行的复杂度为 O(n)，呈线性增长，但至于线性增长的具体程度（是 100n 还是 2n），在大 O 表示法看来，它们是一样的。也就是说，采用这种测量法则，任何两个线性算法都将被视为具有相同的复杂度。

采用大 O 表示法甚至会出现这种一种情况，即带有巨大常量的线性算法，很有可能会比小常量的指数算法更受欢迎，因为该方法无法显示出真实的运行时间。

所以请读者记住，大 O 表示法只是某种度量方法，它所显示的算法的最佳复杂度，并不一定就是真正的最佳（最快）算法。

表 2 复杂度随元素个数对照表

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 复杂度 | | 元素个数 | | | | | | | | | | |
| 种类 | 大 O 表示 | 1 | 2 | 5 | 10 | 50 | 100 | 1 000 | 10 000 |  |  |  | |  |  |
| 常量阶 | O(1) | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |  |  |  | |  |  |
| 对数阶 | O(log(n)) | 1 | 2 | 3 | 4 | 6 | 7 | 10 | 13 |  |  |  | |  |  |
| 线性阶 | O(n) | 1 | 2 | 5 | 10 | 50 | 100 | 1 000 | 10 000 |  |  |  | |  |  |
| 2次方 | O(n2) | 1 | 4 | 25 | 100 | 2500 | 10 000 | 1 000 000 | 100 000 000 |  |  |  | |  |  |

表 2 列出了常用的复杂度随元素个数增长的变化程序。可以看到，当算法处理的元素较少时，各复杂度的差别很小，此时算法效率的优劣往往受被大 O  表示法忽略部分的影响更大。而当处理元素个数越多，各复杂度的差别越大，此时复杂度被忽略的部分就变得无关紧要了。  
因此，在考量算法的复杂度时，输入量 n （操作元素的个数）的值必须足够大才有意义。

**第二章STL序列式容器**

**第一C++ STL容器是什么？**

在实际的开发过程中，合理组织数据的存取与选择处理数据的算法同等重要，存取数据的方式往往会直接影响到对它们进行增删改查操作的复杂程度和时间消耗。事实上，当程序中存在对时耗要求很高的部分时，[数据结构](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)的选择就显得尤为重要，有时甚至直接影响程序执行的成败。  
 值得一提的是，之前我们一直在不断地重复实现一些诸如链表、集合等等这些常见的数据结构，这些代码使用起来往往都十分类似，只是为了适应不同数据的变化，可能需要在一些细节上做不同的处理。  
 那么大家有没有想过，是不是可以重复利用那些已有的实现来完成当前的任务呢？当然是可行的，有些读者已经亲自编写并实现了动态数组类、链表类、集合类等程序，并精心维护着。其实，[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 中提供了专家级的几乎我们所需要的各种容器，功能更好，复用性更高。  
 简单的理解容器，它就是一些模板类的集合，但和普通模板类不同的是，容器中封装的是组织数据的方法（也就是数据结构）。STL 提供有 3 类标准容器，徐分别是序列容器、排序容器和哈希容器，其中后两类容器有时也统称为关联容器。它们各自的含义如表 1 所示。

表 1 STL 容器种类和功能

|  |  |
| --- | --- |
| 容器种类 | 功能 |
| 序列容器 | 主要包括 **vector 向量容器、list 列表容器以及 deque 双端队列容器。**之所以被称为序列容器，**是因为元素在容器中的位置同元素的值无关，即容器不是排序的。将元素插入容器时，指定在什么位置，元素就会位于什么位置**。 |
| 排序容器 | 包括 **set 集合容器、multiset多重集合容器、map映射容器以及 multimap 多重映射容器。**排序容器中的元素默认是由小到大排序好的，即便是插入元素，元素也会插入到适当位置。所以关联容器在查找时具有非常好的性能。 |
| 哈希容器 | [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 新加入 4 种关联式容器，**分别是 unordered\_set 哈希集合、unordered\_multiset 哈希多重集合、unordered\_map 哈希映射以及 unordered\_multimap 哈希多重映射**。和排序容器不同，哈希容器中的元素是未排序的，元素的位置由哈希函数确定。 |

**注意，由于哈希容器直到 C++ 11 才被正式纳入 C++ 标准程序库，而在此之前，“民间”流传着 hash\_set、hash\_multiset、hash\_map、hash\_multimap 版本，不过该版本只能在某些支持 C++ 11 的编译器下使用（如 VS），有些编译器（如 gcc/g++）是不支持的**。

另外，以上 3 类容器的存储方式完全不同，因此使用不同容器完成相同操作的效率也大不相同。所以在实际使用时，要善于根据想实现的功能，选择合适的容器。有关这些容器的具体用法，本章后续会逐个进行介绍。

**第二迭代器是什么，C++ STL迭代器（iterator）用法详解**

无论是序列容器还是关联容器，最常做的操作无疑是遍历容器中存储的元素，而实现此操作，多数情况会选用“迭代器（iterator）”来实现。那么，迭代器到底是什么呢？  
 我们知道，尽管不同容器的内部结构各异，但它们本质上都是用来存储大量数据的，换句话说，都是一串能存储多个数据的存储单元。因此，诸如数据的排序、查找、求和等需要对数据进行遍历的操作方法应该是类似的。  
 既然类似，完全可以利用泛型技术，将它们设计成适用所有容器的通用算法，从而将容器和算法分离开。但实现此目的需要有一个类似中介的装置，它除了要具有对容器进行遍历读写数据的能力之外，还要能对外隐藏容器的内部差异，从而以统一的界面向算法传送数据。  
 这是泛型思维发展的必然结果，于是迭代器就产生了。简单来讲，迭**代器和 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 的[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)非常类似，它可以是需要的任意类型，通**过迭代器可以指向容器中的某个元素，如果需要，还可以对该元素进行读/写操作。

**一迭代器类别**

[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 标准库为每一种标准容器定义了一种迭代器类型，这意味着，不同容器的迭代器也不同，其功能强弱也有所不同。

容器的迭代器的功能强弱，决定了该容器是否支持 STL 中的某种算法。

常用的迭代器按功能强弱分为输入迭代器、输出迭代器、前向迭代器、双向迭代器、随机访问迭代器 5 种。本节主要介绍后面的这 3 种迭代器。

输入迭代器和输出迭代器比较特殊，它们不是把数组或容器当做操作对象，而是把输入流/输出流作为操作对象。有关这 2 个迭代器，我们会在后续章节做详细介绍。

1. **前向迭代器（forward iterator）**

假设 p 是一个前向迭代器，则 p 支持 ++p，p++，\*p 操作，还可以被复制或赋值，可以用 == 和 != 运算符进行比较。此外，两个正向迭代器可以互相赋值。

**（2）双向迭代器（bidirectional iterator）**

双向迭代器具有正向迭代器的全部功能，除此之外，假设 p 是一个双向迭代器，则还可以进行 --p 或者 p-- 操作（即一次向后移动一个位置）。

**（3）随机访问迭代器（random access iterator）**

随机访问迭代器具有双向迭代器的全部功能。除此之外，假设 p 是一个随机访问迭代器，i 是一个整型变量或常量，则 p 还支持以下操作：

p+=i：使得 p 往后移动 i 个元素。

p-=i：使得 p 往前移动 i 个元素。

p+i：返回 p 后面第 i 个元素的迭代器。

p-i：返回 p 前面第 i 个元素的迭代器。

p[i]：返回 p 后面第 i 个元素的引用。  
此外，两个随机访问迭代器 p1、p2 还可以用 <、>、<=、>= 运算符进行比较。另外，表达式 p2-p1 也是有定义的，其返回值表示 p2 所指向元素和 p1 所指向元素的序号之差（也可以说是 p2 和 p1 之间的元素个数减一）。

表 1 所示，是 C++ 11 标准中不同容器指定使用的迭代器类型。

表 1 不同容器的迭代器

|  |  |
| --- | --- |
| 容器 | 对应的迭代器类型 |
| array | 随机访问迭代器 |
| vector | 随机访问迭代器 |
| deque | 随机访问迭代器 |
| list | 双向迭代器 |
| set / multiset | 双向迭代器 |
| map / multimap | 双向迭代器 |
| forward\_list | 前向迭代器 |
| unordered\_map / unordered\_multimap | 前向迭代器 |
| unordered\_set / unordered\_multiset | 前向迭代器 |
| stack | 不支持迭代器 |
| queue | 不支持迭代器 |

**注意，容器适配器 stack 和 queue 没有迭代器，它们包含有一些成员函数，可以用来对元素进行访问。**

**二迭代器的定义方式**

尽管不同容器对应着不同类别的迭代器，但这些迭代器有着较为统一的定义方式，具体分为 4 种，如表 1 所示。

表 2 迭代器的 4 种定义方式

|  |  |
| --- | --- |
| **迭代器定义方式** | **具体格式** |
| 正向迭代器 | 容器类名::iterator  迭代器名; |
| 常量正向迭代器 | 容器类名::const\_iterator  迭代器名; |
| 反向迭代器 | 容器类名::reverse\_iterator  迭代器名; |
| 常量反向迭代器 | 容器类名::const\_reverse\_iterator  迭代器名; |

**值得一提的是，表 2 中的反向迭代器全称为 "反向迭代器适配器"，后续章节会做详细讲解，这里读者只需要知道其用法即可。**

通过定义以上几种迭代器，就可以读取它指向的元素，\*迭代器名就表示迭代器指向的元素。其中，常量迭代器和非常量迭代器的分别在于，通过非常量迭代器还能修改其指向的元素。另外，反向迭代器和正向迭代器的区别在于：

**（1）**对正向迭代器进行 ++ 操作时，迭代器会指向容器中的后一个元素；

**（2）**而对反向迭代器进行 ++ 操作时，迭代器会指向容器中的前一个元素。

**注意，以上 4 种定义迭代器的方式，并不是每个容器都适用。有一部分容器同时支持以上 4 种方式，比如 array、deque、vector；而有些容器只支持其中部分的定义方式，例如 forward\_list 容器只支持定义正向迭代器，不支持定义反向迭代器。**

以上对迭代器做了很详细的介绍，下面就以 vector 容器为例，带领大家实际感受迭代器的用法和功能。通过前面的学习，vector 支持随机访问迭代器，因此遍历 vector 容器有以下几种做法。下面的程序中，每个循环演示了一种做法：

//遍历 vector 容器。

#include <iostream>

//需要引入 vector 头文件

#include <vector>

using namespace *std*;

int main()

{

*vector*<int> v{ 1,2,3,4,5,6,7,8,9,10 }; //v被初始化成有10个元素

*cout* << "第一种遍历方法：" << *endl*;

//size返回元素个数

for (int i = 0; i < v.*size*(); ++i)

*cout* << v[i] << " "; //像普通数组一样使用vector容器

//创建一个正向迭代器，当然，vector也支持其他 3 种定义迭代器的方式

*cout* << *endl* << "第二种遍历方法：" << *endl*;

*vector*<int>::*iterator* i;

//用 != 比较两个迭代器

for (i = v.*begin*(); i != v.*end*(); ++i)

*cout* << \*i << " ";

*cout* << *endl* << "第三种遍历方法：" << *endl*;

for (i = v.*begin*(); i < v.*end*(); ++i) //用 < 比较两个迭代器

*cout* << \*i << " ";

*cout* << *endl* << "第四种遍历方法：" << *endl*;

i = v.*begin*();

while (i < v.*end*()) { //间隔一个输出

*cout* << \*i << " ";

i += 2; // 随机访问迭代器支持 "+= 整数" 的操作

}

}

**运行结果为：**

**第一种遍历方法：  
1 2 3 4 5 6 7 8 9 10  
第二种遍历方法：  
1 2 3 4 5 6 7 8 9 10  
第三种遍历方法：  
1 2 3 4 5 6 7 8 9 10  
第四种遍历方法：  
1 3 5 7 9**

再举一个例子，我们知道，list 容器的迭代器是双向迭代器。假设 v 和 i 的定义如下：

//创建一个 v list容器

*list*<int> v;

//创建一个常量正向迭代器，同样，list也支持其他三种定义迭代器的方式。

*list*<int>::*const\_iterator* i;

则以下代码是合法的：

for (i = v.*begin*(); i != v.*end*(); ++i)

*cout* << \*i;

以下代码则不合法，因为双向迭代器不支持用“<”进行比较：

for (i = v.*begin*(); i < v.*end*(); ++i)

*cout* << \*i;

以下代码也不合法，因为双向迭代器不支持用下标随机访问元素：

for (int i = 0; i < v.*size*(); ++i)

*cout* << v[i];

其实在 C++ 中，数组也是容器。数组的迭代器就是指针，而且是随机访问迭代器。例如，对于数组 int a[10]，int \* 类型的指针就是其迭代器。则 a、a+1、a+2 都是 a 的迭代器。另外，以上有关 vector、list 容器的具体用法，后续章节会做详细讲解。

**第三C++序列式容器（STL序列式容器）是什么**

所谓序列容器，即以线性排列（类似普通数组的存储方式）来存储某一指定类型（例如 int、double 等）的数据，需要特殊说明的是，该类容器并不会自动对存储的元素按照值的大小进行排序。

需要注意的是，序列容器只是一类容器的统称，并不指具体的某个容器，序列容器大致包含以下几类容器：

**（1）array<T,N>（数组容器）**：表示可以存储 N 个 T 类型的元素，是 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 本身提供的一种容器。此类容器一旦建立，其长度就是固定不变的，这意味着不能增加或删除元素，只能改变某个元素的值；

**（2）vector<T>（向量容器）：**用来存放 T 类型的元素，是一个长度可变的序列容器，即在存储空间不足时，会自动申请更多的内存。使用此容器，在尾部增加或删除元素的效率最高（时间复杂度为 O(1) 常数阶），在其它位置插入或删除元素效率较差（时间复杂度为 O(n) 线性阶，其中 n 为容器中元素的个数）；

**（3）deque<T>（双端队列容器）：**和 vector 非常相似，区别在于使用该容器不仅尾部插入和删除元素高效，在头部插入或删除元素也同样高效，时间复杂度都是 O(1) 常数阶，但是在容器中某一位置处插入或删除元素，时间复杂度为 O(n) 线性阶；

**（4）list<T>（链表容器）：**是一个长度可变的、由 T 类型元素组成的序列，它以双向链表的形式组织元素，在这个序列的任何地方都可以高效地增加或删除元素（时间复杂度都为常数阶 O(1)），但访问容器中任意元素的速度要比前三种容器慢，这是因为 list<T> 必须从第一个元素或最后一个元素开始访问，需要沿着链表移动，直到到达想要的元素。

**（5）forward\_list<T>（正向链表容器）：**和 list 容器非常类似，只不过它以单链表的形式组织元素，它内部的元素只能从第一个元素开始访问，是一类比链表容器快、更节省内存的容器。

**注意，其实除此之外，stack<T> 和 queue<T> 本质上也属于序列容器，只不过它们都是在 deque 容器的基础上改头换面而成，通常更习惯称它们为容器适配器，有关它们的介绍，会放到后续章节中。**

图 1 说明了可供使用的序列容器以及它们之间的区别。

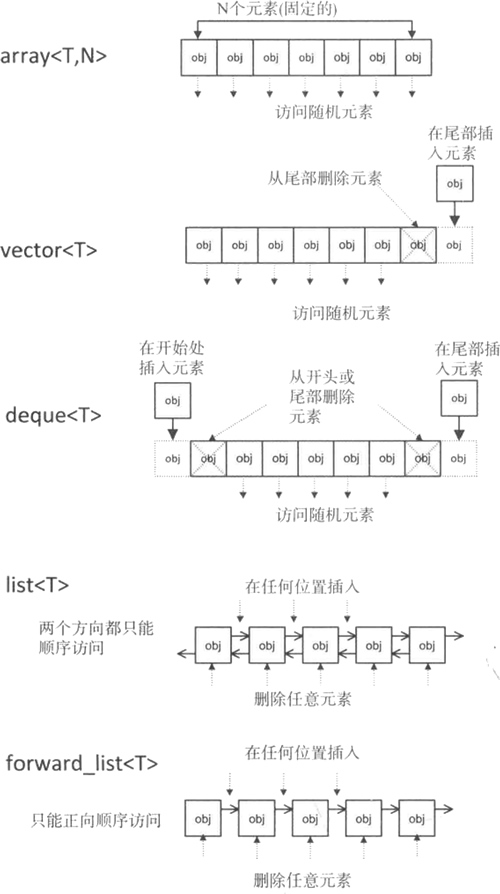


图 1 标准的序列容器

图 1 中每种类型容器的操作都可以高效执行，但进行除此之外的其他操作，效率会稍差一些。在本章的剩余部分，会详细介绍每一类序列容器的具体用法。

**一容器中常见的函数成员**

序列容器包含一些相同的成员函数，它们的功能也相同，本教程会在某个容器的上下文中详细介绍下面的每个函数，但对于每种类型的容器不会重复介绍它们的细节。

表 2 展示了 array、vector 和 deque 容器的函数成员，它们中至少有两个容器实现了同样的函数成员。

表 2 array、vector 和 deque 容器的函数成员

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **函数成员** | **函数功能** | **array<T,N>** | **vector<T>** | **deque<T>** |
| begin() | 返回指向容器中第一个元素的迭代器。 | 是 | 是 | 是 |
| end() | 返回指向容器最后一个元素所在位置后一个位置的迭代器，通常和 begin() 结合使用。 | 是 | 是 | 是 |
| rbegin() | 返回指向最后一个元素的迭代器。 | 是 | 是 | 是 |
| rend() | 返回指向第一个元素所在位置前一个位置的迭代器。 | 是 | 是 | 是 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 | 是 | 是 | 是 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 | 是 | 是 | 是 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 | 是 | 是 | 是 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 | 是 | 是 | 是 |
| assign() | 用新元素替换原有内容。 | - | 是 | 是 |
| operator=() | 复制同类型容器的元素，或者用初始化列表替换现有内容。 | 是 | 是 | 是 |
| size() | 返回实际元素个数。 | 是 | 是 | 是 |
| max\_size() | 返回元素个数的最大值。这通常是一个很大的值，一般是 232-1，所以我们很少会用到这个函数。 | 是 | 是 | 是 |
| capacity() | 返回当前容量。 | - | 是 | - |
| empty() | 判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。 | 是 | 是 | 是 |
| resize() | 改变实际元素的个数。 | - | 是 | 是 |
| shrink \_to\_fit() | 将内存减少到等于当前元素实际所使用的大小。 | - | 是 | 是 |
| front() | 返回第一个元素的引用。 | 是 | 是 | 是 |
| back() | 返回最后一个元素的引用。 | 是 | 是 | 是 |
| operator[]() | 使用索引访问元素。 | 是 | 是 | 是 |
| at() | 使用经过边界检査的索引访问元素。 | 是 | 是 | 是 |
| push\_back() | 在序列的尾部添加一个元素。 | - | 是 | 是 |
| insert() | 在指定的位置插入一个或多个元素。 | - | 是 | 是 |
| emplace() | 在指定的位置直接生成一个元素。 | - | 是 | 是 |
| emplace\_back() | 在序列尾部生成一个元素。 | - | 是 | 是 |
| pop\_back() | 移出序列尾部的元素。 | - | 是 | 是 |
| erase() | 移出一个元素或一段元素。 | - | 是 | 是 |
| clear() | 移出所有的元素，容器大小变为 0。 | - | 是 | 是 |
| swap() | 交换两个容器的所有元素。 | 是 | 是 | 是 |
| data() | 返回指向容器中第一个元素的[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)。 | 是 | 是 | - |

列表中 - 表明对应的容器并没有定义这个函数。

**list 和 forward\_list 容器彼此非常相似，forward\_list 中包含了 list 的大部分成员函数，而未包含那些需要反向遍历的函数。表 3 展示了 list 和 forward\_list 的函数成员。**

表 3 list 和 forward\_list 的函数成员

|  |  |  |  |
| --- | --- | --- | --- |
| **函数成员** | **函数功能** | **list<T>** | **forward\_list<T>** |
| begin() | 返回指向容器中第一个元素的迭代器。 | 是 | 是 |
| end() | 返回指向容器最后一个元素所在位置后一个位置的迭代器。 | 是 | 是 |
| rbegin() | 返回指向最后一个元素的迭代器。 | 是 | - |
| rend() | 返回指向第一个元素所在位置前一个位置的迭代器。 | 是 | - |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 | 是 | 是 |
| before\_begin() | 返回指向第一个元素前一个位置的迭代器。 | - | 是 |
| cbefore\_begin() | 和 before\_begin() 功能相同，只不过在其基础上，增加了 const 属性，即不能用该指针修改元素的值。 | - | 是 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 | 是 | 是 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 | 是 | - |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 | 是 | - |
| assign() | 用新元素替换原有内容。 | 是 | 是 |
| operator=() | 复制同类型容器的元素，或者用初始化列表替换现有内容。 | 是 | 是 |
| size() | 返回实际元素个数。 | 是 | - |
| max\_size() | 返回元素个数的最大值，这通常是一个很大的值，一般是 232-1，所以我们很少会用到这个函数。 | 是 | 是 |
| resize() | 改变实际元素的个数。 | 是 | 是 |
| empty() | 判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。 | 是 | 是 |
| front() | 返回容器中第一个元素的引用。 | 是 | 是 |
| back() | 返回容器中最后一个元素的引用。 | 是 | - |
| push\_back() | 在序列的尾部添加一个元素。 | 是 | - |
| push\_front() | 在序列的起始位置添加一个元素。 | 是 | 是 |
| emplace() | 在指定位置直接生成一个元素。 | 是 | - |
| emplace\_after() | 在指定位置的后面直接生成一个元素。 | - | 是 |
| emplace\_back() | 在序列尾部生成一个元素。 | 是 | - |
| cmplacc\_front() | 在序列的起始位生成一个元索。 | 是 | 是 |
| insert() | 在指定的位置插入一个或多个元素。 | 是 | - |
| insert\_after() | 在指定位置的后面插入一个或多个元素。 | - | 是 |
| pop\_back() | 移除序列尾部的元素。 | 是 | - |
| pop\_front() | 移除序列头部的元素。 | 是 | 是 |
| reverse() | 反转容器中某一段的元素。 | 是 | 是 |
| erase() | 移除指定位置的一个元素或一段元素。 | 是 | - |
| erase\_after() | 移除指定位置后面的一个元素或一段元素。 | - | 是 |
| remove() | 移除所有和参数匹配的元素。 | 是 | 是 |
| remove\_if() | 移除满足一元函数条件的所有元素。 | 是 | 是 |
| unique() | 移除所有连续重复的元素。 | 是 | 是 |
| clear() | 移除所有的元素，容器大小变为 0。 | 是 | 是 |
| swap() | 交换两个容器的所有元素。 | 是 | 是 |
| sort() | 对元素进行排序。 | 是 | 是 |
| merge() | 合并两个有序容器。 | 是 | 是 |
| splice() | 移动指定位置前面的所有元素到另一个同类型的 list 中。 | 是 | - |
| splice\_after() | 移动指定位置后面的所有元素到另一个同类型的 list 中。 | - | 是 |

**第四C++ array(STL array)容器用法详解**

array 容器是 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 标准中新增的序列容器，简单地理解，它就是在 C++ 普通数组的基础上，添加了一些成员函数和全局函数。在使用上，它比普通数组更安全（原因后续会讲），且效率并没有因此变差。

和其它容器不同，array 容器的大小是固定的，无法动态的扩展或收缩，这也就意味着，在使用该容器的过程无法借由增加或移除元素而改变其大小，它只允许访问或者替换存储的元素。

**[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 还提供有可动态扩展或收缩存储空间的 vector 容器，后续章节会对其做详细介绍。**

array 容器以类模板的形式定义在 <array> 头文件，并位于命名空间 std 中，如下所示：

namespace std{

template <typename T, size\_t N>

class array;

}

因此，在使用该容器之前，代码中需引入 <array> 头文件，并默认使用 std 命令空间，如下所示：

#include <array>

u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;

在 array<T,N> 类模板中，T 用于指明容器中的存储的具体数据类型，N 用于指明容器的大小，需要注意的是，这里的 N 必须是常量，不能用变量表示。

array 容器有多种初始化方式，如下代码展示了如何创建具有 4 个 int 类型元素的 array 容器：

std::array<double, 10> values;

由此，就创建好了一个名为 values 的 array 容器，其包含 10 个浮点型元素。但是，由于未显式指定这 10 个元素的值，因此使用这种方式创建的容器中，各个元素的值是不确定的（array 容器不会做默认初始化操作）。

通过如下创建 array 容器的方式，可以将所有的元素初始化为 0 或者和默认元素类型等效的值：

std::array<double, 10> values {};

使用该语句，容器中所有的元素都会被初始化为 0.0。

当然，在创建 array 容器的实例时，也可以像创建常规数组那样对元素进行初始化：

*std*::array<double, 10> values{ 0.5,1.0,1.5,,2.0 };

可以看到，这里只初始化了前 4 个元素，剩余的元素都会被初始化为 0.0。图 1 说明了这一点。

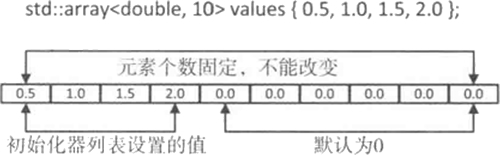


图 1 初始化 array 容器

除此之外，array 容器还提供有很多功能实用的成员函数，如表 2 所示。

表 2 array容器成员函数汇总

|  |  |
| --- | --- |
| **成员函数** | **函数功能** |
| begin() | 返回指向容器中第一个元素的随机访问迭代器。 |
| end() | 返回指向容器最后一个元素之后一个位置的随机访问迭代器，通常和 begin() 结合使用。 |
| rbegin() | 返回指向最后一个元素的随机访问迭代器。 |
| rend() | 返回指向第一个元素之前一个位置的随机访问迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上增加了 const 属性，不能用于修改元素。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| size() | 返回容器中当前元素的数量，其值始终等于初始化 array 类的第二个模板参数 N。 |
| max\_size() | 返回容器可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 N。 |
| empty() | 判断容器是否为空，和通过 size()==0 的判断条件功能相同，但其效率可能更快。 |
| at(n) | 返回容器中 n 位置处元素的引用，该函数自动检查 n 是否在有效的范围内，如果不是则抛出 out\_of\_range 异常。 |
| front() | 返回容器中第一个元素的直接引用，该函数不适用于空的 array 容器。 |
| back() | 返回容器中最后一个元素的直接应用，该函数同样不适用于空的 array 容器。 |
| data() | 返回一个指向容器首个元素的[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)。利用该指针，可实现复制容器中所有元素等类似功能。 |
| fill(val) | 将 val 这个值赋值给容器中的每个元素。 |
| array1.swap(array2) | 交换 array1 和 array2 容器中的所有元素，但前提是它们具有相同的长度和类型。 |

除此之外，C++ 11 标准库还新增加了 begin() 和 end() 这 2 个函数，和 array 容器包含的 begin() 和 end() 成员函数不同的是，标准库提供的这 2 个函数的操作对象，既可以是容器，还可以是普通数组。当操作对象是容器时，它和容器包含的 begin() 和 end() 成员函数的功能完全相同；如果操作对象是普通数组，则 begin() 函数返回的是指向数组第一个元素的指针，同样 end() 返回指向数组中最后一个元素之后一个位置的指针（注意不是最后一个元素）。  
 另外，在 <array> 头文件中还重载了 get() 全局函数，该重载函数的功能是访问容器中指定的元素，并返回该元素的引用。

正是由于 array 容器中包含了 at() 这样的成员函数，使得操作元素时比普通数组更安全。

例如代码演示了表 2 中一部分成员函数的用法和功能：

#include <iostream>

//需要引入 array 头文件

#include <array>

using namespace *std*;

int main()

{

*std*::array<int, 4> values{};

//初始化 values 容器为 {0,1,2,3}

for (int i = 0; i < values.*size*(); i++) {

values.*at*(i) = i;

}

//使用 get() 重载函数输出指定位置元素

*cout* << *get*<3>(values) << *endl*;

//如果容器不为空，则输出容器中所有的元素

if (!values.*empty*()) {

for (auto val = values.*begin*(); val < values.*end*(); val++) {

*cout* << \*val << " ";

}

}

}

注意，代码中的 auto 关键字，可以使编译器自动判定变量的类型。运行这段代码，输出结果为：

3

0 1 2 3

表 2 中其他成员函数的用法，这里不再给出具体实例，有兴趣的读者，可自行根据各个函数的功能描述编写实例代码进行测试。

**第五C++ STL array随机访问迭代器**

在《[C++ STL迭代器（iterator）](http://c.biancheng.net/view/6675.html" \t "http://c.biancheng.net/view/_blank)》一节中，已经对迭代器做了详细的介绍，[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 为 array 容器配备了随机访问迭代器，该类迭代器是功能最强大的迭代器。本节将详细介绍 array 容器的迭代器的用法。  
 在 array 容器的模板类中，和随机访问迭代器相关的成员函数如表 1 所示。

表 1 array 支持迭代器的成员函数

|  |  |
| --- | --- |
| **成员函数** | **功能** |
| begin() | 返回指向容器中第一个元素的正向迭代器；如果是 const 类型容器，在该函数返回的是常量正向迭代器。 |
| end() | 返回指向容器最后一个元素之后一个位置的正向迭代器；如果是 const 类型容器，在该函数返回的是常量正向迭代器。此函数通常和 begin() 搭配使用。 |
| rbegin() | 返回指向最后一个元素的反向迭代器；如果是 const 类型容器，在该函数返回的是常量反向迭代器。 |
| rend() | 返回指向第一个元素之前一个位置的反向迭代器。如果是 const 类型容器，在该函数返回的是常量反向迭代器。此函数通常和 rbegin() 搭配使用。 |
| cbegin() | 和 begin() 功能类似，只不过其返回的迭代器类型为常量正向迭代器，不能用于修改元素。 |
| cend() | 和 end() 功能相同，只不过其返回的迭代器类型为常量正向迭代器，不能用于修改元素。 |
| crbegin() | 和 rbegin() 功能相同，只不过其返回的迭代器类型为常量反向迭代器，不能用于修改元素。 |
| crend() | 和 rend() 功能相同，只不过其返回的迭代器类型为常量反向迭代器，不能用于修改元素。 |

除此之外，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 标准新增的 begin() 和 end() 函数，当操作对象为 array 容器时，也和迭代器有关，其功能分别和表 1 中的 begin()、end() 成员函数相同，具有用法本节后续会做详细介绍。

这些成员函数的具体功能如图 2 所示。

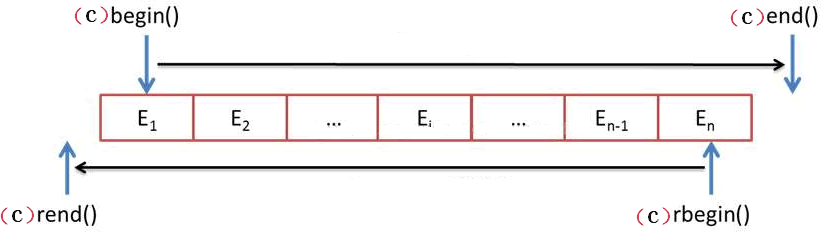


图 2 迭代器的具体功能示意图

可以看到，根据它们的功能并结合实际场景的需要，这些成员函数通常是成对使用的，即 begin()/end()、rbegin()/rend()、cbegin()/cend()、crbegin()/crend() 各自成对搭配使用。不仅如此，这 4 对中 begin()/end() 和 cbegin()/cend()、rbegin()/rend() 和 crbegin()/crend() 的功能大致是相同的（如图 2 所示），唯一的区别就在于其返回的迭代器能否用来修改元素值。

值得一提的是，以上函数在实际使用时，其返回值类型都可以使用 auto 关键字代替，编译器可以自行判断出该迭代器的类型。

**一begin()/end() 和 cbegin()/cend()**

array 容器模板类中的 begin() 和 end() 成员函数返回的都是正向迭代器，它们分别指向「首元素」和「尾元素+1」 的位置。在实际使用时，我们可以利用它们实现初始化容器或者遍历容器中元素的操作。  
例如，可以在循环中显式地使用迭代器来初始化 values 容器的值：

#include <iostream>

//需要引入 array 头文件

#include <array>

using namespace *std*;

int main()

{

array<int, 5>values;

int h = 1;

auto first = values.*begin*();

auto last = values.*end*();

//初始化 values 容器为{1,2,3,4,5}

while (first != last)

{

\*first = h;

++first;

h++;

}

first = values.*begin*();

while (first != last)

{

*cout* << \*first << " ";

++first;

}

return 0;

}

输出结果为：

1 2 3 4 5

可以看出，迭代器对象是由 array 对象的成员函数 begin() 和 end() 返回的。我们可以像使用普通[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)那样上使用迭代器对象。比如代码中，在保存了元素值后，使用前缀 ++ 运算符对 first 进行自增，当 first 等于 end 时，所有的元素都被设完值，循环结束。  
 与此同时，还可以使用全局的 begin() 和 end() 函数来从容器中获取迭代器，因为当操作对象为 array 容器时，它们和 begin()/end() 成员函数是通用的。所以上面代码中，first 和 last 还可以像下面这样定义：

auto first = *std*::*begin*(values);

auto last = *std*::*end*(values);

这样，容器中的一段元素可以由迭代器指定，这让我们有了对它们使用算法的可能。

**需要注意的是，STL 标准库，不是只有 array 容器，当迭代器指向容器中的一个特定元素时，它们不会保留任何关于容器本身的信息，所以我们无法从迭代器中判断，它是指向 array 容器还是指向 vector 容器（该容器后续会讲）。**

除此之外，array 模板类还提供了 cbegin() 和 cend() 成员函数，它们和 begin()/end() 唯一不同的是，前者返回的是 const 类型的正向迭代器，这就意味着，有 cbegin() 和 cend() 成员函数返回的迭代器，可以用来遍历容器内的元素，也可以访问元素，但是不能对所存储的元素进行修改。

举个例子：

#include <iostream>

//需要引入 array 头文件

#include <array>

using namespace *std*;

int main()

{

array<int, 5>values{ 1,2,3,4,5 };

int h = 1;

auto first = values.*cbegin*();

auto last = values.*cend*();

//由于 \*first 为 const 类型，不能用来修改元素

//\*first = 10;

//遍历容器并输出容器中所有元素

while (first != last)

{

//可以使用 const 类型迭代器访问元素

*cout* << \*first << " ";

++first;

}

return 0;

}

此程序的第 14 行代码中，我们尝试使用 first 迭代器修改 values 容器中的值，如果取消注释并运行此程序，编译器会提示你“不能给常量赋值”，即 \*first 是 const 类型常量，所以这么做是不对的。但 17~22 行代码遍历并访问容器的行为，是允许的。

### **二rbegin()/rend() 和 crbegin()/crend()**

array 模板类中还提供了 rbegin()/rend() 和 crbegin()/crend() 成员函数，它们每对都可以分别得到指向最一个元素和第一个元素前一个位置的随机访问迭代器，又称它们为反向迭代器（如图 2 所示）。

需要注意的是，在使用反向迭代器进行 ++ 或 -- 运算时，++ 指的是迭代器向左移动一位，-- 指的是迭代器向右移动一位，即这两个运算符的功能也“互换”了。

反向迭代器用于以逆序的方式处理元素。例如：

#include <iostream>

//需要引入 array 头文件

#include <array>

using namespace *std*;

int main()

{

array<int, 5>values;

int h = 1;

auto first = values.*rbegin*();

auto last = values.*rend*();

//初始化 values 容器为 {5,4,3,2,1}

while (first != last)

{

\*first = h;

++first;

h++;

}

//重新遍历容器，并输入各个元素

first = values.*rbegin*();

while (first != last)

{

*cout* << \*first << " ";

++first;

}

return 0;

}

**运行结果为：**

**1 2 3 4 5**

可以看到，从最后一个元素开始循环，不仅完成了容器的初始化，还遍历输出了容器中的所有元素。结束迭代器指向第一个元素之前的位置，所以当 first 指向第一个元素并 +1 后，循环就结朿了。

在反向迭代器上使用 ++ 递增运算符，会让迭代器用一种和普通正向迭代器移动方向相反的方式移动。

当然，在上面程序中，我们也可以使用 [for 循环](http://c.biancheng.net/view/172.html" \t "http://c.biancheng.net/view/_blank)：

for (auto first = values.*rbegin*(); first != values.*rend*(); ++first) {

*cout* << \*first << " ";

crbegin()/crend() 组合和 rbegin()/crend() 组合的功能唯一的区别在于，前者返回的迭代器为 const 类型，即不能用来修改容器中的元素，除此之外在使用上和后者完全相同。

有关 crbegin()/crend() 成员函数，这里不再给出具体实例，有兴趣的读者，可自行编写代码进行测试。

**第六C++ STL array容器访问元素的几种方式**

当 array 容器创建完成之后，最常做的操作就是获取其中的元素，甚至有时还会通过循环结构获取多个元素。本节就对获取容器中元素的方法做个汇总。

**一访问array容器中单个元素**

首先，可以通过容器名[]的方式直接访问和使用容器中的元素，这和 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 标准数组访问元素的方式相同，例如：

values[4] = values[3] + 2.O \* values[1];

此行代码中，第 5 个元素的值被赋值为右边表达式的值。需要注意的是，使用如上这样方式，由于没有做任何边界检查，所以即便使用越界的索引值去访问或存储元素，也不会被检测到。

为了能够有效地避免越界访问的情况，可以使用 array 容器提供的 at() 成员函数，例如 :

values.*at*(4) = values.*at*(3) + 2.O \* values.*at*(1);

这行代码和前一行语句实现的功能相同，其次当传给 at() 的索引是一个越界值时，程序会抛出 std::out\_of\_range 异常。因此当需要访问容器中某个指定元素时，建议大家使用 at()，除非确定索引没有越界。

**读者可能有这样一个疑问，即为什么 array 容器在重载 [] 运算符时，没有实现边界检查的功能呢？答案很简单，因为性能。如果每次访问元素，都去检查索引值，无疑会产生很多开销。当不存在越界访问的可能时，就能避免这种开销。**

除此之外，array 容器还提供了 get<n> 模板函数，它是一个辅助函数，能够获取到容器的第 n 个元素。需要注意的是，该模板函数中，参数的实参必须是一个在编译时可以确定的常量表达式，所以它不能是一个循环变量。也就是说，它只能访问模板参数指定的元素，编译器在编译时会对它进行检查。  
下面代码展示了如何使用 get<n> 模板函数：

#include <iostream>

#include <array>

#include <string>

using namespace *std*;

int main()

{

array<*string*, 5> words{ "one","two","three","four","five" };

*cout* << *get*<3>(words) << *endl*; // Output words[3]

//cout << get<6>(words) << std::endl; //越界，会发生编译错误

return 0;

}

运行结果为：

four

另外，array 容器提供了 data() 成员函数，通过调用该函数可以得到指向容器首个元素的[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)。通过该指针，我们可以获得容器中的各个元素，例如：

#include <iostream>

#include <array>

using namespace *std*;

int main()

{

array<int, 5> words{ 1,2,3,4,5 };

*cout* << \*(words.*data*() + 1);

return 0;

}

运行结果为：

2

**二访问array容器中多个元素**

我们知道，array 容器提供的 size() 函数能够返回容器中元素的个数（函数返回值为 size\_t 类型），所以能够像下面这样去逐个提取容器中的元素，并计算它们的和：

double total = 0;

for (*size\_t* i = 0; i < values.*size*(); ++i)

{

total += values[i];

}

size() 函数的存在，为 array 容器提供了标准数组所没有的优势，即能够知道它包含多少元素。  
 并且，接受数组容器作为参数的函数，只需要通过调用容器的成员函数 size()，就能得到元素的个数。除此之外，通过调用 array 容器的 empty() 成员函数，即可知道容器中有没有元素（如果容器中没有元素，此函数返回 true），如下所示：

if (values.*empty*())

*std*::*cout* << "The container has no elements.\n";

else

*std*::*cout* << "The container has " << values.*size*() << "elements.\n";

然而，很少会创建空的 array 容器，因为当生成一个 array 容器时，它的元素个数就固定了，而且无法改变，所以生成空 array 容器的唯一方法是将模板的第二个参数指定为 0，但这种情况基本不可能发生。

**array 容器之所以提供 empty() 成员函数的原因，对于其他元素可变或者元素可删除的容器（例如 vector、deque 等）来说，它们使用 empty() 时的机制是一样的，因此为它们提供了一个一致性的操作。**

除了借助 size() 外，对于任何可以使用迭代器的容器，都可以使用基于范围的循环，因此能够更加简便地计算容器中所有元素的和，比如：

double total = 0;

for (auto&& value : values)

total += value;

下面是一个示例，展示了本节关于如何获取 array 容器中元素所讲到的知识：

#include <iostream>

#include <iomanip>

#include <array>

using namespace *std*;

int main()

{

array<int, 5> values1;

array<int, 5> values2;

//初始化 values1 为 {0,1,2,3,4}

for (*size\_t* i = 0; i < values1.*size*(); ++i)

{

values1.*at*(i) = i;

}

*cout* << "values1[0] is : " << values1[0] << *endl*;

*cout* << "values1[1] is : " << values1.*at*(1) << *endl*;

*cout* << "values1[2] is : " << *get*<2>(values1) << *endl*;

//初始化 values2 为{10，11，12，13，14}

int initvalue = 10;

for (auto& value : values2)

{

value = initvalue;

initvalue++;

}

*cout* << "Values1 is : ";

for (auto i = values1.*begin*(); i < values1.*end*(); i++) {

*cout* << \*i << " ";

}

*cout* << *endl* << "Values2 is : ";

for (auto i = values2.*begin*(); i < values2.*end*(); i++) {

*cout* << \*i << " ";

}

return 0;

}

**运行结果为：**

**values1[0] is : 0**

**values1[1] is : 1**

**values1[2] is : 2**

**Values1 is : 0 1 2 3 4**

**Values2 is : 10 11 12 13 14**

**第七C++ array容器：普通数组的“升级版”**

和 C++ 普通数组存储数据的方式一样，C++ 标准库保证使用 array 容器存储的所有元素一定会位于连续且相邻的内存中，通过如下代码也可以验证这一点：

#include <iostream>

#include <array>

using namespace *std*;

int main()

{

array<int, 5>a{ 1,2,3 };

*cout* << &a[2] << " " << &a[0] + 2 << *endl*;

return 0;

}

输出结果为：

004FFD58 004FFD58

可以看到，a 容器中 &a[2] 和 &a[0] + 2 是相等的。因此在实际编程过程中，我们完全有理由去尝试，在原本使用普通数组的位置，改由 array 容器去实现。

**用 array 容器替换普通数组的好处是，array 模板类中已经封装好了大量实用的方法，在提高开发效率的同时，代码的运行效率也会大幅提高。**

举个例子，我们完全可以使用 array 容器去存储 char\* 或 const char\* 类型的字符串：

#include <iostream>

#include <array>

using namespace *std*;

int main()

{

array<char, 50>a{ 1,2,3 };

*strcpy*(&a[0], "http://c.biancheng.net/stl");

*printf*("%s", &a[0]);

return 0;

}

输出结果为：

<http://c.biancheng.net/stl>

注意，array 容器的大小必须保证能够容纳复制进来的数据，而且如果是存储字符串的话，还要保证在存储整个字符串的同时，在其最后放置一个\0作为字符串的结束符。此程序中，strcpy() 在拷贝字符串的同时，会自动在最后添加\0。

其实，代码中的 &a[0] 还可以用 array 模板类提供的 data() 成员函数来替换：

#include <iostream>

#include <array>

using namespace *std*;

int main()

{

array<char, 50>a{ 1,2,3 };

*strcpy*(a.*data*(), "http://c.biancheng.net/stl");

*printf*("%s", a.*data*());

return 0;

}

此程序和上面程序的输出结果完全相同。

**注意，容器的迭代器和指针是不能混用的，即上面代码中不能用 a.begin() 来代替 &a[0] 或者 a.data[]，这可能会引发错误。**

文章前面提到，使用 array 容器代替普通数组，最直接的好处就是 array 模板类中已经为我们写好了很多实用的方法，可以大大提高我们编码效率。例如，array 容器提供的 at() 成员函数，可以有效防止越界操纵数组的情况；fill() 函数可以实现数组的快速初始化；swap() 函数可以轻松实现两个相同数组（类型相同，大小相同）中元素的互换。

#include <iostream>

#include <array>

using namespace *std*;

int main()

{

array<char, 50>addr1{ "http://c.biancheng.net" };

array<char, 50>addr2{ "http://c.biancheng.net/stl" };

addr1.*swap*(addr2);

*printf*("addr1 is：%s\n", addr1.*data*());

*printf*("addr2 is：%s\n", addr2.*data*());

return 0;

}

运行结果为：

addr1 is：http ://c.biancheng.net/stl

addr2 is：http ://c.biancheng.net

另外，当两个 array 容器满足大小相同并且保存元素的类型相同时，两个 array 容器可以直接直接做赋值操作，即将一个容器中的元素赋值给另一个容器。比如：

#include <iostream>

#include <array>

using namespace *std*;

int main()

{

array<char, 50>addr1{ "http://c.biancheng.net" };

array<char, 50>addr2{ "http://c.biancheng.net/stl" };

addr1 = addr2;

*printf*("%s", addr1.*data*());

return 0;

}

运行结果为：

http://c.biancheng.net/stl

不仅如此，在满足以上 2 个条件的基础上，如果其保存的元素也支持比较运算符，就可以用任何比较运算符直接比较两个 array 容器。示例如下：

#include <iostream>

#include <array>

using namespace *std*;

int main()

{

array<char, 50>addr1{ "http://c.biancheng.net" };

array<char, 50>addr2{ "http://c.biancheng.net/stl" };

if (addr1 == addr2) {

*std*::*cout* << "addr1 == addr2" << *std*::*endl*;

}

if (addr1 < addr2) {

*std*::*cout* << "addr1 < addr2" << *std*::*endl*;

}

if (addr1 > addr2) {

*std*::*cout* << "addr1 > addr2" << *std*::*endl*;

}

return 0;

}

运行结果为：

addr1 < addr2

两个容器比较大小的原理，和两个字符串比较大小是一样的，即从头开始，逐个取两容器中的元素进行大小比较（根据 ASCII 码表），直到遇到两个不相同的元素，那个元素的值大，则该容器就大。  
 **总之，读者可以这样认为，array 容器就是普通数组的“升级版”，使用普通数组能实现的，使用 array 容器都可以实现，而且无论是代码功能的实现效率，还是程序执行效率，都比普通数组更高。**

**第八C++ STL vector容器详解**

vector 容器是 [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 中最常用的容器之一，它和 array 容器非常类似，都可以看做是对 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 普通数组的“升级版”。不同之处在于，array 实现的是静态数组（容量固定的数组），而 vector 实现的是一个动态数组，即可以进行元素的插入和删除，在此过程中，vector 会动态调整所占用的内存空间，整个过程无需人工干预。  
 vector 常被称为向量容器，因为该容器擅长在尾部插入或删除元素，在常量时间内就可以完成，时间复杂度为O(1)；而对于在容器头部或者中部插入或删除元素，则花费时间要长一些（移动元素需要耗费时间），时间复杂度为线性阶O(n)。

vector 容器以类模板 vector<T>（ T 表示存储元素的类型）的形式定义在 <vector> 头文件中，并位于 std 命名空间中。因此，在创建该容器之前，代码中需包含如下内容：

#include <vector>

using namespace *std*;

**一创建vector容器的几种方式**

创建 vector 容器的方式有很多，大致可分为以下几种。

（1）如下代码展示了如何创建存储 double 类型元素的一个 vector 容器：

*std*::*vector*<double> values;

注意，这是一个空的 vector 容器，因为容器中没有元素，所以没有为其分配空间。当添加第一个元素（比如使用 push\_back() 函数）时，vector 会自动分配内存。  
 在创建好空容器的基础上，还可以像下面这样通过调用 reserve() 成员函数来增加容器的容量：

values.reserve(20);

这样就设置了容器的内存分配，即至少可以容纳 20 个元素。注意，如果 vector 的容量在执行此语句之前，已经大于或等于 20 个元素，那么这条语句什么也不做；另外，调用 reserve() 不会影响已存储的元素，也不会生成任何元素，即 values 容器内此时仍然没有任何元素。

**还需注意的是，如果调用 reserve() 来增加容器容量，之前创建好的任何迭代器（例如开始迭代器和结束迭代器）都可能会失效，这是因为，为了增加容器的容量，vector<T> 容器的元素可能已经被复制或移到了新的内存地址。所以后续再使用这些迭代器时，最好重新生成一下。**

1. 除了创建空 vector 容器外，还可以在创建的同时指定初始值以及元素个数，比如：

*std*::*vector*<int> primes{ 2, 3, 5, 7, 11, 13, 17, 19 };

这样就创建了一个含有 8 个素数的 vector 容器。

1. 在创建 vector 容器时，也可以指定元素个数：

*std*::*vector*<double> values(20);

如此，values 容器开始时就有 20 个元素，它们的默认初始值都为 0。

**注意，圆括号 () 和大括号 {} 是有区别的，前者（例如 (20) ）表示元素的个数，而后者（例如 {20} ） 则表示 vector 容器中只有一个元素 20。**

如果不想用 0 作为默认值，也可以指定一个其它值，例如：

*std*::*vector*<double> values(20, 1.0);

第二个参数指定了所有元素的初始值，因此这 20 个元素的值都是 1.0。  
值得一提的是，圆括号 () 中的 2 个参数，既可以是常量，也可以用变量来表示，例如：

int num = 20;

double value = 1.0;

*std*::*vector*<double> values(num, value);

（4） 通过存储元素类型相同的其它 vector 容器，也可以创建新的 vector 容器，例如：

*std*::*vector*<char>value1(5, 'c');

*std*::*vector*<char>value2(value1);

由此，value2 容器中也具有 5 个字符 'c'。在此基础上，如果不想复制其它容器中所有的元素，可以用一对[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)或者迭代器来指定初始值的范围，例如：

int array[] = { 1,2,3 };

*std*::*vector*<int>values(array, array + 2);//values 将保存{1,2}

*std*::*vector*<int>value1{ 1,2,3,4,5 };

*std*::*vector*<int>value2(*std*::*begin*(value1), *std*::*begin*(value1) + 3);//value2保存{1,2,3}

由此，value2 容器中就包含了 {1,2,3} 这 3 个元素。

**二vector容器包含的成员函数**

相比 array 容器，vector 提供了更多了成员函数供我们使用，它们各自的功能如表 1 所示。

表 1 vector 容器的成员函数

|  |  |
| --- | --- |
| **函数成员** | **函数功能** |
| begin() | 返回指向容器中第一个元素的迭代器。 |
| end() | 返回指向容器最后一个元素所在位置后一个位置的迭代器，通常和 begin() 结合使用。 |
| rbegin() | 返回指向最后一个元素的迭代器。 |
| rend() | 返回指向第一个元素所在位置前一个位置的迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| size() | 返回实际元素个数。 |
| max\_size() | 返回元素个数的最大值。这通常是一个很大的值，一般是 232-1，所以我们很少会用到这个函数。 |
| resize() | 改变实际元素的个数。 |
| capacity() | 返回当前容量。 |
| empty() | 判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。 |
| reserve() | 增加容器的容量。 |
| shrink \_to\_fit() | 将内存减少到等于当前元素实际所使用的大小。 |
| operator[ ] | 重载了 [ ] 运算符，可以向访问数组中元素那样，通过下标即可访问甚至修改 vector 容器中的元素。 |
| at() | 使用经过边界检查的索引访问元素。 |
| front() | 返回第一个元素的引用。 |
| back() | 返回最后一个元素的引用。 |
| data() | 返回指向容器中第一个元素的指针。 |
| assign() | 用新元素替换原有内容。 |
| push\_back() | 在序列的尾部添加一个元素。 |
| pop\_back() | 移出序列尾部的元素。 |
| insert() | 在指定的位置插入一个或多个元素。 |
| erase() | 移出一个元素或一段元素。 |
| clear() | 移出所有的元素，容器大小变为 0。 |
| swap() | 交换两个容器的所有元素。 |
| emplace() | 在指定的位置直接生成一个元素。 |
| emplace\_back() | 在序列尾部生成一个元素。 |

除此之外，C++ 11 标准库还新增加了 begin() 和 end() 这 2 个函数，和 vector 容器包含的 begin() 和 end() 成员函数不同，标准库提供的这 2 个函数的操作对象，既可以是容器，还可以是普通数组。当操作对象是容器时，它和容器包含的 begin() 和 end() 成员函数的功能完全相同；如果操作对象是普通数组，则 begin() 函数返回的是指向数组第一个元素的指针，同样 end() 返回指向数组中最后一个元素之后一个位置的指针（注意不是最后一个元素）。  
 vector 容器还有一个 std::swap(x , y) 非成员函数（其中 x 和 y 是存储相同类型元素的  vector 容器），它和 swap() 成员函数的功能完全相同，仅使用语法上有差异。  
如下代码演示了表 1 中部分成员函数的用法：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

//初始化一个空vector容量

*vector*<char>value;

//向value容器中的尾部依次添加 S、T、L 字符

value.*push\_back*('S');

value.*push\_back*('T');

value.*push\_back*('L');

//调用 size() 成员函数容器中的元素个数

*printf*("元素个数为：%d\n", value.*size*());

//使用迭代器遍历容器

for (auto i = value.*begin*(); i < value.*end*(); i++) {

*cout* << \*i << " ";

}

*cout* << *endl*;

//向容器开头插入字符

value.*insert*(value.*begin*(), 'C');

*cout* << "首个元素为：" << value.*at*(0) << *endl*;

return 0;

}

**输出结果为：**

**元素个数为：3  
S T L  
首个元素为：C**

**表 1 中这些成员函数的具体用法，后续学习用到时会具体讲解，感兴趣的读者，也可以通过查阅 [STL手册](http://www.cplusplus.com/reference/stl/" \t "http://c.biancheng.net/view/_blank)做详细了解。**

**第九C++ STL vector容器迭代器用法详解**

在《[STL array随机访问迭代器](http://c.biancheng.net/view/6724.html" \t "http://c.biancheng.net/view/_blank)》一节中，详细介绍了 array 容器迭代器，vector 容器迭代器和前者有很多相同之处。比如，vector 容器的迭代器也是随机访问迭代器，并且 vector 模板类提供的操作迭代器的成员函数也和 array 容器一样（如表 1 所示）。

表 1 vector 支持迭代器的成员函数

|  |  |
| --- | --- |
| **成员函数** | **功能** |
| begin() | 返回指向容器中第一个元素的正向迭代器；如果是 const 类型容器，在该函数返回的是常量正向迭代器。 |
| end() | 返回指向容器最后一个元素之后一个位置的正向迭代器；如果是 const 类型容器，在该函数返回的是常量正向迭代器。此函数通常和 begin() 搭配使用。 |
| rbegin() | 返回指向最后一个元素的反向迭代器；如果是 const 类型容器，在该函数返回的是常量反向迭代器。 |
| rend() | 返回指向第一个元素之前一个位置的反向迭代器。如果是 const 类型容器，在该函数返回的是常量反向迭代器。此函数通常和 rbegin() 搭配使用。 |
| cbegin() | 和 begin() 功能类似，只不过其返回的迭代器类型为常量正向迭代器，不能用于修改元素。 |
| cend() | 和 end() 功能相同，只不过其返回的迭代器类型为常量正向迭代器，不能用于修改元素。 |
| crbegin() | 和 rbegin() 功能相同，只不过其返回的迭代器类型为常量反向迭代器，不能用于修改元素。 |
| crend() | 和 rend() 功能相同，只不过其返回的迭代器类型为常量反向迭代器，不能用于修改元素。 |

**除此之外，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 新添加的 begin() 和 end() 全局函数也同样适用于 vector 容器。即当操作对象为 vector 容器时，其功能分别和表 1 中的 begin()、end() 成员函数相同，具体用法本节后续会做详细介绍。**

表 1 中这些成员函数的具体功能如图 2 所示。

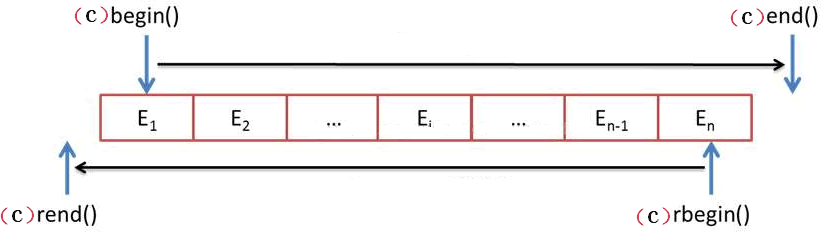


图 2 迭代器的具体功能示意图

从图 2 可以看出，这些成员函数通常是成对使用的，即 begin()/end()、rbegin()/rend()、cbegin()/cend()、crbegin()/crend() 各自成对搭配使用。其中，begin()/end() 和 cbegin/cend() 的功能是类似的，同样 rbegin()/rend() 和 crbegin()/crend() 的功能是类似的。

**值得一提的是，以上函数在实际使用时，其返回值类型都可以使用 auto 关键字代替，编译器可以自行判断出该迭代器的类型。**

**一vector容器迭代器的基本用法**

vector 容器迭代器最常用的功能就是遍历访问容器中存储的元素。  
首先来看 begin() 和 end() 成员函数，它们分别用于指向「首元素」和「尾元素+1」 的位置，下面程序演示了如何使用 begin() 和 end() 遍历 vector 容器并输出其中的元素：

#include <iostream>

//需要引入 vector 头文件

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>values{ 1,2,3,4,5 };

auto first = values.*begin*();

auto end = values.*end*();

while (first != end)

{

*cout* << \*first << " ";

++first;

}

return 0;

}

**输出结果为：**

**1 2 3 4 5**

可以看到，迭代器对象是由 vector 对象的成员函数 begin() 和 end() 返回的。我们可以像使用普通[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)那样上使用它们。比如代码中，在保存了元素值后，使用前缀++运算符对 first 进行自增，当 first 等于 end 时，所有的元素都被设完值，循环结束。  
 与此同时，还可以使用全局的 begin() 和 end() 函数来从容器中获取迭代器，比如将上面代码中第 8、9 行代码用如下代码替换：

auto first = std::begin(values);

auto end = std::end (values);

cbegin()/cend() 成员函数和 begin()/end() 唯一不同的是，前者返回的是 const 类型的正向迭代器，这就意味着，由 cbegin() 和 cend() 成员函数返回的迭代器，可以用来遍历容器内的元素，也可以访问元素，但是不能对所存储的元素进行修改。

#include <iostream>

//需要引入 vector 头文件

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>values{ 1,2,3,4,5 };

auto first = values.*cbegin*();

auto end = values.*cend*();

while (first != end)

{

//\*first = 10;不能修改元素

*cout* << \*first << " ";

++first;

}

return 0;

}

程序第 12 行，由于 first 是 const 类型的迭代器，因此不能用于修改容器中元素的值。  
vector 模板类中还提供了 rbegin() 和 rend() 成员函数，分别表示指向最后一个元素和第一个元素前一个位置的随机访问迭代器，又称它们为反向迭代器（如图 2 所示）。

**需要注意的是，在使用反向迭代器进行 ++ 或 -- 运算时，++ 指的是迭代器向左移动一位，-- 指的是迭代器向右移动一位，即这两个运算符的功能也“互换”了。**

反向迭代器用于以逆序的方式遍历容器中的元素。例如：

#include <iostream>

//需要引入 vector 头文件

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>values{ 1,2,3,4,5 };

auto first = values.*rbegin*();

auto end = values.*rend*();

while (first != end)

{

*cout* << \*first << " ";

++first;

}

return 0;

}

**运行结果为：**

**5 4 3 2 1**

可以看到，从最后一个元素开始循环，遍历输出了容器中的所有元素。结束迭代器指向第一个元素之前的位置，所以当 first 指向第一个元素并 +1 后，循环就结朿了。  
当然，在上面程序中，我们也可以使用 [for 循环](http://c.biancheng.net/view/172.html" \t "http://c.biancheng.net/view/_blank)：

for (auto first = values.rbegin(); first != values.rend(); ++first) {

cout << \*first << " ";

}

crbegin()/crend() 组合和 rbegin()/crend() 组合唯一的区别在于，前者返回的迭代器为 const 类型，即不能用来修改容器中的元素，除此之外在使用上和后者完全相同。

**有关 crbegin()/crend() 成员函数，这里不再给出具体实例，有兴趣的读者，可自行编写代码进行测试。**

**二vector容器迭代器的独特之处**

和 array 容器不同，vector 容器可以随着存储元素的增加，自行申请更多的存储空间。因此，在创建 vector 对象时，我们可以直接创建一个空的 vector 容器，并不会影响后续使用该容器。  
 但这会产生一个问题，即在初始化空的 vector 容器时，不能使用迭代器。也就是说，如下初始化 vector 容器的方法是不行的：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>values;

int val = 1;

for (auto first = values.*begin*(); first < values.*end*(); ++first, val++) {

\*first = val;

//初始化的同时输出值

*cout* << \*first;

}

return 0;

}

运行程序可以看到，什么也没有输出。这是因为，对于空的 vector 容器来说，begin() 和 end() 成员函数返回的迭代器是相等的，即它们指向的是同一个位置。

所以，对于空的 vector 容器来说，可以通过调用 push\_back() 或者借助 resize() 成员函数实现初始化容器的目的。  
 除此之外，vector 容器在申请更多内存的同时，容器中的所有元素可能会被复制或移动到新的内存地址，这会导致之前创建的迭代器失效。  
举个例子：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>values{ 1,2,3 };

*cout* << "values 容器首个元素的地址：" << values.*data*() << *endl*;

auto first = values.*begin*();

auto end = values.*end*();

//增加 values 的容量

values.*reserve*(20);

*cout* << "values 容器首个元素的地址：" << values.*data*() << *endl*;

while (first != end) {

*cout* << \*first;

++first;

}

return 0;

}

**运行程序，显示如下信息并崩溃：**

**values 容器首个元素的地址：0096DFE8  
values 容器首个元素的地址：00965560**

可以看到，values 容器在增加容量之后，首个元素的存储地址发生了改变，此时再使用先前创建的迭代器，显然是错误的。因此，为了保险起见，每当 vector 容器的容量发生变化时，我们都要对之前创建的迭代器重新初始化一遍：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>values{ 1,2,3 };

*cout* << "values 容器首个元素的地址：" << values.*data*() << *endl*;

auto first = values.*begin*();

auto end = values.*end*();

//增加 values 的容量

values.*reserve*(20);

*cout* << "values 容器首个元素的地址：" << values.*data*() << *endl*;

first = values.*begin*();

end = values.*end*();

while (first != end) {

*cout* << \*first;

++first;

}

return 0;

}

**运行结果为：**

**values 容器首个元素的地址：0164DBE8  
values 容器首个元素的地址：01645560  
123**

**第十C++ STL vector容器访问元素的几种方式**

学会如何创建并初始化 vector 容器之后，本节继续来学习如何获取（甚至修改）容器中存储的元素。

**一访问vector容器中单个元素**

首先，vector 容器可以向普通数组那样访问存储的元素，甚至对指定下标处的元素进行修改，比如：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int> values{ 1,2,3,4,5 };

//获取容器中首个元素

*cout* << values[0] << *endl*;

//修改容器中下标为 0 的元素的值

values[0] = values[1] + values[2] + values[3] + values[4];

*cout* << values[0] << *endl*;

return 0;

}

**运行结果为：**

**1  
14**

**显然，vector 的索引从 0 开始，这和普通数组一样。通过使用索引，总是可以访问到 vector 容器中现有的元素。**

值得一提的是，容器名[n]这种获取元素的方式，需要确保下标 n 的值不会超过容器的容量（可以通过 capacity() 成员函数获取），否则会发生越界访问的错误。幸运的是，和 array 容器一样，vector 容器也提供了 at() 成员函数，当传给 at() 的索引会造成越界时，会抛出std::out\_of\_range异常。  
举个例子：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int> values{ 1,2,3,4,5 };

//获取容器中首个元素

*cout* << values.*at*(0) << *endl*;

//修改容器中下标为 0 的元素的值

values.*at*(0) = values.*at*(1) + values.*at*(2) + values.*at*(3) + values.*at*(4);

*cout* << values.*at*(0) << *endl*;

//下面这条语句会发生 out\_of\_range 异常

//cout << values.at(5) << endl;

return 0;

}

读者可能有这样一个疑问，即为什么 vector 容器在重载 [] 运算符时，没有实现边界检查的功能呢？答案很简单，因为性能。如果每次访问元素，都去检查索引值，无疑会产生很多开销。当不存在越界访问的可能时，就能避免这种开销。

除此之外，vector 容器还提供了 2 个成员函数，即 front() 和 back()，它们分别返回 vector 容器中第一个和最后一个元素的引用，通过利用这 2 个函数返回的引用，可以访问（甚至修改）容器中的首尾元素。  
举个例子：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int> values{ 1,2,3,4,5 };

*cout* << "values 首元素为：" << values.*front*() << *endl*;

*cout* << "values 尾元素为：" << values.*back*() << *endl*;

//修改首**运行结果为：**

**1  
14**

元素

values.*front*() = 10;

*cout* << "values 新的首元素为：" << values.*front*() << *endl*;

//修改尾元素

values.*back*() = 20;

*cout* << "values 新的尾元素为：" << values.*back*() << *endl*;

return 0;

}

**输出结果为：**

**values 首元素为：1  
values 尾元素为：5  
values 新的首元素为：10  
values 新的尾元素为：20**

另外，vector 容器还提供了 data() 成员函数，该函数的功能是返回指向容器中首个元素的[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)。通过该指针也可以访问甚至修改容器中的元素。比如：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int> values{ 1,2,3,4,5 };

//输出容器中第 3 个元素的值

*cout* << \*(values.*data*() + 2) << *endl*;

//修改容器中第 2 个元素的值

\*(values.*data*() + 1) = 10;

*cout* << \*(values.*data*() + 1) << *endl*;

return 0;

}

**运行结果为：**

**3**

**10**

**二 访问vector容器中多个元素**

如果想访问 vector 容器中多个元素，可以借助 size() 成员函数，该函数可以返回 vector 容器中实际存储的元素个数。例如：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int> values{ 1,2,3,4,5 };

//从下标 0 一直遍历到 size()-1 处

for (int i = 0; i < values.*size*(); i++) {

*cout* << values[i] << " ";

}

return 0;

}

**运行结果为：**

**1 2 3 4 5**

注意，这里不要使用 capacity() 成员函数，因为它返回的是 vector 容器的容量，而不是实际存储元素的个数，这两者是有差别的。

**关于 vector 容器 capacity() 和 size() 的差别，可以阅读 《[STL vector容量（capacity）和大小（size）的区别](http://c.biancheng.net/view/6770.html" \t "http://c.biancheng.net/view/_blank)》一文。**

或者也可以使用基于范围的循环，此方式将会逐个遍历容器中的元素。比如：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int> values{ 1,2,3,4,5 };

for (auto&& value : values)

*cout* << value << " ";

return 0;

}

运行结果为：

1 2 3 4 5

当然，这里也可以使用 rbegin()/rend()、cbegin()/cend()、crbegin()/crend() 以及全局函数 begin()/end() ，它们都可以实现对容器中元素的访问。

**第十一 C++ vector容量（capacity）和大小（size）的区别**

很多初学者分不清楚 vector 容器的容量（capacity）和大小（size）之间的区别，甚至有人认为它们表达的是一个意思。本节将对 vector 容量和大小各自的含义做一个详细的介绍。  
 vector 容器的容量（用 capacity 表示），指的是在不分配更多内存的情况下，容器可以保存的最多元素个数；而 vector 容器的大小（用 size 表示），指的是它实际所包含的元素个数。  
 对于一个 vector 对象来说，通过该模板类提供的 capacity() 成员函数，可以获得当前容器的容量；通过 size() 成员函数，可以获得容器当前的大小。例如：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*std*::*vector*<int>value{ 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47 };

value.*reserve*(20);

*cout* << "value 容量是：" << value.*capacity*() << *endl*;

*cout* << "value 大小是：" << value.*size*() << *endl*;

return 0;

}

**程序输出结果为：**

**value 容量是：20  
value 大小是：15**

结合该程序的输出结果，图 1 可以更好的说明 vector 容器容量和大小之间的关系。

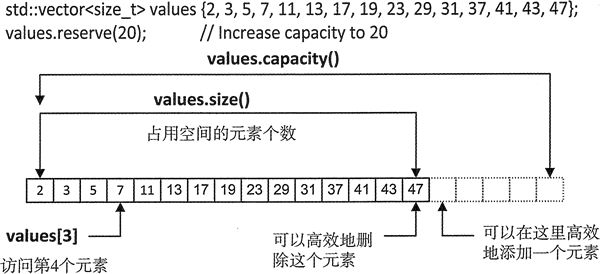


图 1 vector 容量和大小的区别

显然，vector 容器的大小不能超出它的容量，在大小等于容量的基础上，只要增加一个元素，就必须分配更多的内存。注意，这里的“更多”并不是 1 个。换句话说，当 vector 容器的大小和容量相等时，如果再向其添加（或者插入）一个元素，vector 往往会申请多个存储空间，而不仅仅只申请 1 个。

**一旦 vector 容器的内存被重新分配，则和 vector 容器中元素相关的所有引用、指针以及迭代器，都可能会失效，最稳妥的方法就是重新生成。**

举个例子：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>value{ 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47 };

*cout* << "value 容量是：" << value.*capacity*() << *endl*;

*cout* << "value 大小是：" << value.*size*() << *endl*;

*printf*("value首地址：%p\n", value.*data*());

value.*push\_back*(53);

*cout* << "value 容量是(2)：" << value.*capacity*() << *endl*;

*cout* << "value 大小是(2)：" << value.*size*() << *endl*;

*printf*("value首地址： %p", value.*data*());

return 0;

}

**运行结果为：**

**value 容量是：15  
value 大小是：15  
value首地址：01254D40  
value 容量是(2)：22  
value 大小是(2)：16  
value首地址： 01254E80**

可以看到，向“已满”的 vector 容器再添加一个元素，整个 value 容器的存储位置发生了改变，同时 vector 会一次性申请多个存储空间（具体多少，取决于底层算法的实现）。这样做的好处是，可以很大程度上减少 vector 申请空间的次数，当后续再添加元素时，就可以节省申请空间耗费的时间。

**因此，对于 vector 容器而言，当增加新的元素时，有可能很快完成（即直接存在预留空间中）；也有可能会慢一些（扩容之后再放新元素）。**

**一 修改vector容器的容量和大小**

另外，通过前面的学习我们知道，可以调用 reserve() 成员函数来增加容器的容量（但并不会改变存储元素的个数）；而通过调用成员函数 resize() 可以改变容器的大小，并且该函数也可能会导致 vector 容器容量的增加。比如说：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>value{ 2,3,5,7,11,13,17,19,23,29,31,37,41,43,47 };

*cout* << "value 容量是：" << value.*capacity*() << *endl*;

*cout* << "value 大小是：" << value.*size*() << *endl*;

value.*reserve*(20);

*cout* << "value 容量是(2)：" << value.*capacity*() << *endl*;

*cout* << "value 大小是(2)：" << value.*size*() << *endl*;

//将元素个数改变为 21 个，所以会增加 6 个默认初始化的元素

value.*resize*(21);

//将元素个数改变为 21 个，新增加的 6 个元素默认值为 99。

//value.resize(21,99);

//当需要减小容器的大小时，会移除多余的元素。

//value.resize(20);

*cout* << "value 容量是(3)：" << value.*capacity*() << *endl*;

*cout* << "value 大小是(3)：" << value.*size*() << *endl*;

return 0;

}

**运行结果为：**

**value 容量是：15  
value 大小是：15  
value 容量是(2)：20  
value 大小是(2)：15  
value 容量是(3)：30  
value 大小是(3)：21**

程序中给出了关于 resize() 成员函数的 3 种不同的用法，有兴趣的读者可自行查看不同用法的运行结果。

可以看到，仅通过 reserve() 成员函数增加 value 容器的容量，其大小并没有改变；但通过 resize() 成员函数改变 value 容器的大小，它的容量可能会发生改变。另外需要注意的是，通过 resize() 成员函数减少容器的大小（多余的元素会直接被删除），不会影响容器的容量。

**二 vector容器容量和大小的数据类型**

在实际场景中，我们可能需要将容器的容量和大小保存在变量中，要知道 vector<T> 对象的容量和大小类型都是 vector<T>::size\_type 类型。因此，当定义一个变量去保存这些值时，可以如下所示：

vector<int>::size\_type cap = value.capacity();

vector<int>::size\_type size = value.size();

size\_type 类型是定义在由 vector 类模板生成的 vecotr 类中的，它表示的真实类型和操作系统有关，在 32 位架构下普遍表示的是 unsigned int 类型，而在 64 位架构下普通表示 unsigned long 类型。  
当然，我们还可以使用 auto 关键字代替 vector<int>::size\_type，比如：

auto cap = value.capacity();

auto size = value.size();

**第十二 深度剖析C++ vector容器的底层实现机制**

STL 众多容器中，vector 是最常用的容器之一，其底层所采用的数据结构非常简单，就只是一段连续的线性内存空间。  
 通过分析 vector 容器的源代码不难发现，它就是使用 3 个迭代器（可以理解成指针）来表示的：

//\_Alloc 表示内存分配器，此参数几乎不需要我们关心

template <class *\_Ty*, class \_Alloc = *allocator*<*\_Ty*>>

class vector {

...

protected:

*pointer* \_Myfirst;

*pointer* \_Mylast;

*pointer* \_Myend;

};

其中，\_Myfirst 指向的是 vector 容器对象的起始字节位置；\_Mylast 指向当前最后一个元素的末尾字节；\_myend 指向整个 vector 容器所占用内存空间的末尾字节。  
图 1 演示了以上这 3 个迭代器分别指向的位置。

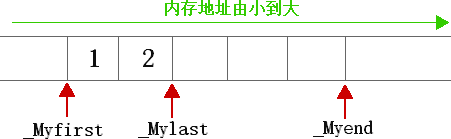


图 1 vector实现原理示意图

如图 1 所示，通过这 3 个迭代器，就可以表示出一个已容纳 2 个元素，容量为 5 的 vector 容器。  
  
在此基础上，将 3 个迭代器两两结合，还可以表达不同的含义，例如：

（1）\_Myfirst 和 \_Mylast 可以用来表示 vector 容器中目前已被使用的内存空间；

（2）\_Mylast 和 \_Myend 可以用来表示 vector 容器目前空闲的内存空间；

（3）\_Myfirst 和 \_Myend 可以用表示 vector 容器的容量。

**对于空的 vector 容器，由于没有任何元素的空间分配，因此 \_Myfirst、\_Mylast 和 \_Myend 均为 null。**

通过灵活运用这 3 个迭代器，vector 容器可以轻松的实现诸如首尾标识、大小、容器、空容器判断等几乎所有的功能，比如：

template <class *\_Ty*, class \_Alloc = *allocator*<*\_Ty*>>

class vector {

public：

*iterator* begin() { return *\_Myfirst*; }

*iterator* end() { return *\_Mylast*; }

*size\_type* size() const { return *size\_type*(end() - begin()); }

*size\_type* capacity() const { return *size\_type*(*\_Myend* - begin()); }

bool empty() const { return begin() == end(); }

*reference* operator[] (*size\_type* n) { return \*(begin() + n); }

*reference* front() { return \*begin(); }

*reference* back() { return \*(end() - 1); }

...

};

**一 vector扩大容量的本质**

另外需要指明的是，当 vector 的大小和容量相等（size==capacity）也就是满载时，如果再向其添加元素，那么 vector 就需要扩容。vector 容器扩容的过程需要经历以下 3 步：

（1）完全弃用现有的内存空间，重新申请更大的内存空间；

（2）将旧内存空间中的数据，按原有顺序移动到新的内存空间中；

（3）最后将旧的内存空间释放。

**这也就解释了，为什么 vector 容器在进行扩容后，与其相关的指针、引用以及迭代器可能会失效的原因。**

由此可见，vector 扩容是非常耗时的。为了降低再次分配内存空间时的成本，每次扩容时 vector 都会申请比用户需求量更多的内存空间（这也就是 vector 容量的由来，即 capacity>=size），以便后期使用。

**vector 容器扩容时，不同的编译器申请更多内存空间的量是不同的。以 VS 为例，它会扩容现有容器容量的 50%。**

**第十三C++ STL vector添加元素（push\_back()和emplace\_back()）详解**

要知道，向 vector 容器中添加元素的唯一方式就是使用它的成员函数，如果不调用成员函数，非成员函数既不能添加也不能删除元素。这意味着，vector 容器对象必须通过它所允许的函数去访问，迭代器显然不行。  
 在 《[STL vector容器详解](http://c.biancheng.net/view/6749.html" \t "http://c.biancheng.net/view/_blank)》一节中，已经给大家列出了 vector 容器提供的所有成员函数，在这些成员函数中，可以用来给容器中添加元素的函数有 2 个，分别是 push\_back() 和 empl**ace\_back() 函数。**

**有读者可能认为还有 insert() 和 emplace() 成员函数，严格意义上讲，这 2 个成员函数的功能是向容器中的指定位置插入元素，后续章节会对它们做详细的介绍。**

**一push\_back()**

该成员函数的功能是在 vector 容器尾部添加一个元素，用法也非常简单，比如：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int> values{};

values.*push\_back*(1);

values.*push\_back*(2);

for (int i = 0; i < values.*size*(); i++) {

*cout* << values[i] << " ";

}

return 0;

}

程序中，第 7 行代码表示向 values 容器尾部添加一个元素，但由于当前 values 容器是空的，因此新添加的元素 1 无疑成为了容器中首个元素；第 8 行代码实现的功能是在现有元素 1 的后面，添加元素 2。  
**运行程序，输出结果为：**

**1 2**

**二emplace\_back()**

该函数是 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 新增加的，其功能和 push\_back() 相同，都是在 vector 容器的尾部添加一个元素。  
emplace\_back() 成员函数的用法也很简单，这里直接举个例子：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int> values{};

values.*emplace\_back*(1);

values.*emplace\_back*(2);

for (int i = 0; i < values.*size*(); i++) {

*cout* << values[i] << " ";

}

return 0;

}

**运行结果为：**

**1 2**

**读者可能会发现，以上 2 段代码，只是用 emplace\_back() 替换了 push\_back()，既然它们实现的功能是一样的，那么 C++ 11 标准中为什么要多此一举呢？**

**三 emplace\_back()和push\_back()的区别**

emplace\_back() 和 push\_back() 的区别，就在于底层实现的机制不同。push\_back() 向容器尾部添加元素时，首先会创建这个元素，然后再将这个元素拷贝或者移动到容器中（如果是拷贝的话，事后会自行销毁先前创建的这个元素）；而 emplace\_back() 在实现时，则是直接在容器尾部创建这个元素，省去了拷贝或移动元素的过程。  
 为了让大家清楚的了解它们之间的区别，我们创建一个包含类对象的 vector 容器，如下所示：

#include <vector>

#include <iostream>

using namespace *std*;

class testDemo

{

public:

testDemo(int num) :num(num) {

*std*::*cout* << "调用构造函数" << *endl*;

}

testDemo(const testDemo& other) :num(other.num) {

*std*::*cout* << "调用拷贝构造函数" << *endl*;

}

testDemo(testDemo&& other) :num(other.num) {

*std*::*cout* << "调用移动构造函数" << *endl*;

}

private:

int num;

};

int main()

{

*cout* << "emplace\_back:" << *endl*;

*std*::*vector*<testDemo> demo1;

demo1.*emplace\_back*(2);

*cout* << "push\_back:" << *endl*;

*std*::*vector*<testDemo> demo2;

demo2.*push\_back*(2);

}

**运行结果为：**

**emplace\_back:  
调用构造函数  
push\_back:  
调用构造函数  
调用移动构造函数**

在此基础上，读者可尝试将 testDemo 类中的移动构造函数注释掉，再运行程序会发现，运行结果变为：

**emplace\_back:  
调用构造函数  
push\_back:  
调用构造函数  
调用拷贝构造函数**

由此可以看出，push\_back() 在底层实现时，会优先选择调用移动构造函数，如果没有才会调用拷贝构造函数。  
**显然完成同样的操作，push\_back() 的底层实现过程比 emplace\_back() 更繁琐，换句话说，emplace\_back() 的执行效率比 push\_back() 高。因此，在实际使用时，建议大家优先选用 emplace\_back()。**

**第十四 C++ STL vector插入元素（insert()和emplace()）详解**

vector容器提供了 insert() 和 emplace() 这 2 个成员函数，用来实现在容器指定位置处插入元素，本节将对它们的用法做详细的讲解。

**另外，如果想实现在 vector 容器尾部添加元素，可阅读《[vector添加元素](http://c.biancheng.net/view/6826.html" \t "http://c.biancheng.net/view/_blank)》一节。**

**一 insert()**

insert() 函数的功能是在 vector 容器的指定位置插入一个或多个元素。该函数的语法格式有多种，如表 1 所示。

表 1 insert() 成员函数语法格式

|  |  |
| --- | --- |
| **语法格式** | **用法说明** |
| iterator insert(pos,elem) | 在迭代器 pos 指定的位置之前插入一个新元素elem，并返回表示新插入元素位置的迭代器。 |
| iterator insert(pos,n,elem) | 在迭代器 pos 指定的位置之前插入 n 个元素 elem，并返回表示第一个新插入元素位置的迭代器。 |
| iterator insert(pos,first,last) | 在迭代器 pos 指定的位置之前，插入其他容器（不仅限于vector）中位于 [first,last) 区域的所有元素，并返回表示第一个新插入元素位置的迭代器。 |
| iterator insert(pos,initlist) | 在迭代器 pos 指定的位置之前，插入初始化列表（用大括号{}括起来的多个元素，中间有逗号隔开）中所有的元素，并返回表示第一个新插入元素位置的迭代器。 |

下面的例子，演示了如何使用 insert() 函数向 vector 容器中插入元素。

#include <iostream>

#include <vector>

#include <array>

using namespace *std*;

int main()

{

*std*::*vector*<int> demo{ 1,2 };

//第一种格式用法

demo.*insert*(demo.*begin*() + 1, 3);//{1,3,2}

//第二种格式用法

demo.*insert*(demo.*end*(), 2, 5);//{1,3,2,5,5}

//第三种格式用法

*std*::array<int, 3>test{ 7,8,9 };

demo.*insert*(demo.*end*(), test.*begin*(), test.*end*());//{1,3,2,5,5,7,8,9}

//第四种格式用法

demo.*insert*(demo.*end*(), { 10,11 });//{1,3,2,5,5,7,8,9,10,11}

for (int i = 0; i < demo.*size*(); i++) {

*cout* << demo[i] << " ";

}

return 0;

}

**运行结果为：**

**1 3 2 5 5 7 8 9 10 11**

**二 emplace()**

emplace() 是 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 标准新增加的成员函数，用于在 vector 容器指定位置之前插入一个新的元素。

**再次强调，emplace() 每次只能插入一个元素，而不是多个。**

该函数的语法格式如下：

**iterator emplace (const\_iterator pos, args...);**

其中，pos 为指定插入位置的迭代器；args... 表示与新插入元素的构造函数相对应的多个参数；该函数会返回表示新插入元素位置的迭代器。

**简单的理解 args...，即被插入元素的构造函数需要多少个参数，那么在 emplace() 的第一个参数的后面，就需要传入相应数量的参数。**

#include <vector>

#include <iostream>

using namespace *std*;

int main()

{

*std*::*vector*<int> demo1{ 1,2 };

//emplace() 每次只能插入一个 int 类型元素

demo1.*emplace*(demo1.*begin*(), 3);

for (int i = 0; i < demo1.*size*(); i++) {

*cout* << demo1[i] << " ";

}

return 0;

}

**运行结果为：**

**3 1 2**

既然 emplace() 和 insert() 都能完成向 vector 容器中插入新元素，那么谁的运行效率更高呢？答案是 emplace()。在说明原因之前，通过下面这段程序，就可以直观看出两者运行效率的差异：

#include <vector>

#include <iostream>

using namespace *std*;

class testDemo

{

public:

testDemo(int num) :num(num) {

*std*::*cout* << "调用构造函数" << *endl*;

}

testDemo(const testDemo& other) :num(other.num) {

*std*::*cout* << "调用拷贝构造函数" << *endl*;

}

testDemo(testDemo&& other) :num(other.num) {

*std*::*cout* << "调用移动构造函数" << *endl*;

}

testDemo& operator=(const testDemo& other);

private:

int num;

};

testDemo& testDemo::operator=(const testDemo& other) {

this->num = other.num;

return \*this;

}

int main()

{

*cout* << "insert:" << *endl*;

*std*::*vector*<testDemo> demo2{};

demo2.*insert*(demo2.*begin*(), testDemo(1));

*cout* << "emplace:" << *endl*;

*std*::*vector*<testDemo> demo1{};

demo1.*emplace*(demo1.*begin*(), 1);

return 0;

}

**运行结果为：**

**insert:  
调用构造函数  
调用移动构造函数  
emplace:  
调用构造函数**

**注意，当拷贝构造函数和移动构造函数同时存在时，insert() 会优先调用移动构造函数。**

可以看到，通过 insert() 函数向 vector 容器中插入 testDemo 类对象，需要调用类的构造函数和移动构造函数（或拷贝构造函数）；而通过 emplace() 函数实现同样的功能，只需要调用构造函数即可。  
**简单的理解，就是 emplace() 在插入元素时，是在容器的指定位置直接构造元素，而不是先单独生成，再将其复制（或移动）到容器中。因此，在实际使用中，推荐大家优先使用 emplace()。**

**第十五C++ STL vector删除元素的几种方式**

前面提到，无论是向现有 vector 容器中访问元素、添加元素还是插入元素，都只能借助 vector 模板类提供的成员函数，但删除 vector 容器的元素例外，完成此操作除了可以借助本身提供的成员函数，还可以借助一些全局函数。  
 基于不同场景的需要，删除 vecotr 容器的元素，可以使用表 1 中所示的函数（或者函数组合）。

表 1 删除 vector 容器元素的几种方式

|  |  |
| --- | --- |
| **函数** | **功能** |
| pop\_back() | 删除 vector 容器中最后一个元素，该容器的大小（size）会减 1，但容量（capacity）不会发生改变。 |
| erase(pos) | 删除 vector 容器中 pos 迭代器指定位置处的元素，并返回指向被删除元素下一个位置元素的迭代器。该容器的大小（size）会减 1，但容量（capacity）不会发生改变。 |
| swap(beg)、pop\_back() | 先调用 swap() 函数交换要删除的目标元素和容器最后一个元素的位置，然后使用 pop\_back() 删除该目标元素。 |
| erase(beg,end) | 删除 vector 容器中位于迭代器 [beg,end)指定区域内的所有元素，并返回指向被删除区域下一个位置元素的迭代器。该容器的大小（size）会减小，但容量（capacity）不会发生改变。 |
| remove() | 删除容器中所有和指定元素值相等的元素，并返回指向最后一个元素下一个位置的迭代器。值得一提的是，调用该函数不会改变容器的大小和容量。 |
| clear() | 删除 vector 容器中所有的元素，使其变成空的 vector 容器。该函数会改变 vector 的大小（变为 0），但不是改变其容量。 |

下面就表 1 中罗列的这些函数，一一讲解它们的具体用法。  
pop\_back() 成员函数的用法非常简单，它不需要传入任何的参数，也没有返回值。举个例子：

#include <vector>

#include <iostream>

using namespace *std*;

int main()

{

*vector*<int>demo{ 1,2,3,4,5 };

demo.*pop\_back*();

//输出 dmeo 容器新的size

*cout* << "size is :" << demo.*size*() << *endl*;

//输出 demo 容器新的容量

*cout* << "capacity is :" << demo.*capacity*() << *endl*;

for (int i = 0; i < demo.*size*(); i++) {

*cout* << demo[i] << " ";

}

return 0;

}

**运行结果为：**

**size is :4  
capacity is :5  
1 2 3 4**

可以发现，相比原 demo 容器，新的 demo 容器删除了最后一个元素 5，容器的大小减了 1，但容量没变。  
如果想删除 vector 容器中指定位置处的元素，可以使用 erase() 成员函数，该函数的语法格式为：

**iterator erase (pos);**

其中，pos 为指定被删除元素位置的迭代器，同时该函数会返回一个指向删除元素所在位置下一个位置的迭代器。  
下面的例子演示了 erase() 函数的具体用法：

#include <vector>

#include <iostream>

using namespace *std*;

int main()

{

*vector*<int>demo{ 1,2,3,4,5 };

auto iter = demo.*erase*(demo.*begin*() + 1);//删除元素 2

//输出 dmeo 容器新的size

*cout* << "size is :" << demo.*size*() << *endl*;

//输出 demo 容器新的容量

*cout* << "capacity is :" << demo.*capacity*() << *endl*;

for (int i = 0; i < demo.*size*(); i++) {

*cout* << demo[i] << " ";

}

//iter迭代器指向元素 3

*cout* << *endl* << \*iter << *endl*;

return 0;

}

**运行结果为：**

**size is :4  
capacity is :5  
1 3 4 5  
3**

通过结果不能看出，erase() 函数在删除元素时，会将删除位置后续的元素陆续前移，并将容器的大小减 1。  
 另外，如果不在意容器中元素的排列顺序，可以结合 swap() 和 pop\_back() 函数，同样可以实现删除容器中指定位置元素的目的。

**注意，swap() 函数在头文件 <algorithm> 和 <utility> 中都有定义，使用时引入其中一个即可。**

例如：

#include <vector>

#include <iostream>

#include <algorithm>

using namespace *std*;

int main()

{

*vector*<int>demo{ 1,2,3,4,5 };

//交换要删除元素和最后一个元素的位置

*swap*(\*(*std*::*begin*(demo) + 1), \*(*std*::*end*(demo) - 1));//等同于 swap(demo[1],demo[4])

//交换位置后的demo容器

for (int i = 0; i < demo.*size*(); i++) {

*cout* << demo[i] << " ";

}

demo.*pop\_back*();

*cout* << *endl* << "size is :" << demo.*size*() << *endl*;

*cout* << "capacity is :" << demo.*capacity*() << *endl*;

//输出demo 容器中剩余的元素

for (int i = 0; i < demo.*size*(); i++) {

*cout* << demo[i] << " ";

}

return 0;

}

**运行结果为：**

**1 5 3 4 2  
size is :4  
capacity is :5  
1 5 3 4**  
 当然，除了删除容器中单个元素，还可以删除容器中某个指定区域内的所有元素，同样可以使用 erase() 成员函数实现。该函数有 2 种基本格式，前面介绍了一种，这里使用另一种：

**iterator erase (iterator first, iterator last);**

**其中 first 和 last 是指定被删除元素区域的迭代器，同时该函数会返回指向此区域之后一个位置的迭代器。**  
举个例子：

#include <vector>

#include <iostream>

using namespace *std*;

int main()

{

*std*::*vector*<int> demo{ 1,2,3,4,5 };

//删除 2、3

auto iter = demo.*erase*(demo.*begin*() + 1, demo.*end*() - 2);

*cout* << "size is :" << demo.*size*() << *endl*;

*cout* << "capacity is :" << demo.*capacity*() << *endl*;

for (int i = 0; i < demo.*size*(); i++) {

*cout* << demo[i] << " ";

}

return 0;

}

**运行结果为：**

**size is :3  
capacity is :5  
1 4 5**

**可以看到**，和删除单个元素一样，删除指定区域内的元素时，也会将该区域后续的元素前移，并缩小容器的大小。  
如果要删除容器中和指定元素值相同的所有元素，可以使用 remove() 函数，该函数定义在 <algorithm> 头文件中。例如：

#include <vector>

#include <iostream>

#include <algorithm>

using namespace *std*;

int main()

{

*vector*<int>demo{ 1,3,3,4,3,5 };

//交换要删除元素和最后一个元素的位置

auto iter = *std*::*remove*(demo.*begin*(), demo.*end*(), 3);

*cout* << "size is :" << demo.*size*() << *endl*;

*cout* << "capacity is :" << demo.*capacity*() << *endl*;

//输出剩余的元素

for (auto first = demo.*begin*(); first < iter; ++first) {

*cout* << \*first << " ";

}

return 0;

}

**运行结果为：**

**size is :6  
capacity is :6  
1 4 5**

注意，在对容器执行完 remove() 函数之后，由于该函数并没有改变容器原来的大小和容量，因此无法使用之前的方法遍历容器，而是需要向程序中那样，借助 remove() 返回的迭代器完成正确的遍历。

**remove() 的实现原理是，在遍历容器中的元素时，一旦遇到目标元素，就做上标记，然后继续遍历，直到找到一个非目标元素，即用此元素将最先做标记的位置覆盖掉，同时将此非目标元素所在的位置也做上标记，等待找到新的非目标元素将其覆盖。因此，如果将上面程序中 demo 容器的元素全部输出，得到的结果为 1 4 5 4 3 5。**

另外还可以看到，既然通过 remove() 函数删除掉 demo 容器中的多个指定元素，该容器的大小和容量都没有改变，其剩余位置还保留了之前存储的元素。我们可以使用 erase() 成员函数删掉这些 "无用" 的元素。  
比如，修改上面的程序：

#include <vector>

#include <iostream>

#include <algorithm>

using namespace *std*;

int main()

{

*vector*<int>demo{ 1,3,3,4,3,5 };

//交换要删除元素和最后一个元素的位置

auto iter = *std*::*remove*(demo.*begin*(), demo.*end*(), 3);

demo.*erase*(iter, demo.*end*());

*cout* << "size is :" << demo.*size*() << *endl*;

*cout* << "capacity is :" << demo.*capacity*() << *endl*;

//输出剩余的元素

for (int i = 0; i < demo.*size*(); i++) {

*cout* << demo[i] << " ";

}

return 0;

}

**运行结果为：**

**size is :3  
capacity is :6  
1 4 5**

**remove()用于删除容器中指定元素时，常和 erase() 成员函数搭配使用。**  
如果想删除容器中所有的元素，则可以使用 clear() 成员函数，例如：

#include <vector>

#include <iostream>

#include <algorithm>

using namespace *std*;

int main()

{

*vector*<int>demo{ 1,3,3,4,3,5 };

//交换要删除元素和最后一个元素的位置

demo.*clear*();

*cout* << "size is :" << demo.*size*() << *endl*;

*cout* << "capacity is :" << demo.*capacity*() << *endl*;

return 0;

}

**运行结果为：**

**size is :0  
capacity is :6**

**第十六 如何避免vector容器进行不必要的扩容？**

前面提到，我们可以将 vector 容器看做是一个动态数组。换句话说，在不超出 vector 最大容量限制（max\_size() 成员方法的返回值）的前提下，该类型容器可以自行扩充容量来满足用户存储更多元素的需求。  
 值得一提的是，vector 容器扩容的整个过程，和 realloc() 函数的实现方法类似，大致分为以下 4 个步骤：

（1）分配一块大小是当前 vector 容量几倍的新存储空间。注意，多数 STL 版本中的 vector 容器，其容器都会以 2 的倍数增长，也就是说，每次 vector 容器扩容，它们的容量都会提高到之前的 2 倍；

（2）将 vector 容器存储的所有元素，依照原有次序从旧的存储空间复制到新的存储空间中；

（3）析构掉旧存储空间中存储的所有元素；

（4）释放旧的存储空间。

通过以上分析不难看出，vector 容器的扩容过程是非常耗时的，并且当容器进行扩容后，之前和该容器相关的所有指针、迭代器以及引用都会失效。因此在使用 vector 容器过程中，我们应尽量避免执行不必要的扩容操作。  
要实现这个目标，可以借助 vector 模板类中提供的 reserve() 成员方法。不过在讲解如何用 reserve() 方法避免 vector 容器进行不必要的扩容操作之前，vector 模板类中还提供有几个和 reserve() 功能类似的成员方法，很容易混淆，这里有必要为读者梳理一下，如表 1 所示。

表 1 vector模板类中功能类似的成员方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| size() | 告诉我们当前 vector 容器中已经存有多少个元素，但仅通过此方法，无法得知 vector 容器有多少存储空间。 |
| capacity() | 告诉我们当前 vector 容器总共可以容纳多少个元素。如果想知道当前 vector 容器有多少未被使用的存储空间，可以通过 capacity()-size() 得知。注意，如果 size() 和 capacity() 返回的值相同，则表明当前 vector 容器中没有可用存储空间了，这意味着，下一次向 vector 容器中添加新元素，将导致 vector 容器扩容。 |
| resize(n) | 强制 vector 容器必须存储 n 个元素，注意，如果 n 比 size() 的返回值小，则容器尾部多出的元素将会被析构（删除）；如果 n 比 size() 大，则 vector 会借助默认构造函数创建出更多的默认值元素，并将它们存储到容器末尾；如果 n 比 capacity() 的返回值还要大，则 vector 会先扩增，在添加一些默认值元素。 |
| reserve(n) | 强制 vector 容器的容量至少为 n。注意，如果 n 比当前 vector 容器的容量小，则该方法什么也不会做；反之如果 n 比当前 vector 容器的容量大，则 vector 容器就会扩容。 |

通过对以上几个成员方法功能的分析，我们可以总结出一点，即只要有新元素要添加到 vector 容器中而恰好此时 vector 容器的容量不足时，该容器就会自动扩容。  
 因此，避免 vector 容器执行不必要的扩容操作的关键在于，在使用 vector 容器初期，就要将其容量设为足够大的值。换句话说，在 vector 容器刚刚构造出来的那一刻，就应该借助 reserve() 成员方法为其扩充足够大的容量。  
举个例子，假设我们想创建一个包含 1~1000 的 vector<int>，通常会这样实现：

**vector<int>myvector;**

**for (int i = 1; i <= 1000; i++) {**

**myvector.push\_back(i);**

**}**

值得一提的是，上面代码的整个循环过程中，vector 容器会进行 2~10 次自动扩容（多数的 STL 标准库版本中，vector 容器通常会扩容至当前容量的 2 倍，而这里 1000≈2 10），程序的执行效率可想而知。  
 在上面程序的基础上，下面代码演示了如何使用 reserve() 成员方法尽量避免 vector 容器执行不必要的扩容操作：

**vector<int>myvector;**

**myvector.reserve(1000);**

**cout << myvector.capacity();**

**for (int i = 1; i <= 1000; i++) {**

**myvector.push\_back(i);**

**}**

相比前面的代码实现，整段程序在运行过程中，vector 容器的容量仅扩充了 1 次，执行效率大大提高。  
 当然在实际场景中，我们可能并不知道 vector 容器到底要存储多少个元素。这种情况下，可以先预留出足够大的空间，当所有元素都存储到 vector 容器中之后，再去除多余的容量。

**关于怎样去除 vector 容器多余的容量，可以借助该容器模板类提供的 shrink\_to\_fit() 成员方法，另外后续还会讲解如何使用 swap() 成员方法去除 vector 容器多余的容量，两种方法都可以。**

**第十七vector swap()成员方法还可以这样用！**

《[如何避免vector容器进行不必要的扩容](http://c.biancheng.net/view/vip_7711.html" \t "http://c.biancheng.net/view/_blank)》一节中，遗留了一个问题，即如何借助 swap() 成员方法去除 vector 容器中多余的容量？本节将就此问题给读者做详细的讲解。  
 我们知道，在使用 vector 容器的过程中，其容器会根据需要自行扩增。比如，使用 push\_back()、insert()、emplace() 等成员方法向 vector 容器中添加新元素时，如果当前容器已满（即 size() == capacity()），则它会自行扩容以满足添加新元素的需求。当然，还可以调用 reserve() 成员方法来手动提升当前 vector 容器的容量。  
  
举个例子（程序一）：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>myvector;

*cout* << "1、当前 myvector 拥有 " << myvector.*size*() << " 个元素，容量为 " << myvector.*capacity*() << *endl*;

//利用 myvector 容器存储 10 个元素

for (int i = 1; i <= 10; i++) {

myvector.*push\_back*(i);

}

*cout* << "2、当前 myvector 拥有 " << myvector.*size*() << " 个元素，容量为 " << myvector.*capacity*() << *endl*;

//手动为 myvector 扩容

myvector.*reserve*(1000);

*cout* << "3、当前 myvector 拥有 " << myvector.*size*() << " 个元素，容量为 " << myvector.*capacity*() << *endl*;

return 0;

}

**程序执行结果为：**

**1、当前 myvector 拥有 0 个元素，容量为 0  
2、当前 myvector 拥有 10 个元素，容量为 13  
3、当前 myvector 拥有 10 个元素，容量为 1000**  
 除了可以添加元素外，vector 模板类中还提供了 pop\_back()、erase()、clear() 等成员方法，可以轻松实现删除容器中已存储的元素。但需要注意得是，借助这些成员方法只能删除指定的元素，容器的容量并不会因此而改变。  
 例如在程序一的基础上，末尾（return 0 之前）添加如下语句：

myvector.erase(myvector.begin());

cout << "4、当前 myvector 拥有 " << myvector.size() << " 个元素，容量为 " << myvector.capacity() << endl;

myvector.pop\_back();

cout << "5、当前 myvector 拥有 " << myvector.size() << " 个元素，容量为 " << myvector.capacity() << endl;

myvector.clear();

cout << "6、当前 myvector 拥有 " << myvector.size() << " 个元素，容量为 " << myvector.capacity() << endl;

此段代码的执行结果为：

4、当前 myvector 拥有 9 个元素，容量为 1000  
5、当前 myvector 拥有 8 个元素，容量为 1000  
6、当前 myvector 拥有 0 个元素，容量为 1000

显然，myvector 容器存储的元素个数在减少，但容量并不会减小。  
  
幸运的是，myvector 模板类中提供有一个 shrink\_to\_fit() 成员方法，该方法的功能是将当前 vector 容器的容量缩减至和实际存储元素的个数相等。例如，在程序一的基础上，添加如下语句：

myvector.shrink\_to\_fit();

cout << "7、当前 myvector 拥有 " << myvector.size() << " 个元素，容量为 " << myvector.capacity() << endl;

该语句的执行结果为：

7、当前 myvector 拥有 10 个元素，容量为 10

**显然，myvector 容器的容量由 1000 缩减到了 10。**

**一利用swap()方法去除vector多余容量**

除此之外，vector 模板类中还提供有 swap() 成员方法，该方法的基础功能是交换 2 个相同类型的 vector 容器（交换容量和存储的所有元素），但其也能用于去除 vector 容器多余的容量。  
 如果想用 swap() 成员方法去除当前 vector 容器多余的容量时，可以套用如下的语法格式：

vector<T>(x).swap(x);

其中，x 指当前要操作的容器，T 为该容器存储元素的类型。  
下面程序演示了此语法格式的 swap() 方法的用法和功能：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>myvector;

//手动为 myvector 扩容

myvector.*reserve*(1000);

*cout* << "1、当前 myvector 拥有 " << myvector.*size*() << " 个元素，容量为 " << myvector.*capacity*() << *endl*;

//利用 myvector 容器存储 10 个元素

for (int i = 1; i <= 10; i++) {

myvector.*push\_back*(i);

}

//将 myvector 容量缩减至 10

*vector*<int>(myvector).*swap*(myvector);

*cout* << "2、当前 myvector 拥有 " << myvector.*size*() << " 个元素，容量为 " << myvector.*capacity*() << *endl*;

return 0;

}

**程序执行结果为：**

**1、当前 myvector 拥有 0 个元素，容量为 1000  
2、当前 myvector 拥有 10 个元素，容量为 10**

显然，第 16 行代码成功将 myvector 容器的容量 1000 修改为 10，此行代码的执行流程可细分为以下 3 步：  
1) 先执行 vector<int>(myvector)，此表达式会调用 vector 模板类中的拷贝构造函数，从而创建出一个临时的 vector 容器（后续称其为 tempvector）。  
值得一提的是，tempvector 临时容器并不为空，因为我们将 myvector 作为参数传递给了复制构造函数，该函数会将 myvector 容器中的所有元素拷贝一份，并存储到 tempvector 临时容器中。

**注意，vector 模板类中的拷贝构造函数只会为拷贝的元素分配存储空间。换句话说，tempvector 临时容器中没有空闲的存储空间，其容量等于存储元素的个数。**  
2) 然后借助 swap() 成员方法对 tempvector 临时容器和 myvector 容器进行调换，此过程不仅会交换 2 个容器存储的元素，还会交换它们的容量。换句话说经过 swap() 操作，myvetor 容器具有了 tempvector 临时容器存储的所有元素和容量，同时 tempvector 也具有了原 myvector 容器存储的所有元素和容量。  
3) 当整条语句执行结束时，临时的 tempvector 容器会被销毁，其占据的存储空间都会被释放。注意，这里释放的其实是原 myvector 容器占用的存储空间。  
经过以上 3 个步骤，就成功的将 myvector 容器的容量由 100 缩减至 10。

利用swap()方法清空vector容器

在以上内容的学习过程中，如果读者善于举一反三，应该不难想到，swap() 方法还可以用来清空 vector 容器。  
 当 swap() 成员方法用于清空 vector 容器时，可以套用如下的语法格式：

vector<T>().swap(x);

其中，x 指当前要操作的容器，T 为该容器存储元素的类型。  
**注意，和上面语法格式唯一的不同之处在于，这里没有为 vector<T>() 表达式传递任何参数。这意味着，此表达式将调用 vector 模板类的默认构造函数，而不再是复制构造函数。也就是说，此格式会先生成一个空的 vector 容器，再借助 swap() 方法将空容器交换给 x，从而达到清空 x 的目的。**  
下面程序演示了此语法格式的 swap() 方法的用法和功能：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>myvector;

//手动为 myvector 扩容

myvector.*reserve*(1000);

*cout* << "1、当前 myvector 拥有 " << myvector.*size*() << " 个元素，容量为 " << myvector.*capacity*() << *endl*;

//利用 myvector 容器存储 10 个元素

for (int i = 1; i <= 10; i++) {

myvector.*push\_back*(i);

}

//清空 myvector 容器

*vector*<int>().*swap*(myvector);

*cout* << "2、当前 myvector 拥有 " << myvector.*size*() << " 个元素，容量为 " << myvector.*capacity*() << *endl*;

return 0;

}

**程序执行结果为：**

**1、当前 myvector 拥有 0 个元素，容量为 1000  
2、当前 myvector 拥有 0 个元素，容量为 0**

**第十八切忌，vector<bool>不是存储bool类型元素的vector容器！**

前面章节中，已经详细介绍了 vector<T> 容器的功能和用法。特别需要提醒的是，在使用 vector 容器时，要尽量避免使用该容器存储 bool 类型的元素，即避免使用 vector<bool>。  
  
具体来讲，不推荐使用 vector<bool> 的原因有以下 2 个：

（1）严格意义上讲，vector<bool> 并不是一个 STL 容器；

（2）vector<bool> 底层存储的并不是 bool 类型值。

读者可能会感到有些困惑，别着急，继续往下读。

**一vector<bool>不是容器**

值得一提的是，对于是否为 STL 容器，C++ 标准库中有明确的判断条件，其中一个条件是：如果 cont 是包含对象 T 的 STL 容器，且该容器中重载了 [ ] 运算符（即支持 operator[]），则以下代码必须能够被编译：

**T \*p = &cont[0];**

此行代码的含义是，借助 operator[ ] 获取一个 cont<T> 容器中存储的 T 对象，同时将这个对象的地址赋予给一个 T 类型的指针。  
  
这就意味着，如果 vector<bool> 是一个 STL 容器，则下面这段代码是可以通过编译的：

**//创建一个 vector<bool> 容器**

**vector<bool>cont{0,1};**

**//试图将指针 p 指向 cont 容器中第一个元素**

**bool \*p = &cont[0];**

但不幸的是，此段代码不能通过编译。原因在于 vector<bool> 底层采用了独特的存储机制。  
 实际上，为了节省空间，vector<bool> 底层在存储各个 bool 类型值时，每个 bool 值都只使用一个比特位（二进制位）来存储。也就是说在 vector<bool> 底层，一个字节可以存储 8 个 bool 类型值。在这种存储机制的影响下，operator[ ] 势必就需要返回一个指向单个比特位的引用，但显然这样的引用是不存在的。

**C++ 标准中解决这个问题的方案是，令 operator[] 返回一个代理对象（proxy object）。有关代理对象，由于不是本节重点，这里不再做描述，有兴趣的读者可自行查阅相关资料。**

同样对于指针来说，其指向的最小单位是字节，无法另其指向单个比特位。综上所述可以得出一个结论，即上面第 2 行代码中，用 = 赋值号连接 bool \*p 和 &cont[0] 是矛盾的。  
 由于 vector<bool> 并不完全满足 C++ 标准中对容器的要求，所以严格意义上来说它并不是一个 STL 容器。可能有读者会问，既然 vector<bool> 不完全是一个容器，为什么还会出现在 C++ 标准中呢？  
 这和一个雄心勃勃的试验有关，还要从前面提到的代理对象开始说起。由于代理对象在 C++ 软件开发中很受欢迎，引起了 C++ 标准委员会的注意，他们决定以开发 vector<bool> 作为一个样例，来演示 STL 中的容器如何通过代理对象来存取元素，这样当用户想自己实现一个基于代理对象的容器时，就会有一个现成的参考模板。  
 然而开发人员在实现 vector<bool> 的过程中发现，既要创建一个基于代理对象的容器，同时还要求该容器满足 C++ 标准中对容器的所有要求，是不可能的。由于种种原因，这个试验最终失败了，但是他们所做过的尝试（即开发失败的 vector<bool>）遗留在了 C++ 标准中。

**至于将 vector<bool> 遗留到 C++ 标准中，是无心之作，还是有意为之，这都无关紧要，重要的是让读者明白，vector<bool> 不完全满足 C++ 标准中对容器的要求，尽量避免在实际场景中使用它！**

**二如何避免使用vector<bool>**

那么，如果在实际场景中需要使用 vector<bool> 这样的存储结构，该怎么办呢？很简单，可以选择使用 deque<bool> 或者 bitset 来替代 vector<bool>。  
  
 要知道，deque 容器几乎具有 vecotr 容器全部的功能（拥有的成员方法也仅差 reserve() 和 capacity()），而且更重要的是，deque 容器可以正常存储 bool 类型元素。

**有关 deque 容器的具体用法，后续章节会做详细讲解。**

还可以考虑用 bitset 代替 vector<bool>，其本质是一个模板类，可以看做是一种类似数组的存储结构。和后者一样，bitset 只能用来存储 bool 类型值，且底层存储机制也采用的是用一个比特位来存储一个 bool 值。  
 和 vector 容器不同的是，bitset 的大小在一开始就确定了，因此不支持插入和删除元素；另外 bitset 不是容器，所以不支持使用迭代器。

**有关 bitset 的用法，感兴趣的读者可查阅 C++ 官方提供的 [bitset使用手册](http://www.cplusplus.com/reference/bitset/bitset/" \t "http://c.biancheng.net/view/_blank)。**

**第十九C++ STL deque容器（详解版）**

deque 是 double-ended queue 的缩写，又称双端队列容器。  
  
前面章节中，我们已经系统学习了 vector 容器，值得一提的是，deque 容器和 vecotr 容器有很多相似之处，比如：

**（1）deque 容器也擅长在序列尾部添加或删除元素（时间复杂度为O(1)），而不擅长在序列中间添加或删除元素。**

**（2）deque 容器也可以根据需要修改自身的容量和大小。**

和 vector 不同的是，deque 还擅长在序列头部添加或删除元素，所耗费的时间复杂度也为常数阶O(1)。并且更重要的一点是，deque 容器中存储元素并不能保证所有元素都存储到连续的内存空间中。

**当需要向序列两端频繁的添加或删除元素时，应首选 deque 容器。**

deque 容器以模板类 deque<T>（T 为存储元素的类型）的形式在 <deque> 头文件中，并位于 std 命名空间中。因此，在使用该容器之前，代码中需要包含下面两行代码：

#include <deque>

u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;

注意，std 命名空间也可以在使用 deque 容器时额外注明，两种方式都可以。

**一创建deque容器的几种方式**

创建 deque 容器，根据不同的实际场景，可选择使用如下几种方式。  
  
1) 创建一个没有任何元素的空 deque 容器：

std::deque<int> d;

和空 array 容器不同，空的 deque 容器在创建之后可以做添加或删除元素的操作，因此这种简单创建 deque 容器的方式比较常见。  
  
2) 创建一个具有 n 个元素的 deque 容器，其中每个元素都采用对应类型的默认值：

std::deque<int> d(10);

此行代码创建一个具有 10 个元素（默认都为 0）的 deque 容器。  
  
3) 创建一个具有 n 个元素的 deque 容器，并为每个元素都指定初始值，例如：

std::deque<int> d(10, 5)

如此就创建了一个包含 10 个元素（值都为 5）的 deque 容器。  
  
4) 在已有 deque 容器的情况下，可以通过拷贝该容器创建一个新的 deque 容器，例如：

std::deque<int> d1(5);

std::deque<int> d2(d1);

注意，采用此方式，必须保证新旧容器存储的元素类型一致。  
  
5) 通过拷贝其他类型容器中指定区域内的元素（也可以是普通数组），可以创建一个新容器，例如：

**//拷贝普通数组，创建deque容器**

**int a[] = { 1,2,3,4,5 };**

**std::deque<int>d(a, a + 5);**

**//适用于所有类型的容器**

**std::array<int, 5>arr{ 11,12,13,14,15 };**

**std::deque<int>d(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15}**

**二deque容器可利用的成员函数**

基于 deque 双端队列的特点，该容器包含一些 array、vector 容器都没有的成员函数。

表 1 中罗列了 deque 容器提供的所有成员函数。

表 1 deque 容器的成员函数

|  |  |
| --- | --- |
| **函数成员** | **函数功能** |
| begin() | 返回指向容器中第一个元素的迭代器。 |
| end() | 返回指向容器最后一个元素所在位置后一个位置的迭代器，通常和 begin() 结合使用。 |
| rbegin() | 返回指向最后一个元素的迭代器。 |
| rend() | 返回指向第一个元素所在位置前一个位置的迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| size() | 返回实际元素个数。 |
| max\_size() | 返回容器所能容纳元素个数的最大值。这通常是一个很大的值，一般是 232-1，我们很少会用到这个函数。 |
| resize() | 改变实际元素的个数。 |
| empty() | 判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。 |
| shrink \_to\_fit() | 将内存减少到等于当前元素实际所使用的大小。 |
| at() | 使用经过边界检查的索引访问元素。 |
| front() | 返回第一个元素的引用。 |
| back() | 返回最后一个元素的引用。 |
| assign() | 用新元素替换原有内容。 |
| push\_back() | 在序列的尾部添加一个元素。 |
| push\_front() | 在序列的头部添加一个元素。 |
| pop\_back() | 移除容器尾部的元素。 |
| pop\_front() | 移除容器头部的元素。 |
| insert() | 在指定的位置插入一个或多个元素。 |
| erase() | 移除一个元素或一段元素。 |
| clear() | 移出所有的元素，容器大小变为 0。 |
| swap() | 交换两个容器的所有元素。 |
| emplace() | 在指定的位置直接生成一个元素。 |
| emplace\_front() | 在容器头部生成一个元素。和 push\_front() 的区别是，该函数直接在容器头部构造元素，省去了复制移动元素的过程。 |
| emplace\_back() | 在容器尾部生成一个元素。和 push\_back() 的区别是，该函数直接在容器尾部构造元素，省去了复制移动元素的过程。 |

和 vector 相比，额外增加了实现在容器头部添加和删除元素的成员函数，同时删除了 capacity()、reserve() 和 data() 成员函数。

和 array、vector 相同，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 标准库新增的 begin() 和 end() 这 2 个全局函数也适用于 deque 容器。这 2 个函数的操作对象既可以是容器，也可以是普通数组。当操作对象是容器时，它和容器包含的 begin() 和 end() 成员函数的功能完全相同；如果操作对象是普通数组，则 begin() 函数返回的是指向数组第一个元素的[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)，同样 end() 返回指向数组中最后一个元素之后一个位置的指针（注意不是最后一个元素）。  
 deque 容器还有一个std::swap(x , y) 非成员函数（其中 x 和 y 是存储相同类型元素的 deque 容器），它和 swap() 成员函数的功能完全相同，仅使用语法上有差异。  
  
如下代码演示了表 1 中部分成员函数的用法：

#include <iostream>

#include <deque>

using namespace *std*;

int main()

{

//初始化一个空deque容量

*deque*<int>d;

//向d容器中的尾部依次添加 1，2,3

d.*push\_back*(1); //{1}

d.*push\_back*(2); //{1,2}

d.*push\_back*(3); //{1,2,3}

//向d容器的头部添加 0

d.*push\_front*(0); //{0,1,2,3}

//调用 size() 成员函数输出该容器存储的字符个数。

*printf*("元素个数为：%d\n", d.*size*());

//使用迭代器遍历容器

for (auto i = d.*begin*(); i < d.*end*(); i++) {

*cout* << \*i << " ";

}

*cout* << *endl*;

return 0;

}

**运行结果为：**

**元素个数为：4  
0 1 2 3**

**表 1 中这些成员函数的具体用法，后续学习用到时会具体讲解，感兴趣的读者，也可以通过查阅 [STL手册](http://www.cplusplus.com/reference/stl/" \t "http://c.biancheng.net/view/_blank)做详细了解。**

**第二十 C++ STL deque容器迭代器用法详解**

deque 容器迭代器的类型为随机访问迭代器，deque 模板类提供了表 1 所示这些成员函数，通过调用这些函数，可以获得表示不同含义的随机访问迭代器。

**有关迭代器及其类型的介绍，可以阅读《[C++ STL迭代器（iterator）](http://c.biancheng.net/view/6675.html" \t "http://c.biancheng.net/view/_blank)》一节，本节不再做具体介绍。**

表 1 deque 支持迭代器的成员函数

|  |  |
| --- | --- |
| **成员函数** | **功能** |
| begin() | 返回指向容器中第一个元素的正向迭代器；如果是 const 类型容器，在该函数返回的是常量正向迭代器。 |
| end() | 返回指向容器最后一个元素之后一个位置的正向迭代器；如果是 const 类型容器，在该函数返回的是常量正向迭代器。此函数通常和 begin() 搭配使用。 |
| rbegin() | 返回指向最后一个元素的反向迭代器；如果是 const 类型容器，在该函数返回的是常量反向迭代器。 |
| rend() | 返回指向第一个元素之前一个位置的反向迭代器。如果是 const 类型容器，在该函数返回的是常量反向迭代器。此函数通常和 rbegin() 搭配使用。 |
| cbegin() | 和 begin() 功能类似，只不过其返回的迭代器类型为常量正向迭代器，不能用于修改元素。 |
| cend() | 和 end() 功能相同，只不过其返回的迭代器类型为常量正向迭代器，不能用于修改元素。 |
| crbegin() | 和 rbegin() 功能相同，只不过其返回的迭代器类型为常量反向迭代器，不能用于修改元素。 |
| crend() | 和 rend() 功能相同，只不过其返回的迭代器类型为常量反向迭代器，不能用于修改元素。 |

**[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 新添加的 begin() 和 end() 全局函数也同样适用于 deque 容器。即当操作对象为 deque 容器时，其功能分别和表 1 中的 begin()、end() 成员函数相同，具体用法本节后续会做详细介绍。**

表 1 中这些成员函数的具体功能如图 2 所示。

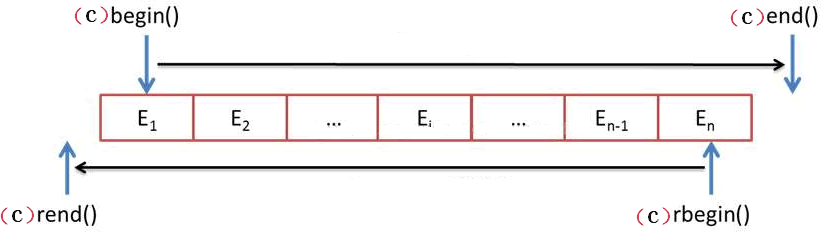


图 2 迭代器的具体功能示意图

从图 2 可以看出，这些成员函数通常是成对使用的，即 begin()/end()、rbegin()/rend()、cbegin()/cend()、crbegin()/crend() 各自成对搭配使用。其中，begin()/end() 和 cbegin/cend() 的功能是类似的，同样 rbegin()/rend() 和 crbegin()/crend() 的功能是类似的。

**值得一提的是，以上函数在实际使用时，其返回值类型都可以使用 auto 关键字代替，编译器可以自行判断出该迭代器的类型。**

**一deque容器迭代器的基本用法**

deque 容器迭代器常用来遍历容器中存储的各个元素。  
begin() 和 end() 分别用于指向「首元素」和「尾元素+1」 的位置，下面程序演示了如何使用 begin() 和 end() 遍历 deque 容器并输出其中的元素：

#include <iostream>

#include <deque>

using namespace *std*;

int main()

{

*deque*<int>d{ 1,2,3,4,5 };

//从容器首元素，遍历至最后一个元素

for (auto i = d.*begin*(); i < d.*end*(); i++) {

*cout* << \*i << " ";

}

return 0;

}

**运行结果为：**

**1 2 3 4 5**  
前面提到，[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 还提供有全局的 begin() 和 end() 函数，当操作对象为容器时，它们的功能是上面的 begin()/end() 成员函数一样。例如，将上面程序中的第 8~10 行代码可以用如下代码替换：

for (auto i = begin(d); i < end(d); i++) {

cout << \*i << " ";

}

重新编译运行程序，会发现输出结果和上面一致。  
cbegin()/cend() 成员函数和 begin()/end() 唯一不同的是，前者返回的是 const 类型的正向迭代器，这就意味着，由 cbegin() 和 cend() 成员函数返回的迭代器，可以用来遍历容器内的元素，也可以访问元素，但是不能对所存储的元素进行修改。

#include <iostream>

#include <deque>

using namespace *std*;

int main()

{

*deque*<int>d{ 1,2,3,4,5 };

auto first = d.*cbegin*();

auto end = d.*cend*();

//常量迭代器不能用来修改容器中的元素值

//\*(first + 1) = 6;//尝试修改容器中元素 2 的值

//\*(end - 1) = 10;//尝试修改容器中元素 5 的值

//常量迭代器可以用来遍历容器、访问容器中的元素

while (first < end) {

*cout* << \*first << " ";

++first;

}

return 0;

}

**运行结果：**

**1 2 3 4 5**

程序中，由于 first 和 end 都是常量迭代器，因此第 10、11 行修改容器内元素值的操作都是非法的。  
deque 模板类中还提供了 rbegin() 和 rend() 成员函数，它们分别表示指向最后一个元素和第一个元素前一个位置的随机访问迭代器，又常称为反向迭代器（如图 2 所示）。

**需要注意的是，在使用反向迭代器进行 ++ 或 -- 运算时，++ 指的是迭代器向左移动一位，-- 指的是迭代器向右移动一位，即这两个运算符的功能也“互换”了。**

反向迭代器用于以逆序的方式遍历容器中的元素。例如：

#include <iostream>

#include <deque>

using namespace *std*;

int main()

{

*deque*<int>d{ 1,2,3,4,5 };

for (auto i = d.*rbegin*(); i < d.*rend*(); i++) {

*cout* << \*i << " ";

}

return 0;

}

**运行结果为：**

**5 4 3 2 1**  
crbegin()/crend() 组合和 rbegin()/crend() 组合唯一的区别在于，前者返回的迭代器为 const 类型迭代器，不能用来修改容器中的元素，除此之外在使用上和后者完全相同。

**二deque容器迭代器的使用注意事项**

首先需要注意的一点是，迭代器的功能是遍历容器，在遍历的同时可以访问（甚至修改）容器中的元素，但迭代器不能用来初始化空的 deque 容器。  
例如，如下代码中注释部分是错误的用法：

#include <iostream>

#include <vector>

using namespace *std*;

int main()

{

*vector*<int>values;

auto first = values.*begin*();

//\*first = 1;

return 0;

}

对于空的 deque 容器来说，可以通过 push\_back()、push\_front() 或者 resize() 成员函数实现向（空）deque 容器中添加元素。

除此之外，当向 deque 容器添加元素时，deque 容器会申请更多的内存空间，同时其包含的所有元素可能会被复制或移动到新的内存地址（原来占用的内存会释放），这会导致之前创建的迭代器失效。

举个例子：

#include <iostream>

#include <deque>

using namespace *std*;

int main()

{

*deque*<int>d;

d.*push\_back*(1);

auto first = d.*begin*();

*cout* << \*first << *endl*;

//添加元素，会导致 first 失效

d.*push\_back*(1);

*cout* << \*first << *endl*;

return 0;

}

程序中第 12 行代码，会导致程序运行崩溃，其原因就在于在创建 first 迭代器之后，deque 容器做了添加元素的操作，导致 first 失效。

**在对容器做添加元素的操作之后，如果仍需要使用之前以创建好的迭代器，为了保险起见，一定要重新生成。**

**第二十一 深度剖析deque容器底层实现原理**

事实上，STL 中每个容器的特性，和它底层的实现机制密切相关，deque 自然也不例外。《[C++ STL deque容器](http://c.biancheng.net/view/6860.html" \t "http://c.biancheng.net/view/_blank)》一节中提到，deque 容器擅长在序列的头部和尾部添加或删除元素。本节将介绍 deque 容器的底层实现机制，探究其拥有此特点的原因。  
  **想搞清楚 deque 容器的实现机制，需要先了解 deque 容器的存储结构以及 deque 容器迭代器的实现原理。**

**一deque容器的存储结构**

和 vector 容器采用连续的线性空间不同，deque 容器存储数据的空间是由一段一段等长的连续空间构成，各段空间之间并不一定是连续的，可以位于在内存的不同区域。  
 为了管理这些连续空间，deque 容器用数组（数组名假设为 map）存储着各个连续空间的首地址。也就是说，map 数组中存储的都是指针，指向那些真正用来存储数据的各个连续空间（如图 1 所示）。

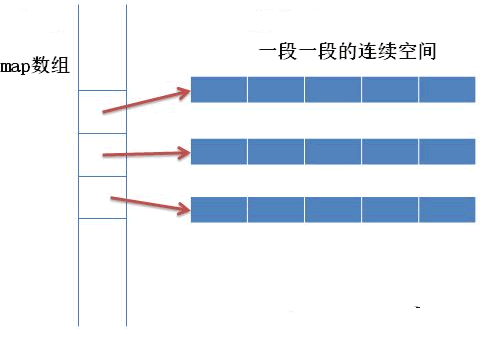


图 1 deque容器的底层存储机制

通过建立 map 数组，deque 容器申请的这些分段的连续空间就能实现“整体连续”的效果。换句话说，当 deque 容器需要在头部或尾部增加存储空间时，它会申请一段新的连续空间，同时在 map 数组的开头或结尾添加指向该空间的指针，由此该空间就串接到了 deque 容器的头部或尾部。

**有读者可能会问，如果 map 数组满了怎么办？很简单，再申请一块更大的连续空间供 map 数组使用，将原有数据（很多指针）拷贝到新的 map 数组中，然后释放旧的空间。**

deque 容器的分段存储结构，提高了在序列两端添加或删除元素的效率，但也使该容器迭代器的底层实现变得更复杂。

**二deque容器迭代器的底层实现**

由于 deque 容器底层将序列中的元素分别存储到了不同段的连续空间中，因此要想实现迭代器的功能，必须先解决如下 2 个问题：

（1）迭代器在遍历 deque 容器时，必须能够确认各个连续空间在 map 数组中的位置；

（2）迭代器在遍历某个具体的连续空间时，必须能够判断自己是否已经处于空间的边缘位置。如果是，则一旦前进或者后退，就需要跳跃到上一个或者下一个连续空间中。

为了实现遍历 deque 容器的功能，deque 迭代器定义了如下的结构：

template<class T,...>

struct \_\_deque\_iterator{

...

T\* cur;

T\* first;

T\* last;

map\_pointer node;//map\_pointer 等价于 T\*\*

}

可以看到，迭代器内部包含 4 个指针，它们各自的作用为：

cur：指向当前正在遍历的元素；

first：指向当前连续空间的首地址；

last：指向当前连续空间的末尾地址；

node：它是一个二级指针，用于指向 map 数组中存储的指向当前连续空间的指针。

借助这 4 个指针，deque 迭代器对随机访问迭代器支持的各种运算符进行了重载，能够对 deque 分段连续空间中存储的元素进行遍历。例如：

//当迭代器处于当前连续空间边缘的位置时，如果继续遍历，就需要跳跃到其它的连续空间中，该函数可用来实现此功能

void set\_node(map\_pointer new\_node) {

*node* = new\_node;//记录新的连续空间在 map 数组中的位置

*first* = \*new\_node; //更新 first 指针

//更新 last 指针，difference\_type(buffer\_size())表示每段连续空间的长度

last = *first* + *difference\_type*(buffer\_size());

}

//重载 \* 运算符

*reference* operator\*() const { return \**cur*; }

*pointer* operator->() const { return &(operator \*()); }

//重载前置 ++ 运算符

self& operator++() {

++*cur*;

//处理 cur 处于连续空间边缘的特殊情况

if (*cur* == last) {

//调用该函数，将迭代器跳跃到下一个连续空间中

set\_node(*node* + 1);

//对 cur 重新赋值

*cur* = *first*;

}

return \*this;

}

//重置前置 -- 运算符

self& operator--() {

//如果 cur 位于连续空间边缘，则先将迭代器跳跃到前一个连续空间中

if (*cur* == *first*) {

set\_node(*node* - 1);

*cur* == last;

}

--*cur*;

return \*this;

}

**三deque容器的底层实现**

了解了 deque 容器底层存储序列的结构，以及 deque 容器迭代器的内部结构之后，接下来看看 deque 容器究竟是如何实现的。  
 deque 容器除了维护先前讲过的 map 数组，还需要维护 start、finish 这 2 个 deque 迭代器。以下为 deque 容器的定义：

//\_Alloc为内存分配器

template<class *\_Ty*,

class \_Alloc = *allocator*<*\_Ty*>>

class deque {

...

protected:

*iterator* start;

*iterator* finish;

map\_pointer map;

...

}

其中，start 迭代器记录着 map 数组中首个连续空间的信息，finish 迭代器记录着 map 数组中最后一个连续空间的信息。另外需要注意的是，和普通 deque 迭代器不同，start 迭代器中的 cur 指针指向的是连续空间中首个元素；而 finish 迭代器中的 cur 指针指向的是连续空间最后一个元素的下一个位置。  
  
因此，deque 容器的底层实现如图 2 所示。

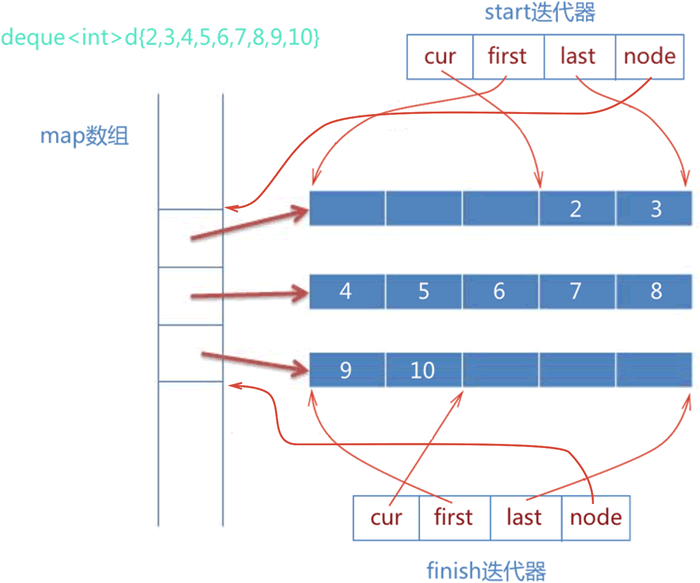


图 3 deque容器的底层实现

借助 start 和 finish，以及 deque 迭代器中重载的诸多运算符，就可以实现 deque 容器提供的大部分成员函数，比如：

//begin() 成员函数

iterator begin() {return start;}

//end() 成员函数

iterator end() { return finish;}

//front() 成员函数

reference front(){return \*start;}

//back() 成员函数

reference back(){

iterator tmp = finish;

--tmp;

return \*tmp;

}

//size() 成员函数

size\_type size() const{return finish - start;}//deque迭代器重载了 - 运算符

//enpty() 成员函数

bool empty() const{return finish == start;}

**第二十二C++ STL deque容器访问元素（4种方法）**

通过《[STL deque容器](http://c.biancheng.net/view/6860.html" \t "http://c.biancheng.net/view/_blank)》一节，详细介绍了如何创建一个 deque 容器，本节继续讲解如何访问（甚至修改）deque 容器存储的元素。  
 和 array、vector 容器一样，可以采用普通数组访问存储元素的方式，访问 deque 容器中的元素，比如：

#include <iostream>

#include <deque>

using namespace *std*;

int main()

{

*deque*<int>d{ 1,2,3,4 };

*cout* << d[1] << *endl*;

//修改指定下标位置处的元素

d[1] = 5;

*cout* << d[1] << *endl*;

return 0;

}

**运行结果为：**

**2  
5**

可以看到，容器名[n]的这种方式，不仅可以访问容器中的元素，还可以对其进行修改。但需要注意的是，使用此方法需确保下标 n 的值不会超过容器中存储元素的个数，否则会发生越界访问的错误。  
 如果想有效地避免越界访问，可以使用 deque 模板类提供的 at() 成员函数，由于该函数会返回容器中指定位置处元素的引用形式，因此利用该函数的返回值，既可以访问指定位置处的元素，如果需要还可以对其进行修改。  
不仅如此，at() 成员函数会自行判定访问位置是否越界，如果越界则抛出std::out\_of\_range异常。例如：

#include <iostream>

#include <deque>

using namespace *std*;

int main()

{

*deque*<int>d{ 1,2,3,4 };

*cout* << d.*at*(1) << *endl*;

d.*at*(1) = 5;

*cout* << d.*at*(1) << *endl*;

//下面这条语句会抛出 out\_of\_range 异常

//cout << d.at(10) << endl;

return 0;

}

**运行结果为：**

**2  
5**

**读者可能有这样一个疑问，即为什么 deque 容器在重载 [] 运算符时，没有实现边界检查的功能呢？答案很简单，因为性能。如果每次访问元素，都去检查索引值，无疑会产生很多开销。当不存在越界访问的可能时，就能避免这种开销。**  
 除此之外，deque 容器还提供了 2 个成员函数，即 front() 和 back()，它们分别返回 vector 容器中第一个和最后一个元素的引用，通过利用它们的返回值，可以访问（甚至修改）容器中的首尾元素。

#include <iostream>

#include <deque>

using namespace *std*;

int main()

{

*deque*<int> d{ 1,2,3,4,5 };

*cout* << "deque 首元素为：" << d.*front*() << *endl*;

*cout* << "deque 尾元素为：" << d.*back*() << *endl*;

//修改首元素

d.*front*() = 10;

*cout* << "deque 新的首元素为：" << d.*front*() << *endl*;

//修改尾元素

d.*back*() = 20;

*cout* << "deque 新的尾元素为：" << d.*back*() << *endl*;

return 0;

}

**运行结果为：**

**deque 首元素为：1  
deque 尾元素为：5  
deque 新的首元素为：10  
deque 新的尾元素为：20**  
 注意，和 vector 容器不同，deque 容器没有提供 data() 成员函数，同时 deque 容器在存储元素时，也无法保证其会将元素存储在连续的内存空间中，因此尝试使用[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)去访问 deque 容器中指定位置处的元素，是非常危险的。  
 另外，结合 deque 模板类中和迭代器相关的成员函数，可以实现遍历 deque 容器中指定区域元素的方法。例如：

#include <iostream>

#include <deque>

using namespace *std*;

int main()

{

*deque*<int> d{ 1,2,3,4,5 };

//从元素 2 开始遍历

auto first = d.*begin*() + 1;

//遍历至 5 结束（不包括 5）

auto end = d.*end*() - 1;

while (first < end) {

*cout* << \*first << " ";

++first;

}

return 0;

}

**运行结果为：**

**2 3 4**

**当然，deque 模板类中和迭代器相关的成员函数，还有很多，大家可以阅读《[STL deque容器迭代器](http://c.biancheng.net/view/6866.html" \t "http://c.biancheng.net/view/_blank)》做详细了解。**

**第二十三 C++ STL deque容器添加和删除元素方法完全攻略**

deque 容器中，无论是添加元素还是删除元素，都只能借助 deque 模板类提供的成员函数。表 1 中罗列的是所有和添加或删除容器内元素相关的 deque 模板类中的成员函数。

表 1 和添加或删除deque容器中元素相关的成员函数

|  |  |
| --- | --- |
| **成员函数** | **功能** |
| push\_back() | 在容器现有元素的尾部添加一个元素，和 emplace\_back() 不同，该函数添加新元素的过程是，先构造元素，然后再将该元素移动或复制到容器的尾部。 |
| pop\_back() | 移除容器尾部的一个元素。 |
| push\_front() | 在容器现有元素的头部添加一个元素，和 emplace\_back() 不同，该函数添加新元素的过程是，先构造元素，然后再将该元素移动或复制到容器的头部。 |
| pop\_front() | 移除容器尾部的一个元素。 |
| emplace\_back() | [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 新添加的成员函数，其功能是在容器尾部生成一个元素。和 push\_back() 不同，该函数直接在容器头部构造元素，省去了复制或移动元素的过程。 |
| emplace\_front() | C++ 11 新添加的成员函数，其功能是在容器头部生成一个元素。和 push\_front() 不同，该函数直接在容器头部构造元素，省去了复制或移动元素的过程。 |
| insert() | 在指定的位置直接生成一个元素。和 emplace() 不同的是，该函数添加新元素的过程是，先构造元素，然后再将该元素移动或复制到容器的指定位置。 |
| emplace() | C++ 11 新添加的成员函数，其功能是 insert() 相同，即在指定的位置直接生成一个元素。和 insert() 不同的是，emplace() 直接在容器指定位置构造元素，省去了复制或移动元素的过程。 |
| erase() | 移除一个元素或某一区域内的多个元素。 |
| clear() | 删除容器中所有的元素。 |

**在实际应用中，常用 emplace()、emplace\_front() 和 emplace\_back() 分别代替 insert()、push\_front() 和 push\_back()，具体原因本节后续会讲。**

以上这些成员函数中，除了 insert() 函数的语法格式比较多，其他函数都只有一种用法（erase() 有 2 种语法格式），下面这段程序演示了它们的具体用法：

#include <deque>

#include <iostream>

using namespace *std*;

int main()

{

*deque*<int>d;

//调用push\_back()向容器尾部添加数据。

d.*push\_back*(2); //{2}

//调用pop\_back()移除容器尾部的一个数据。

d.*pop\_back*(); //{}

//调用push\_front()向容器头部添加数据。

d.*push\_front*(2);//{2}

//调用pop\_front()移除容器头部的一个数据。

d.*pop\_front*();//{}

//调用 emplace 系列函数，向容器中直接生成数据。

d.*emplace\_back*(2); //{2}

d.*emplace\_front*(3); //{3,2}

//emplace() 需要 2 个参数，第一个为指定插入位置的迭代器，第二个是插入的值。

d.*emplace*(d.*begin*() + 1, 4);//{3,4,2}

for (auto i : d) {

*cout* << i << " ";

}

//erase()可以接受一个迭代器表示要删除元素所在位置

//也可以接受 2 个迭代器，表示要删除元素所在的区域。

d.*erase*(d.*begin*());//{4,2}

d.*erase*(d.*begin*(), d.*end*());//{}，等同于 d.clear()

return 0;

}

**运行结果为：**

**3 4 2**  
这里重点讲一下 insert() 函数的用法。insert() 函数的功能是在 deque 容器的指定位置插入一个或多个元素。该函数的语法格式有多种，如表 2 所示。

表 2 insert() 成员函数语法格式

|  |  |
| --- | --- |
| **语法格式** | **功能** |
| iterator insert(pos,elem) | 在迭代器 pos 指定的位置之前插入一个新元素elem，并返回表示新插入元素位置的迭代器。 |
| iterator insert(pos,n,elem) | 在迭代器 pos 指定的位置之前插入 n 个元素 elem，并返回表示第一个新插入元素位置的迭代器。 |
| iterator insert(pos,first,last) | 在迭代器 pos 指定的位置之前，插入其他容器（不仅限于vector）中位于 [first,last) 区域的所有元素，并返回表示第一个新插入元素位置的迭代器。 |
| iterator insert(pos,initlist) | 在迭代器 pos 指定的位置之前，插入初始化列表（用大括号{}括起来的多个元素，中间有逗号隔开）中所有的元素，并返回表示第一个新插入元素位置的迭代器。 |

下面的程序演示了 insert() 函数的这几种用法：

#include <iostream>

#include <deque>

#include <array>

using namespace *std*;

int main()

{

*std*::*deque*<int> d{ 1,2 };

//第一种格式用法

d.*insert*(d.*begin*() + 1, 3);//{1,3,2}

//第二种格式用法

d.*insert*(d.*end*(), 2, 5);//{1,3,2,5,5}

//第三种格式用法

*std*::array<int, 3>test{ 7,8,9 };

d.*insert*(d.*end*(), test.*begin*(), test.*end*());//{1,3,2,5,5,7,8,9}

//第四种格式用法

d.*insert*(d.*end*(), { 10,11 });//{1,3,2,5,5,7,8,9,10,11}

for (int i = 0; i < d.*size*(); i++) {

*cout* << d[i] << " ";

}

return 0;

}

**运行结果为：**

**1,3,2,5,5,7,8,9,10,11**

**emplace系列函数的优势**

有关 emplace()、emplace\_front() 和 emplace\_back() 分别和 insert()、push\_front() 和 push\_back() 在运行效率上的对比，可以通过下面的程序体现出来：

#include <deque>

#include <iostream>

using namespace *std*;

class testDemo

{

public:

testDemo(int num) :num(num) {

*std*::*cout* << "调用构造函数" << *endl*;

}

testDemo(const testDemo& other) :num(other.num) {

*std*::*cout* << "调用拷贝构造函数" << *endl*;

}

testDemo(testDemo&& other) :num(other.num) {

*std*::*cout* << "调用移动构造函数" << *endl*;

}

testDemo& operator=(const testDemo& other);

private:

int num;

};

testDemo& testDemo::operator=(const testDemo& other) {

this->num = other.num;

return \*this;

}

int main()

{

//emplace和insert

*cout* << "emplace:" << *endl*;

*std*::*deque*<testDemo> demo1;

demo1.*emplace*(demo1.*begin*(), 2);

*cout* << "insert:" << *endl*;

*std*::*deque*<testDemo> demo2;

demo2.*insert*(demo2.*begin*(), 2);

//emplace\_front和push\_front

*cout* << "emplace\_front:" << *endl*;

*std*::*deque*<testDemo> demo3;

demo3.*emplace\_front*(2);

*cout* << "push\_front:" << *endl*;

*std*::*deque*<testDemo> demo4;

demo4.*push\_front*(2);

//emplace\_back()和push\_back()

*cout* << "emplace\_back:" << *endl*;

*std*::*deque*<testDemo> demo5;

demo5.*emplace\_back*(2);

*cout* << "push\_back:" << *endl*;

*std*::*deque*<testDemo> demo6;

demo6.*push\_back*(2);

return 0;

}

**运行结果为：**

**emplace:  
调用构造函数  
insert:  
调用构造函数  
调用移动构造函数  
emplace\_front:  
调用构造函数  
push\_front:  
调用构造函数  
调用移动构造函数  
emplace\_back:  
调用构造函数  
push\_back:  
调用构造函数  
调用移动构造函数**

可以看到，相比和它同功能的函数，emplace 系列函数都只调用了构造函数，而没有调用移动构造函数，这无疑提高了代码的运行效率。

**第二十四C++ list（STL list）容器完全攻略（超级详细）**

[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) list 容器，又称双向链表容器，即该容器的底层是以双向链表的形式实现的。这意味着，list 容器中的元素可以分散存储在内存空间里，而不是必须存储在一整块连续的内存空间中。  
图 1 展示了 list 双向链表容器是如何存储元素的。

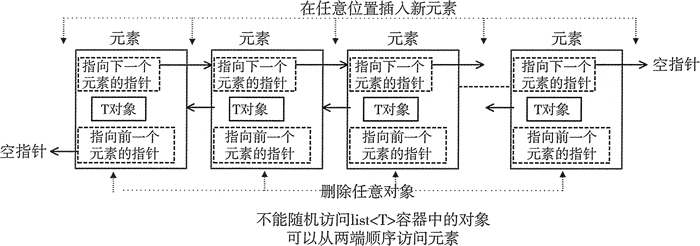


图 1 list 双向链表容器的存储结构示意图

可以看到，list 容器中各个元素的前后顺序是靠[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)来维系的，每个元素都配备了 2 个指针，分别指向它的前一个元素和后一个元素。其中第一个元素的前向指针总为 null，因为它前面没有元素；同样，尾部元素的后向指针也总为 null。  
 基于这样的存储结构，list 容器具有一些其它容器（array、vector 和 deque）所不具备的优势，即它可以在序列已知的任何位置快速插入或删除元素（时间复杂度为O(1)）。并且在 list 容器中移动元素，也比其它容器的效率高。  
 使用 list 容器的缺点是，它不能像 array 和 vector 那样，通过位置直接访问元素。举个例子，如果要访问 list 容器中的第 6 个元素，它不支持容器对象名[6]这种语法格式，正确的做法是从容器中第一个元素或最后一个元素开始遍历容器，直到找到该位置。

实际场景中，如何需要对序列进行大量添加或删除元素的操作，而直接访问元素的需求却很少，这种情况建议使用 list 容器存储序列。

list 容器以模板类 list<T>（T 为存储元素的类型）的形式在<list>头文件中，并位于 std 命名空间中。因此，在使用该容器之前，代码中需要包含下面两行代码：

#include <list>

u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;

**注意，std 命名空间也可以在使用 list 容器时额外注明，两种方式都可以。**

**一 list容器的创建**

根据不同的使用场景，有以下 5 种创建 list 容器的方式供选择。  
1) 创建一个没有任何元素的空 list 容器：

std::list<int> values;

和空 array 容器不同，空的 list 容器在创建之后仍可以添加元素，因此创建 list 容器的方式很常用。  
  
2) 创建一个包含 n 个元素的 list 容器：

std::list<int> values(10);

通过此方式创建 values 容器，其中包含 10 个元素，每个元素的值都为相应类型的默认值（int类型的默认值为 0）。  
  
3) 创建一个包含 n 个元素的 list 容器，并为每个元素指定初始值。例如：

std::list<int> values(10, 5);

如此就创建了一个包含 10 个元素并且值都为 5 个 values 容器。  
  
4) 在已有 list 容器的情况下，通过拷贝该容器可以创建新的 list 容器。例如：

std::list<int> value1(10);

std::list<int> value2(value1);

注意，采用此方式，必须保证新旧容器存储的元素类型一致。  
  
5) 通过拷贝其他类型容器（或者普通数组）中指定区域内的元素，可以创建新的 list 容器。例如：

[纯文本复制](http://c.biancheng.net/view/6892.html)

//拷贝普通数组，创建list容器

int a[] = { 1,2,3,4,5 };

std::list<int> values(a, a+5);

//拷贝其它类型的容器，创建 list 容器

std::array<int, 5>arr{ 11,12,13,14,15 };

std::list<int>values(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15}

**二list容器可用的成员函数**

表 2 中罗列出了 list 模板类提供的所有成员函数以及各自的功能。

表 2 list 容器可用的成员函数

|  |  |
| --- | --- |
| **成员函数** | **功能** |
| begin() | 返回指向容器中第一个元素的双向迭代器。 |
| end() | 返回指向容器中最后一个元素所在位置的下一个位置的双向迭代器。 |
| rbegin() | 返回指向最后一个元素的反向双向迭代器。 |
| rend() | 返回指向第一个元素所在位置前一个位置的反向双向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| empty() | 判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。 |
| size() | 返回当前容器实际包含的元素个数。 |
| max\_size() | 返回容器所能包含元素个数的最大值。这通常是一个很大的值，一般是 232-1，所以我们很少会用到这个函数。 |
| front() | 返回第一个元素的引用。 |
| back() | 返回最后一个元素的引用。 |
| assign() | 用新元素替换容器中原有内容。 |
| emplace\_front() | 在容器头部生成一个元素。该函数和 push\_front() 的功能相同，但效率更高。 |
| push\_front() | 在容器头部插入一个元素。 |
| pop\_front() | 删除容器头部的一个元素。 |
| emplace\_back() | 在容器尾部直接生成一个元素。该函数和 push\_back() 的功能相同，但效率更高。 |
| push\_back() | 在容器尾部插入一个元素。 |
| pop\_back() | 删除容器尾部的一个元素。 |
| emplace() | 在容器中的指定位置插入元素。该函数和 insert() 功能相同，但效率更高。 |
| insert() | 在容器中的指定位置插入元素。 |
| erase() | 删除容器中一个或某区域内的元素。 |
| swap() | 交换两个容器中的元素，必须保证这两个容器中存储的元素类型是相同的。 |
| resize() | 调整容器的大小。 |
| clear() | 删除容器存储的所有元素。 |
| splice() | 将一个 list 容器中的元素插入到另一个容器的指定位置。 |
| remove(val) | 删除容器中所有等于 val 的元素。 |
| remove\_if() | 删除容器中满足条件的元素。 |
| unique() | 删除容器中相邻的重复元素，只保留一个。 |
| merge() | 合并两个事先已排好序的 list 容器，并且合并之后的 list 容器依然是有序的。 |
| sort() | 通过更改容器中元素的位置，将它们进行排序。 |
| reverse() | 反转容器中元素的顺序。 |

除此之外，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 标准库还新增加了 begin() 和 end() 这 2 个函数，和 list 容器包含的 begin() 和 end() 成员函数不同，标准库提供的这 2 个函数的操作对象，既可以是容器，还可以是普通数组。当操作对象是容器时，它和容器包含的 begin() 和 end() 成员函数的功能完全相同；如果操作对象是普通数组，则 begin() 函数返回的是指向数组第一个元素的指针，同样 end() 返回指向数组中最后一个元素之后一个位置的指针（注意不是最后一个元素）。  
  **list 容器还有一个std::swap(x , y)非成员函数（其中 x 和 y 是存储相同类型元素的 list 容器），它和 swap() 成员函数的功能完全相同，仅使用语法上有差异。**  
如下代码演示了表 2 中部分成员函数的用法：

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

//创建空的 list 容器

*std*::*list*<double> values;

//向容器中添加元素

values.*push\_back*(3.1);

values.*push\_back*(2.2);

values.*push\_back*(2.9);

*cout* << "values size：" << values.*size*() << *endl*;

//对容器中的元素进行排序

values.*sort*();

//使用迭代器输出list容器中的元素

for (*std*::*list*<double>::*iterator* it = values.*begin*(); it != values.*end*(); ++it) {

*std*::*cout* << \*it << " ";

}

return 0;

}

**运行结果为：**

**values size：3  
2.2 2.9 3.1**

**表 2 中这些成员函数的具体用法，后续学习用到时会具体讲解，感兴趣的读者，也可以通过查阅[STL手册](http://www.cplusplus.com/reference/stl/" \t "http://c.biancheng.net/view/_blank)做详细了解**。

**第二十五C++ STL list迭代器及用法**

只有运用迭代器，才能访问 list 容器中存储的各个元素。list 模板类提供了如表 1 所示的这些迭代器函数。

表 1 list 容器迭代器函数

|  |  |
| --- | --- |
| **迭代器函数** | **功能** |
| begin() | 返回指向容器中第一个元素的双向迭代器（正向迭代器）。 |
| end() | 返回指向容器中最后一个元素所在位置的下一个位置的双向迭代器。（正向迭代器）。 |
| rbegin() | 返回指向最后一个元素的反向双向迭代器。 |
| rend() | 返回指向第一个元素所在位置前一个位置的反向双向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，正向迭代器增加了 const 属性，即不能用于修改元素。 |
| cend() | 和 end() 功能相同，只不过在其基础上，正向迭代器增加了 const 属性，即不能用于修改元素。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，反向迭代器增加了 const 属性，即不能用于修改元素。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，反向迭代器增加了 const 属性，即不能用于修改元素。 |

**除此之外，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 新添加的 begin() 和 end() 全局函数也同样适用于 list 容器。即当操作对象为 list 容器时，其功能分别和表 1 中的 begin()、end() 成员函数相同**。

表 1 中各个成员函数的功能如图 2 所示。

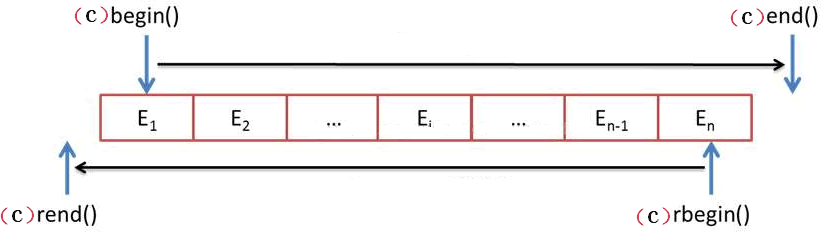


图 2 list 容器迭代器的功能示意图

**注意，list 容器的底层实现结构为双向链表，图 2 这种表示仅是为了方便理解各个迭代器函数的功能。**

从图 2 可以看出，这些成员函数通常是成对使用的，即 begin()/end()、rbegin()/rend()、cbegin()/cend()、crbegin()/crend() 各自成对搭配使用。其中，begin()/end() 和 cbegin/cend() 的功能是类似的，同样 rbegin()/rend() 和 crbegin()/crend() 的功能是类似的。  
**前面章节已经详细介绍了 array、vector、deque 容器的迭代器，和它们相比，list 容器迭代器最大的不同在于，其配备的迭代器类型为双向迭代器，而不再是随机访问迭代器。**  
  
这意味着，假设 p1 和 p2 都是双向迭代器，则它们支持使用 ++p1、 p1++、 p1--、 p1++、 \*p1、 p1==p2 以及 p1!=p2 运算符，但不支持以下操作（其中 i 为整数）：

（1）p1[i]：不能通过下标访问 list 容器中指定位置处的元素。

（2）p1-=i、 p1+=i、 p1+i 、p1-i：双向迭代器 p1 不支持使用 -=、+=、+、- 运算符。

（3）p1<p2、 p1>p2、 p1<=p2、 p1>=p2：双向迭代器 p1、p2 不支持使用 <、 >、 <=、 >= 比较运算符。

**有关迭代器类别和功能的具体介绍，可阅读 《[C++ STL迭代器](http://c.biancheng.net/view/6675.html" \t "http://c.biancheng.net/view/_blank)》一节。**

下面这个程序演示了如何使用表 1 中的迭代器遍历 list 容器中的各个元素。

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

//创建 list 容器

*std*::*list*<char> values{ 'h','t','t','p',':','/','/','c','.','b','i','a','n','c','h','e','n','g','.','n','e','t' };

//使用begin()/end()迭代器函数对输出list容器中的元素

for (*std*::*list*<char>::*iterator* it = values.*begin*(); it != values.*end*(); ++it) {

*std*::*cout* << \*it;

}

*cout* << *endl*;

//使用 rbegin()/rend()迭代器函数输出 lsit 容器中的元素

for (*std*::*list*<char>::*reverse\_iterator* it = values.*rbegin*(); it != values.*rend*(); ++it) {

*std*::*cout* << \*it;

}

return 0;

}

输出结果为：

http://c.biancheng.net  
ten.gnehcnaib.c//:ptth

**注意，程序中比较迭代器之间的关系，用的是 != 运算符，因为它不支持 < 等运算符。另外在实际场景中，所有迭代器函数的返回值都可以传给使用 auto 关键字定义的变量，因为编译器可以自行判断出该迭代器的类型。**  
 值得一提的是，list 容器在进行插入（insert()）、接合（splice()）等操作时，都不会造成原有的 list 迭代器失效，甚至进行删除操作，而只有指向被删除元素的迭代器失效，其他迭代器不受任何影响。  
举个例子：

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

//创建 list 容器

*std*::*list*<char> values{ 'h','t','t','p',':','/','/','c','.','b','i','a','n','c','h','e','n','g','.','n','e','t' };

//创建 begin 和 end 迭代器

*std*::*list*<char>::*iterator* begin = values.*begin*();

*std*::*list*<char>::*iterator* end = values.*end*();

//头部和尾部插入字符 '1'

values.*insert*(begin, '1');

values.*insert*(end, '1');

while (begin != end)

{

*std*::*cout* << \*begin;

++begin;

}

return 0;

}

**运行结果为：**

**http://c.biancheng.net1**

**可以看到，在进行插入操作之后，仍使用先前创建的迭代器遍历容器，虽然程序不会出错，但由于插入位置的不同，可能会遗漏新插入的元素。**

**第二十六C++ list容器底层存储结构**

前面在讲 STL list 容器时提到，该容器的底层是用双向链表实现的，甚至一些 STL 版本中（比如 SGI STL），list 容器的底层实现使用的是双向循环链表。

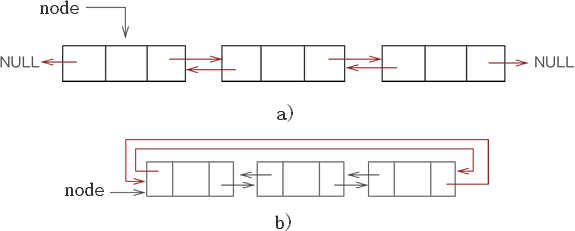


图 1 双向链表（ a) ）和双向循环链表（ b) ）

图 1 中，node 表示链表的头指针。

如图 1 所示，使用链表存储数据，并不会将它们存储到一整块连续的内存空间中。恰恰相反，各元素占用的存储空间（又称为节点）是独立的、分散的，它们之间的线性关系通过指针（图 1 以箭头表示）来维持。

list 容器节点结构

通过图 1 可以看到，双向链表的各个节点中存储的不仅仅是元素的值，还应包含 2 个指针，分别指向前一个元素和后一个元素。  
 通过查看 list 容器的源码实现，其对节点的定义如下：

template<typename T,...>

struct \_\_List\_node{

//...

\_\_list\_node<T>\* prev;

\_\_list\_node<T>\* next;

T myval;

//...

}

注意，为了方便读者理解，此代码以及本节后续代码，都省略了和本节核心内容不相关的内容，如果读者对此部分感兴趣，可查看 list 容器实现源码。

可以看到，list 容器定义的每个节点中，都包含 \*prev、\*next 和 myval。其中，prev 指针用于指向前一个节点；next 指针用于指向后一个节点；myval 用于存储当前元素的值。

**一 list容器迭代器的底层实现**

和 array、vector 这些容器迭代器的实现方式不同，由于 list 容器的元素并不是连续存储的，所以该容器迭代器中，必须包含一个可以指向 list 容器的指针，并且该指针还可以借助重载的 \*、++、--、==、!= 等运算符，实现迭代器正确的递增、递减、取值等操作。  
因此，list 容器迭代器的实现代码如下：

template<tyepname T, ...>

struct \_\_list\_iterator {

\_\_list\_node<T>\* node;

//...

//重载 == 运算符

bool operator==(const \_\_list\_iterator& x) { return node == x.node; }

//重载 != 运算符

bool operator!=(const \_\_list\_iterator& x) { return node != x.node; }

//重载 \* 运算符，返回引用类型

T\* operator \*() const { return \*(node).myval; }

//重载前置 ++ 运算符

\_\_list\_iterator<T>& operator ++() {

node = (\*node).*next*;

return \*this;

}

//重载后置 ++ 运算符

\_\_list\_iterator<T>& operator ++(int) {

\_\_list\_iterator<T> tmp = \*this;

++(\*this);

return tmp;

}

//重载前置 -- 运算符

\_\_list\_iterator<T>& operator--() {

node = (\*node).*prev*;

return \*this;

}

//重载后置 -- 运算符

\_\_list\_iterator<T> operator--(int) {

\_\_list\_iterator<T> tmp = \*this;

--(\*this);

return tmp;

}

//...

}

可以看到，迭代器的移动就是通过操作节点的指针实现的。

**二list容器的底层实现**

本节开头提到，不同版本的 STL 标准库中，list 容器的底层实现并不完全一致，但原理基本相同。这里以 SGI STL 中的 list 容器为例，讲解该容器的具体实现过程。  
 SGI STL 标准库中，list 容器的底层实现为双向循环链表，相比双向链表结构的好处是在构建 list 容器时，只需借助一个指针即可轻松表示 list 容器的首尾元素。  
如下是 SGI STL 标准库中对 list 容器的定义：

template <class T,...>

class list

{

//...

//指向链表的头节点，并不存放数据

\_\_list\_node<T>\* node;

//...以下还有list 容器的构造函数以及很多操作函数

}

另外，为了更方便的实现 list 模板类提供的函数，该模板类在构建容器时，会刻意在容器链表中添加一个空白节点，并作为 list 链表的首个节点（又称头节点）。

**使用双向链表实现的 list 容器，其内部通常包含 2 个指针，并分别指向链表中头部的空白节点和尾部的空白节点（也就是说，其包含 2 个空白节点）。**

比如，我们经常构造空的 list 容器，其用到的构造函数如下所示：

list() { empty\_initialize(); }

// 用于空链表的建立

void empty\_initialize()

{

node = get\_node();//初始化节点

node->next = node; // 前置节点指向自己

node->prev = node; // 后置节点指向自己

}

显然，即便是创建空的 list 容器，它也包含有 1 个节点。  
除此之外，list 模板类中还提供有带参的构造函数，它们的实现过程大致分为以下 2 步：

（1）调用 empty\_initialize() 函数，构造带有头节点的空 list 容器链表；

（2）将各个参数按照次序插入到空的 list 容器链表中。

由此可以总结出，list 容器实际上就是一个带有头节点的双向循环链表。如图 2 所示，此为存有 2 个元素的 list 容器：

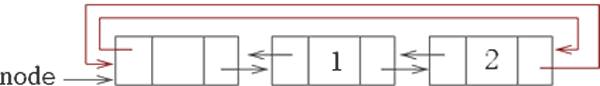


图 1 list 容器底层存储示意图

在此基础上，通过借助 node 头节点，就可以实现 list 容器中的所有成员函数，比如：

//begin()成员函数

\_\_list\_iterator<T> begin(){return (\*node).next;}

//end()成员函数

\_\_list\_iterator<T> end(){return node;}

//empty()成员函数

bool empty() const{return (\*node).next == node;}

//front()成员函数

T& front() {return \*begin();}

//back()成员函数

T& back() {return \*(--end();)}

//...

以上也只是罗列了 list 容器中一部分成员函数的实现方法，其它成员函数的具体实现，这里不再具体描述，感兴趣的读者，可下载 list 容器的实现源码。

**第二十七C++ list（STL list）访问元素的几种方法**

不同于之前学过的 [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 容器，访问 list 容器中存储元素的方式很有限，即要么使用 front() 和 back() 成员函数，要么使用 list 容器迭代器。

**list 容器不支持随机访问，未提供下标操作符 [] 和 at() 成员函数，也没有提供 data() 成员函数。**

通过 front() 和 back() 成员函数，可以分别获得 list 容器中第一个元素和最后一个元素的引用形式。举个例子：

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

*std*::*list*<int> mylist{ 1,2,3,4 };

int& first = mylist.*front*();

int& last = mylist.*back*();

*cout* << first << " " << last << *endl*;

first = 10;

last = 20;

*cout* << mylist.*front*() << " " << mylist.*back*() << *endl*;

return 0;

}

**输出结果为：**

**1 4  
10 20**

可以看到，通过 front() 和 back() 的返回值，我们不仅能分别获取当前 list 容器中的首尾元素，必要时还能修改它们的值。  
  
除此之外，如果想访问 list 容存储的其他元素，就只能使用 list 容器的迭代器。例如：

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

const *std*::*list*<int> mylist{ 1,2,3,4,5 };

auto it = mylist.*begin*();

*cout* << \*it << " ";

++it;

while (it != mylist.*end*())

{

*cout* << \*it << " ";

++it;

}

return 0;

}

**运行结果为：**

**1 2 3 4 5**

值得一提的是，对于非 const 类型的 list 容器，迭代器不仅可以访问容器中的元素，也可以对指定元素的值进行修改。

**当然，对于修改容器指定元素的值，list 模板类提供有专门的成员函数 assign()，感兴趣的读者可自行查找该成员函数的用法。**

**第二十八C++ STL list添加（插入）元素方法详解**

前面章节介绍了如何创建 list 容器，在此基础上，本节继续讲解如何向现有 list 容器中添加或插入新的元素。  
list 模板类中，与“添加或插入新元素”相关的成员方法有如下几个：

push\_front()：向 list 容器首个元素前添加新元素；

push\_back()：向 list 容器最后一个元素后添加新元素；

emplace\_front()：在容器首个元素前直接生成新的元素；

emplace\_back()：在容器最后一个元素后直接生成新的元素；

emplace()：在容器的指定位置直接生成新的元素；

insert()：在指定位置插入新元素；

splice()：将其他 list 容器存储的多个元素添加到当前 list 容器的指定位置处。

以上这些成员方法中，除了 insert() 和 splice() 方法有多种语法格式外，其它成员方法都仅有 1 种语法格式，下面程序演示了它们的具体用法。

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

*std*::*list*<int> values{ 1,2,3 };

values.*push\_front*(0);//{0,1,2,3}

values.*push\_back*(4); //{0,1,2,3,4}

values.*emplace\_front*(-1);//{-1,0,1,2,3,4}

values.*emplace\_back*(5); //{-1,0,1,2,3,4,5}

//emplace(pos,value),其中 pos 表示指明位置的迭代器，value为要插入的元素值

values.*emplace*(values.*end*(), 6);//{-1,0,1,2,3,4,5,6}

for (auto p = values.*begin*(); p != values.*end*(); ++p) {

*cout* << \*p << " ";

}

return 0;

}

**输出结果为：**

**-1,0,1,2,3,4,5,6**

**一list insert()成员方法**

insert() 成员方法的语法格式有 4 种，如表 1 所示。

表 1 insert() 成员方法语法格式

|  |  |
| --- | --- |
| **语法格式** | **用法说明** |
| iterator insert(pos,elem) | 在迭代器 pos 指定的位置之前插入一个新元素 elem，并返回表示新插入元素位置的迭代器。 |
| iterator insert(pos,n,elem) | 在迭代器 pos 指定的位置之前插入 n 个元素 elem，并返回表示第一个新插入元素位置的迭代器。 |
| iterator insert(pos,first,last) | 在迭代器 pos 指定的位置之前，插入其他容器（例如 array、vector、deque 等）中位于 [first,last) 区域的所有元素，并返回表示第一个新插入元素位置的迭代器。 |
| iterator insert(pos,initlist) | 在迭代器 pos 指定的位置之前，插入初始化列表（用大括号 { } 括起来的多个元素，中间有逗号隔开）中所有的元素，并返回表示第一个新插入元素位置的迭代器。 |

下面的程序演示了如何使用 insert() 方法向 list 容器中插入元素。

#include <iostream>

#include <list>

#include <array>

using namespace *std*;

int main()

{

*std*::*list*<int> values{ 1,2 };

//第一种格式用法

values.*insert*(values.*begin*(), 3);//{3,1,2}

//第二种格式用法

values.*insert*(values.*end*(), 2, 5);//{3,1,2,5,5}

//第三种格式用法

*std*::array<int, 3>test{ 7,8,9 };

values.*insert*(values.*end*(), test.*begin*(), test.*end*());//{3,1,2,5,5,7,8,9}

//第四种格式用法

values.*insert*(values.*end*(), { 10,11 });//{3,1,2,5,5,7,8,9,10,11}

for (auto p = values.*begin*(); p != values.*end*(); ++p)

{

*cout* << \*p << " ";

}

return 0;

}

**输出结果为：**

**3 1 2 5 5 7 8 9 10 11**

学到这里，读者有没有发现，同样是实现插入元素的功能，无论是 push\_front()、push\_back() 还是 insert()，都有以 emplace 为名且功能和前者相同的成员函数。这是因为，后者是 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 标准新添加的，在大多数场景中，都可以完全替代前者实现同样的功能。更重要的是，实现同样的功能，emplace 系列方法的执行效率更高。

**有关 list 模板类中 emplace 系列函数执行效率更高的原因，前面在讲解 deque 容器模板类中的 emplace 系列函数时已经讲过，读者可阅读《[C++ STL deque容器添加和删除元素](http://c.biancheng.net/view/6877.html" \t "http://c.biancheng.net/view/_blank)》一节做详细了解**。

**二list splice()成员方法**

和 insert() 成员方法相比，splice() 成员方法的作用对象是其它 list 容器，其功能是将其它 list 容器中的元素添加到当前 list 容器中指定位置处。  
splice() 成员方法的语法格式有 3 种，如表 2 所示。

表 2 splice() 成员方法的用法

|  |  |
| --- | --- |
| **语法格式** | **功能** |
| void splice (iterator position, list& x); | position 为迭代器，用于指明插入位置；x 为另一个 list 容器。 此格式的 splice() 方法的功能是，将 x 容器中存储的所有元素全部移动当前 list 容器中 position 指明的位置处。 |
| void splice (iterator position, list& x, iterator i); | position 为迭代器，用于指明插入位置；x 为另一个 list 容器；i 也是一个迭代器，用于指向 x 容器中某个元素。 此格式的 splice() 方法的功能是将 x 容器中 i 指向的元素移动到当前容器中 position 指明的位置处。 |
| void splice (iterator position, list& x, iterator first, iterator last); | position 为迭代器，用于指明插入位置；x 为另一个 list 容器；first 和 last 都是迭代器，[fist,last) 用于指定 x 容器中的某个区域。 此格式的 splice() 方法的功能是将 x 容器 [first, last) 范围内所有的元素移动到当前容器 position 指明的位置处。 |

我们知道，list 容器底层使用的是链表存储结构，splice() 成员方法移动元素的方式是，将存储该元素的节点从 list 容器底层的链表中摘除，然后再链接到当前 list 容器底层的链表中。这意味着，当使用 splice() 成员方法将 x 容器中的元素添加到当前容器的同时，该元素会从 x 容器中删除。  
  
下面程序演示了 splice() 成员方法的用法：

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

//创建并初始化 2 个 list 容器

*list*<int> mylist1{ 1,2,3,4 }, mylist2{ 10,20,30 };

*list*<int>::*iterator* it = ++mylist1.*begin*(); //指向 mylist1 容器中的元素 2

//调用第一种语法格式

mylist1.*splice*(it, mylist2); // mylist1: 1 10 20 30 2 3 4

// mylist2:

// it 迭代器仍然指向元素 2，只不过容器变为了 mylist1

//调用第二种语法格式，将 it 指向的元素 2 移动到 mylist2.begin() 位置处

mylist2.*splice*(mylist2.*begin*(), mylist1, it); // mylist1: 1 10 20 30 3 4

// mylist2: 2

// it 仍然指向元素 2

//调用第三种语法格式，将 [mylist1.begin(),mylist1.end())范围内的元素移动到 mylist.begin() 位置处

mylist2.*splice*(mylist2.*begin*(), mylist1, mylist1.*begin*(), mylist1.*end*());//mylist1:

//mylist2:1 10 20 30 3 4 2

*cout* << "mylist1 包含 " << mylist1.*size*() << "个元素" << *endl*;

*cout* << "mylist2 包含 " << mylist2.*size*() << "个元素" << *endl*;

//输出 mylist2 容器中存储的数据

*cout* << "mylist2:";

for (auto iter = mylist2.*begin*(); iter != mylist2.*end*(); ++iter) {

*cout* << \*iter << " ";

}

return 0;

}

**程序执行结果为：**

**mylist1 包含 0个元素  
mylist2 包含 7个元素  
mylist2:1 10 20 30 3 4 2**

**第二十九empty()和size()都可以判断容器是否为空，谁更好？**

到目前为止，我们已经了解了 C++ STL 标准库中 vector、deque 和 list 这 3 个容器的功能和具体用法。学习过程中，读者是否想过一个问题，即这些容器的模板类中都提供了 empty() 成员方法和 size() 成员方法，它们都可以用来判断容器是否为空。  
  
换句话说，假设有一个容器 cont，则此代码：

if(cont.size() == 0)

本质上和如下代码是等价的：

if(cont.empty())

那么，在实际场景中，到底应该使用哪一种呢？  
  
建议使用 empty() 成员方法。理由很简单，无论是哪种容器，只要其模板类中提供了 empty() 成员方法，使用此方法都可以保证在 O(1) 时间复杂度内完成对“容器是否为空”的判断；但对于 list 容器来说，使用 size() 成员方法判断“容器是否为空”，可能要消耗 O(n) 的时间复杂度。

**注意，这个结论不仅适用于 vector、deque 和 list 容器，后续还会讲解更多容器的用法，该结论也依然适用。**

那么，为什么 list 容器这么特殊呢？这和 list 模板类提供了独有的 splice() 成员方法有关。

**一深度剖析选用empty()的原因**

做一个大胆的设想，假设我们自己就是 list 容器的设计者。从始至终，我们的目标都是让 list 成为标准容器，并被广泛使用，因此打造“高效率的 list”成为我们奋斗的目标。  
 在实现 list 容器的过程中我们发现，用户经常需要知道当前 list 容器中存有多少个元素，于是我们想让 size() 成员方法的执行效率达到 O(1)。为了实现这个目的，我们必须重新设计 list，使它总能以最快的效率知道自己含有多少个元素。

要想令 size() 的执行效率达到 O(1)，最直接的实现方式是：在 list 模板类中设置一个专门用于统计存储元素数量的 size 变量，其位于所有成员方法的外部。与此同时，在每一个可用来为容器添加或删除元素的成员方法中，添加对 size 变量值的更新操作。由此，size() 成员方法只需返回 size 这个变量即可（时间复杂度为 O(1)）。

不仅如此，由于 list 容器底层采用的是链式存储结构（也就是链表），该结构最大的特点就是，某一链表中存有元素的节点，无需经过拷贝就可以直接链接到其它链表中，且整个过程只需要消耗 O(1) 的时间复杂度。考虑到很多用户之所以选用 list 容器，就是看中了其底层存储结构的这一特性。因此，作为 list 容器设计者的我们，自然也想将 splice() 方法的时间复杂度设计为 O(1)。  
  
这里就产生了一个矛盾，即如果将 size() 设计为 O(1) 时间复杂度，则由于 splice() 成员方法会修改 list 容器存储元素的个数，因此该方法中就需要添加更新 size 变量的代码（更新方式无疑是通过遍历链表来实现），这也就意味着 splice() 成员方法的执行效率将无法达到 O(1)；反之，如果将 splice() 成员方法的执行效率提高到 O(1)，则 size() 成员方法将无法实现 O(1) 的时间复杂度。

也就是说，list 容器中的 size() 和 splice() 总有一个要做出让步，即只能实现其中一个方法的执行效率达到 O(1)。

值得一提的是，不同版本的 STL 标准库，其底层解决此冲突的抉择是不同的。以本教程所用的 C++ STL 标准模板库为例，其选择将 splice() 成员方法的执行效率达到 O(1)，而 size() 成员方法的执行效率为 O(n)。当然，有些版本的 STL 标准库中，选择将 size() 方法的执行效率设计为 O(1)。  
 但不论怎样，选用 empty() 判断容器是否为空，效率总是最高的。所以，如果程序中需要判断当前容器是否为空，应优先考虑使用 empty()。

**第三十C++ STL list删除元素详解**

对 list 容器存储的元素执行删除操作，需要借助该容器模板类提供的成员函数。幸运的是，相比其它 [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 容器模板类，list 模板类提供了更多用来实现此操作的成员函数（如表 1 所示）。

表 1 实现 list 容器删除元素的成员函数

|  |  |
| --- | --- |
| **成员函数** | **功能** |
| pop\_front() | 删除位于 list 容器头部的一个元素。 |
| pop\_back() | 删除位于 list 容器尾部的一个元素。 |
| erase() | 该成员函数既可以删除 list 容器中指定位置处的元素，也可以删除容器中某个区域内的多个元素。 |
| clear() | 删除 list 容器存储的所有元素。 |
| remove(val) | 删除容器中所有等于 val 的元素。 |
| unique() | 删除容器中相邻的重复元素，只保留一份。 |
| remove\_if() | 删除容器中满足条件的元素。 |

其中，pop\_front()、pop\_back() 和 clear() 的用法非常简单，这里仅给出一个样例，不再过多解释：

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

*list*<int>values{ 1,2,3,4 };

//删除当前容器中首个元素

values.*pop\_front*();//{2,3,4}

//删除当前容器最后一个元素

values.*pop\_back*();//{2,3}

//清空容器，删除容器中所有的元素

values.*clear*(); //{}

for (auto begin = values.*begin*(); begin != values.*end*(); ++begin)

{

*cout* << \*begin << " ";

}

return 0;

}

**运行程序，可以看到输出结果为“空”。**  
  
erase() 成员函数有以下 2 种语法格式：

iterator erase (iterator position);  
iterator erase (iterator first, iterator last);

利用第一种语法格式，可实现删除 list 容器中 position 迭代器所指位置处的元素，例如：

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

*list*<int>values{ 1,2,3,4,5 };

//指向元素 1 的迭代器

auto del = values.*begin*();

//迭代器右移，改为指向元素 2

++del;

values.*erase*(del); //{1,3,4,5}

for (auto begin = values.*begin*(); begin != values.*end*(); ++begin)

{

*cout* << \*begin << " ";

}

return 0;

}

**运行结果为：**

**1 3 4 5**

利用第二种语法格式，可实现删除 list 容器中 first 迭代器和 last 迭代器限定区域内的所有元素（包括 first 指向的元素，但不包括 last 指向的元素）。例如：

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

*list*<int>values{ 1,2,3,4,5 };

//指定删除区域的左边界

auto first = values.*begin*();

++first;//指向元素 2

//指向删除区域的右边界

auto last = values.*end*();

--last;//指向元素 5

//删除 2、3 和 4

values.*erase*(first, last);

for (auto begin = values.*begin*(); begin != values.*end*(); ++begin)

{

*cout* << \*begin << " ";

}

return 0;

}

**运行结果为：**

**1 5**  
 erase() 成员函数是按照被删除元素所在的位置来执行删除操作，如果想根据元素的值来执行删除操作，可以使用 remove() 成员函数。例如：

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

*list*<char>values{ 'a','b','c','d' };

values.*remove*('c');

for (auto begin = values.*begin*(); begin != values.*end*(); ++begin)

{

*cout* << \*begin << " ";

}

return 0;

}

**运行结果为：**

**a b d**  
unique() 函数也有以下 2 种语法格式：

void unique()  
void unique（BinaryPredicate）//传入一个二元谓词函数  
以上 2 种格式都能实现去除 list 容器中相邻重复的元素，仅保留一份。但第 2 种格式的优势在于，我们能自定义去重的规则，例如：

#include <iostream>

#include <list>

using namespace *std*;

//二元谓词函数

bool demo(double first, double second)

{

return (int(first) == int(second));

}

int main()

{

*list*<double> mylist{ 1,1.2,1.2,3,4,4.5,4.6 };

//删除相邻重复的元素，仅保留一份

mylist.*unique*();//{1, 1.2, 3, 4, 4.5, 4.6}

for (auto it = mylist.*begin*(); it != mylist.*end*(); ++it)

*cout* << \*it << ' ';

*cout* << *endl*;

//demo 为二元谓词函数，是我们自定义的去重规则

mylist.*unique*(demo);

for (auto it = mylist.*begin*(); it != mylist.*end*(); ++it)

*std*::*cout* << \*it << ' ';

return 0;

}

**运行结果为：**

**1 1.2 3 4 4.5 4.6  
1 3 4**

**注意，除了以上一定谓词函数的方式，还可以使用 [lamba表达式](http://c.biancheng.net/view/433.html" \t "http://c.biancheng.net/view/_blank)以及[函数对象](http://c.biancheng.net/view/354.html" \t "http://c.biancheng.net/view/_blank)的方式定义。**

可以看到，通过调用无参的 unique()，仅能删除相邻重复（也就是相等）的元素，而通过我们自定义去重的规则，可以更好的满足在不同场景下去重的需求。  
 除此之外，通过将自定义的谓词函数（不限定参数个数）传给 remove\_if() 成员函数，list 容器中能使谓词函数成立的元素都会被删除。举个例子：

#include <iostream>

#include <list>

using namespace *std*;

int main()

{

*std*::*list*<int> mylist{ 15, 36, 7, 17, 20, 39, 4, 1 };

//删除 mylist 容器中能够使 lamba 表达式成立的所有元素。

mylist.*remove\_if*([](int value) {return (value < 10); }); //{15 36 17 20 39}

for (auto it = mylist.*begin*(); it != mylist.*end*(); ++it)

*std*::*cout* << ' ' << \*it;

return 0;

}

**运行结果为：**

**15 36 17 20 39**

**第三十一 C++ STL forward\_list容器完全攻略**

forward\_list 是 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 新添加的一类容器，其底层实现和 list 容器一样，采用的也是链表结构，只不过 forward\_list 使用的是单链表，而 list 使用的是双向链表（如图 1 所示）。

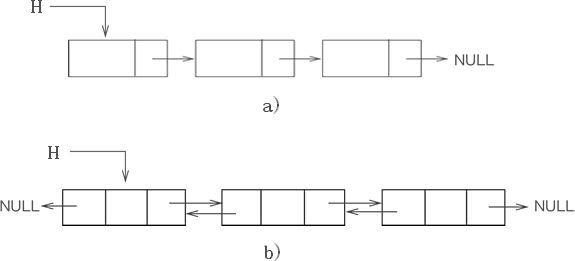


图 1 单链表（ a) ）和双向链表（ b) ）

图 1 中，H 表示链表的表头。

通过图 1 不难看出，使用链表存储数据最大的特点在于，其并不会将数据进行集中存储（向数组那样），换句话说，链表中数据的存储位置是分散的、随机的，整个链表中数据的线性关系通过[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)来维持。  
 因此，forward\_list 容器具有和 list 容器相同的特性，即擅长在序列的任何位置进行插入元素或删除元素的操作，但对于访问存储的元素，没有其它容器（如 array、vector）的效率高。  
 另外，由于单链表没有双向链表那样灵活，因此相比 list 容器，forward\_list 容器的功能受到了很多限制。比如，由于单链表只能从前向后遍历，而不支持反向遍历，因此 forward\_list 容器只提供前向迭代器，而不是双向迭代器。这意味着，forward\_list 容器不具有 rbegin()、rend() 之类的成员函数。

**有关迭代器的具体分类以及各种迭代器的具体功能，可以阅读《[C++ STL迭代器](http://c.biancheng.net/view/6675.html" \t "http://c.biancheng.net/view/_blank)》一节**。  
那么，既然 forward\_list 容器具有和 list 容器相同的特性，list 容器还可以提供更多的功能函数，forward\_list 容器有什么存在的必要呢？  
 当然有，forward\_list 容器底层使用单链表，也不是一无是处。比如，存储相同个数的同类型元素，单链表耗用的内存空间更少，空间利用率更高，并且对于实现某些操作单链表的执行效率也更高。

**效率高是选用 forward\_list 而弃用 list 容器最主要的原因，换句话说，只要是 list 容器和 forward\_list 容器都能实现的操作，应优先选择 forward\_list 容器。**

**一 forward\_list容器的创建**

由于 forward\_list 容器以模板类 forward\_list<T>（T 为存储元素的类型）的形式被包含在<forward\_list>头文件中，并定义在 std 命名空间中。因此，在使用该容器之前，代码中需包含下面两行代码：

#include <forward\_list>

u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;

std 命名空间也可以在使用 forward\_list 容器时额外注明，两种方式都可以。

创建 forward\_list 容器的方式，大致分为以下 5 种。  
1) 创建一个没有任何元素的空 forward\_list 容器：

std::forward\_list<int> values;

由于 forward\_list 容器在创建后也可以添加元素，因此这种创建方式很常见。  
2) 创建一个包含 n 个元素的 forward\_list 容器：

std::forward\_list<int> values(10);

通过此方式创建 values 容器，其中包含 10 个元素，每个元素的值都为相应类型的默认值（int类型的默认值为 0）。  
3) 创建一个包含 n 个元素的 forward\_list 容器，并为每个元素指定初始值。例如：

std::forward\_list<int> values(10, 5);

如此就创建了一个包含 10 个元素并且值都为 5 个 values 容器。  
4) 在已有 forward\_list 容器的情况下，通过拷贝该容器可以创建新的 forward\_list 容器。例如：

std::forward\_list<int> value1(10);

std::forward\_list<int> value2(value1);

注意，采用此方式，必须保证新旧容器存储的元素类型一致。  
  
5) 通过拷贝其他类型容器（或者普通数组）中指定区域内的元素，可以创建新的 forward\_list 容器。例如：

//拷贝普通数组，创建forward\_list容器

int a[] = { 1,2,3,4,5 };

std::forward\_list<int> values(a, a+5);

//拷贝其它类型的容器，创建forward\_list容器

std::array<int, 5>arr{ 11,12,13,14,15 };

std::forward\_list<int>values(arr.begin()+2, arr.end());//拷贝arr容器中的{13,14,15}

**二 forward\_list容器支持的成员函数**

表 2 中罗列出了 forward\_list 模板类提供的所有成员函数以及各自的功能。

表 2 forward\_list 容器可用的成员函数

|  |  |
| --- | --- |
| **成员函数** | **功能** |
| before\_begin() | 返回一个前向迭代器，其指向容器中第一个元素之前的位置。 |
| begin() | 返回一个前向迭代器，其指向容器中第一个元素的位置。 |
| end() | 返回一个前向迭代器，其指向容器中最后一个元素之后的位置。 |
| cbefore\_begin() | 和 before\_begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。 |
| empty() | 判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。 |
| max\_size() | 返回容器所能包含元素个数的最大值。这通常是一个很大的值，一般是 232-1，所以我们很少会用到这个函数。 |
| front() | 返回第一个元素的引用。 |
| assign() | 用新元素替换容器中原有内容。 |
| push\_front() | 在容器头部插入一个元素。 |
| emplace\_front() | 在容器头部生成一个元素。该函数和 push\_front() 的功能相同，但效率更高。 |
| pop\_front() | 删除容器头部的一个元素。 |
| emplace\_after() | 在指定位置之后插入一个新元素，并返回一个指向新元素的迭代器。和 insert\_after() 的功能相同，但效率更高。 |
| insert\_after() | 在指定位置之后插入一个新元素，并返回一个指向新元素的迭代器。 |
| erase\_after() | 删除容器中某个指定位置或区域内的所有元素。 |
| swap() | 交换两个容器中的元素，必须保证这两个容器中存储的元素类型是相同的。 |
| resize() | 调整容器的大小。 |
| clear() | 删除容器存储的所有元素。 |
| splice\_after() | 将某个 forward\_list 容器中指定位置或区域内的元素插入到另一个容器的指定位置之后。 |
| remove(val) | 删除容器中所有等于 val 的元素。 |
| remove\_if() | 删除容器中满足条件的元素。 |
| unique() | 删除容器中相邻的重复元素，只保留一个。 |
| merge() | 合并两个事先已排好序的 forward\_list 容器，并且合并之后的 forward\_list 容器依然是有序的。 |
| sort() | 通过更改容器中元素的位置，将它们进行排序。 |
| reverse() | 反转容器中元素的顺序。 |

除此之外，C++ 11 标准库还新增加了 begin() 和 end() 这 2 个函数，和 forward\_list 容器包含的 begin() 和 end() 成员函数不同，标准库提供的这 2 个函数的操作对象，既可以是容器，还可以是普通数组。当操作对象是容器时，它和容器包含的 begin() 和 end() 成员函数的功能完全相同；如果操作对象是普通数组，则 begin() 函数返回的是指向数组第一个元素的指针，同样 end() 返回指向数组中最后一个元素之后一个位置的指针（注意不是最后一个元素）。  
 forward\_list 容器还有一个std::swap(x , y)非成员函数（其中 x 和 y 是存储相同类型元素的 forward\_list 容器），它和 swap() 成员函数的功能完全相同，仅使用语法上有差异。  
下面的样例演示了表 2 中部分成员函数的用法：

#include <iostream>

#include <forward\_list>

using namespace *std*;

int main()

{

*std*::*forward\_list*<int> values{ 1,2,3 };

values.*emplace\_front*(4);//{4,1,2,3}

values.emplace\_after(values.before\_begin(), 5); //{5,4,1,2,3}

values.*reverse*();//{3,2,1,4,5}

for (auto it = values.*begin*(); it != values.*end*(); ++it) {

*cout* << \*it << " ";

}

return 0;

}

**运行结果为：**

**3 2 1 4 5**

**表 2 中这些成员函数的具体用法，后续学习用到时会具体讲解，感兴趣的读者，也可以通过查阅[STL手册](http://www.cplusplus.com/reference/stl/" \t "http://c.biancheng.net/view/_blank)做详细了解。**

**三 和使用forward\_list容器相关的函数**

通过表 2 我们知道，forward\_list 容器中是不提供 size() 函数的，但如果想要获取 forward\_list 容器中存储元素的个数，可以使用头文件 <iterator> 中的 dis[tan](http://c.biancheng.net/ref/tan.html" \t "http://c.biancheng.net/view/_blank)ce() 函数。举个例子：

#include <iostream>

#include <forward\_list>

#include <iterator>

using namespace *std*;

int main()

{

*std*::*forward\_list*<int> my\_words{ 1,2,3,4 };

int count = *std*::*distance*(*std*::*begin*(my\_words), *std*::*end*(my\_words));

*cout* << count;

return 0;

}

**运行结果为：**

**4**  
并且，forward\_list 容器迭代器的移动除了使用 ++ 运算符单步移动，还能使用 advance() 函数，比如：

#include <iostream>

#include <forward\_list>

using namespace *std*;

int main()

{

*std*::*forward\_list*<int> values{ 1,2,3,4 };

auto it = values.*begin*();

*advance*(it, 2);

while (it != values.*end*())

{

*cout* << \*it << " ";

++it;

}

return 0;

}

**运行结果为：**

**3 4**

**第三章 STL关联式容器**

**第一 C++ STL关联式容器是什么？**

在《[C++ STL容器](http://c.biancheng.net/view/6560.html)》一节中讲到，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 容器大致分为 2 类，即序列式容器和关联式容器。其中，序列式容器（包括 array、vector、list、deque 和 forward\_list）已经在前面章节中做了详细的介绍，从本节开始，将逐个对 C++ [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 标准库中的所有关联式容器做详细的讲解。  
提到 C++ STL 关联式容器，读者可能会以下一些疑问：

（1）关联式容器是什么，具有哪些特点？

（2）和序列式容器相比，关联式容器有什么不同？

（3）关联式容器的种类有哪些？  
别急，读完本文，这些疑问都会迎刃而解。

**一 C++ STL关联式容器是什么**

通过学习所有的序列式容器不难发现，无论是哪种序列式容器，其存储的都是 C++ 基本数据类型（诸如 int、double、float、string 等）或使用结构体自定义类型的元素。例如，如下是一个存储 int 类型元素的 vector 容器：

std::vector<int> primes {2, 3, 5, 7, 11, 13, 17, 19};  
关联式容器则大不一样，此类容器在存储元素值的同时，还会为各元素额外再配备一个值（又称为“键”，其本质也是一个 C++ 基础数据类型或自定义类型的元素），它的功能是在使用关联式容器的过程中，如果已知目标元素的键的值，则直接通过该键就可以找到目标元素，而无需再通过遍历整个容器的方式。

**弃用序列式容器，转而选用关联式容器存储元素，往往就是看中了关联式容器可以快速查找、读取或者删除所存储的元素，同时该类型容器插入元素的效率也比序列式容器高。**

也就是说，使用关联式容器存储的元素，都是一个一个的“键值对”（ <key,value> ），这是和序列式容器最大的不同。除此之外，序列式容器中存储的元素默认都是未经过排序的，而使用关联式容器存储的元素，默认会根据各元素的键值的大小做升序排序。  
 **注意，关联式容器所具备的这些特性，归咎于 STL 标准库在实现该类型容器时，底层选用了 「红黑树」这种[数据结构](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)来组织和存储各个键值对。有关红黑树组织和存储数据的方式，我们已经在数据结构中做了详细的介绍，读者可猛击《[红黑树](http://c.biancheng.net/view/vip_3433.html" \t "http://c.biancheng.net/view/_blank)》一文做详细了解。**

**二 C++ STL关联式容器种类**

C++ STL 标准库提供了 4 种关联式容器，分别为 map、set、multimap、multiset，其各自的特点如表 1 所示。

表 1 C++ STL关联式容器类别

|  |  |
| --- | --- |
| **关联式容器名称** | **特点** |
| map | 定义在 <map> 头文件中，使用该容器存储的数据，其各个元素的键必须是唯一的（即不能重复），该容器会根据各元素键的大小，默认进行升序排序（调用 std::less<T>）。 |
| set | 定义在 <set> 头文件中，使用该容器存储的数据，各个元素键和值完全相同，且各个元素的值不能重复（保证了各元素键的唯一性）。该容器会自动根据各个元素的键（其实也就是元素值）的大小进行升序排序（调用 std::less<T>）。 |
| multimap | 定义在 <map> 头文件中，和 map 容器唯一的不同在于，multimap 容器中存储元素的键可以重复。 |
| multiset | 定义在 <set> 头文件中，和 set 容器唯一的不同在于，multiset 容器中存储元素的值可以重复（一旦值重复，则意味着键也是重复的）。 |

**除此之外，C++ 11 还新增了 4 种哈希容器，即 unordered\_map、unordered\_multimap 以及 unordered\_set、unordered\_multiset。严格来说，它们也属于关联式容器，但由于哈希容器底层采用的是哈希表，而不是红黑树，因此本教程将它们分开进行讲解（有关哈希容器，将放在后续章节做详细讲解）。**

为了让读者直观地认识到关联式容器的特性，这里为 map 容器为例，编写了一个样例（如下所示）。对于该程序，读者只需体会关联式容器的特性即可，无需纠结 map 容器的具体用法。

#include <iostream>

#include <map> //使用 map 容器，必须引入该头文件

#include <string>

using namespace *std*;

int *main*()

{

//创建一个空的 map 关联式容器，该容器中存储的键值对，其中键为 string 字符串，值也为 string 字符串类型

*map*<*string*, *string*> mymap;

//向 mymap 容器中添加数据

mymap["http://c.biancheng.net/c/"] = "C语言教程";

mymap["http://c.biancheng.net/python/"] = "Python教程";

mymap["http://c.biancheng.net/java/"] = "Java教程";

//使用 map 容器的迭代器，遍历 mymap 容器，并输出其中存储的各个键值对

for (*map*<*string*, *string*>::*iterator* it = mymap.*begin*(); it != mymap.*end*(); ++it) {

//输出各个元素中的键和值

*cout* << it->*first* << " => " << it->*second* << '\n';

}

return 0;

}

**程序输出结果为：**

**http://c.biancheng.net/c/ => C语言教程  
http://c.biancheng.net/java/ => Java教程  
http://c.biancheng.net/python/ => Python教程**

通过分析该程序的执行过程不难看出，mymap 关联式容器中的存储了以下 3 个键值对：

**<"http://c.biancheng.net/c/", "C语言教程">  
<"http://c.biancheng.net/python/", "Python教程">  
<"http://c.biancheng.net/java/", "Java教程">**

但需要注意的一点是，由于 map 容器在存储元素时，会根据各个元素键的大小自动调整元素的顺序（默认按照升序排序），因此该容器最终存储的元素顺序为：

**<"http://c.biancheng.net/c/", "C语言教程">  
<"http://c.biancheng.net/java/", "Java教程">  
<"http://c.biancheng.net/python/", "Python教程">**

有关 map 容器以及表 1 中其它关联式容器的具体用法，后续章节会做详细介绍。

**第二 C++ STL pair用法详解**

我们知道，关联式容器存储的是“键值对”形式的数据，比如：

**<"C语言教程", "http://c.biancheng.net/c/">  
<"[Python](http://c.biancheng.net/python/" \t "http://c.biancheng.net/view/_blank)教程", "http://c.biancheng.net/python/">  
<"[Java](http://c.biancheng.net/java/" \t "http://c.biancheng.net/view/_blank)教程", "http://c.biancheng.net/java/">**

如上所示，每行都表示一个键值对，其中第一个元素作为键（key），第二个元素作为值（value）。

**注意，基于各个关联式容器存储数据的特点，只有各个键值对中的键和值全部对应相等时，才能使用 set 和 multiset 关联式容器存储，否则就要选用 map 或者 multimap 关联式容器。**

考虑到“键值对”并不是普通类型数据，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 标准库提供了 pair 类模板，其专门用来将 2 个普通元素 first 和 second（可以是 C++ 基本数据类型、结构体、类自定的类型）创建成一个新元素<first, second>。通过其构成的元素格式不难看出，使用 pair 类模板来创建“键值对”形式的元素，再合适不过。  
 **注意，pair 类模板定义在<utility>头文件中，所以在使用该类模板之前，需引入此头文件。另外值得一提的是，在 C++ 11 标准之前，pair 类模板中提供了以下 3 种构造函数：**

**#1) 默认构造函数，即创建空的 pair 对象**

**pair();**

**#2) 直接使用 2 个元素初始化成 pair 对象**

**pair (const first\_type& a, const second\_type& b);**

**#3) 拷贝（复制）构造函数，即借助另一个 pair 对象，创建新的 pair 对象**

**template<class U, class V> pair (const pair<U,V>& pr);**

在 C++ 11 标准中，在引入[右值引用](http://c.biancheng.net/view/439.html" \t "http://c.biancheng.net/view/_blank)的基础上，pair 类模板中又增添了如下 2 个构造函数：

**#4) 移动构造函数**

**template<class U, class V> pair (pair<U,V>&& pr);**

**#5) 使用右值引用参数，创建 pair 对象**

**template<class U, class V> pair (U&& a, V&& b);**

除此之外，C++ 11 标准中 pair 类模板还新增加了如下一种构造函数：

**pair (piecewise\_construct\_t pwc, tuple<Args1...> first\_args, tuple<Args2...> second\_args);**

**该构造 pair 类模板的方式很少用到，因此本节不再对其进行详细介绍，感兴趣的读者可自行查阅资料。**

#include <iostream>

#include <utility> // pair

#include <string> // string

using namespace *std*;

int *main*() {

// 调用构造函数 1，也就是默认构造函数

*pair* <*string*, double> pair1;

// 调用第 2 种构造函数

*pair* <*string*, *string*> pair2("STL教程", "http://c.biancheng.net/stl/");

// 调用拷贝构造函数

*pair* <*string*, *string*> pair3(pair2);

//调用移动构造函数

*pair* <*string*, *string*> pair4(*make\_pair*("C++教程", "http://c.biancheng.net/cplus/"));

// 调用第 5 种构造函数

*pair* <*string*, *string*> pair5(*string*("Python教程"), *string*("http://c.biancheng.net/python/"));

*cout* << "pair1: " << pair1.*first* << " " << pair1.*second* << *endl*;

*cout* << "pair2: " << pair2.*first* << " " << pair2.*second* << *endl*;

*cout* << "pair3: " << pair3.*first* << " " << pair3.*second* << *endl*;

*cout* << "pair4: " << pair4.*first* << " " << pair4.*second* << *endl*;

*cout* << "pair5: " << pair5.*first* << " " << pair5.*second* << *endl*;

return 0;

}

**程序输出结果为：**

**pair1:  0  
pair2: STL教程 http://c.biancheng.net/stl/  
pair3: STL教程 http://c.biancheng.net/stl/  
pair4: C++教程 http://c.biancheng.net/cplus/  
pair5: Python教程 http://c.biancheng.net/python/**

上面程序在创建 pair4 对象时，调用了 make\_pair() 函数，它也是 <utility> 头文件提供的，其功能是生成一个 pair 对象。因此，当我们将 make\_pair() 函数的返回值（是一个临时对象）作为参数传递给 pair() 构造函数时，其调用的是移动构造函数，而不是拷贝构造函数。  
  
在上面程序的基础上，C++ 11 还允许我们手动为 pair1 对象赋值，比如：

**pair1.first = "Java教程";**

**pair1.second = "http://c.biancheng.net/java/";**

**cout << "new pair1: " << pair1.first << " " << pair1.second << endl;**

**执行结果为：**

**new pair1: Java教程 http://c.biancheng.net/java/**

同时，上面程序中 pair4 对象的创建过程，还可以写入如下形式，它们是完全等价的：

**pair <string, string> pair4 = make\_pair("C++教程", "http://c.biancheng.net/cplus/");**

**cout << "pair4: " << pair4.first << " " << pair4.second << endl;**<utility>头文件中除了提供创建 pair 对象的方法之外，还为 pair 对象重载了 <、<=、>、>=、==、!= 这 6 的运算符，其运算规则是：对于进行比较的 2 个 pair 对象，先比较 pair.first 元素的大小，如果相等则继续比较 pair.second 元素的大小。

**注意，对于进行比较的 2 个 pair 对象，其对应的键和值的类型比较相同，否则将没有可比性，同时编译器提示没有相匹配的运算符，即找不到合适的重载运算符。**

#include <iostream>

#include <utility> // pair

#include <string> // string

using namespace *std*;

int *main*() {

*pair* <*string*, int> pair1("STL教程", 20);

*pair* <*string*, int> pair2("C++教程", 20);

*pair* <*string*, int> pair3("C++教程", 30);

//pair1和pair2的key不同，value相同

if (pair1 != pair2) {

*cout* << "pair != pair2" << *endl*;

}

//pair2和pair3的key相同，value不同

if (pair2 != pair3) {

*cout* << "pair2 != pair3" << *endl*;

}

return 0;

}

**程序执行结果为：**

**pair != pair2  
pair2 != pair3**

最后需要指出的是，pair类模板还提供有一个 swap() 成员函数，能够互换 2 个 pair 对象的键值对，其操作成功的前提是这 2 个 pair 对象的键和值的类型要相同。例如：

#include <iostream>

#include <utility> // pair

#include <string> // string

using namespace *std*;

int *main*() {

*pair* <*string*, int> pair1("pair", 10);

*pair* <*string*, int> pair2("pair2", 20);

//交换 pair1 和 pair2 的键值对

pair1.*swap*(pair2);

*cout* << "pair1: " << pair1.*first* << " " << pair1.*second* << *endl*;

*cout* << "pair2: " << pair2.*first* << " " << pair2.*second* << *endl*;

return 0;

}

**程序执行结果为：**

**pair1: pair2 20  
pair2: pair 10**

**第三 C++ STL map容器详解**

作为关联式容器的一种，map 容器存储的都是 pair 对象，也就是用 pair 类模板创建的键值对。其中，各个键值对的键和值可以是任意数据类型，包括 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 基本数据类型（int、double 等）、使用结构体或类自定义的类型。

**通常情况下，map 容器中存储的各个键值对都选用 string 字符串作为键的类型。**

与此同时，在使用 map 容器存储多个键值对时，该容器会自动根据各键值对的键的大小，按照既定的规则进行排序。默认情况下，map 容器选用std::less<T>排序规则（其中 T 表示键的数据类型），其会根据键的大小对所有键值对做升序排序。当然，根据实际情况的需要，我们可以手动指定 map 容器的排序规则，既可以选用 [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 标准库中提供的其它排序规则（比如std::greater<T>），也可以自定义排序规则。

**关于如何自定义 map 容器的排序规则，后续章节会做详细讲解。**

**另外需要注意的是，使用 map 容器存储的各个键值对，键的值既不能重复也不能被修改。换句话说，map 容器中存储的各个键值对不仅键的值独一无二，键的类型也会用 const 修饰，这意味着只要键值对被存储到 map 容器中，其键的值将不能再做任何修改。**

**前面提到，map 容器存储的都是 pair 类型的键值对元素，更确切的说，该容器存储的都是 pair<const K, T> 类型（其中 K 和 T 分别表示键和值的数据类型）的键值对元素。**

**map 容器定义在 <map> 头文件中，并位于 std 命名空间中。因此，如果想使用 map 容器，代码中应包含如下语句：**

**#include <map>**

**u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;**

注意，第二行代码不是必需的，如果不用，则后续程序中在使用 map 容器时，需手动注明 std 命名空间（强烈建议初学者使用）。

**map 容器的模板定义如下：**

**template < class Key, // 指定键（key）的类型**

**class T, // 指定值（value）的类型**

**class Compare = less<Key>, // 指定排序规则**

**class Alloc = allocator<pair<const Key,T> > // 指定分配器对象的类型**

**> class map;**

可以看到，map 容器模板有 4 个参数，其中后 2 个参数都设有默认值。大多数场景中，我们只需要设定前 2 个参数的值，有些场景可能会用到第 3 个参数，但最后一个参数几乎不会用到。

**一 创建C++ map容器的几种方法**

map 容器的模板类中包含多种构造函数，因此创建 map 容器的方式也有多种，下面就几种常用的创建 map 容器的方法，做一一讲解。  
1) 通过调用 map 容器类的默认构造函数，可以创建出一个空的 map 容器，比如：

**std::map<std::string, int>myMap;**

如果程序中已经默认指定了 std 命令空间，这里可以省略 std::。

通过此方式创建出的 myMap 容器，初始状态下是空的，即没有存储任何键值对。鉴于空 map 容器可以根据需要随时添加新的键值对，因此创建空 map 容器是比较常用的。  
2) 当然在创建 map 容器的同时，也可以进行初始化，比如：

**std::map<std::string, int>myMap{ {"C语言教程",10},{"STL教程",20} };**

由此，myMap 容器在初始状态下，就包含有 2 个键值对。  
**再次强调，map 容器中存储的键值对，其本质都是 pair 类模板创建的 pair 对象。因此，下面程序也可以创建出一模一样的 myMap 容器：**

**std::map<std::string, int>myMap{std::make\_pair("C语言教程",10),std::make\_pair("STL教程",20)};**  
3) 除此之外，在某些场景中，可以利用先前已创建好的 map 容器，再创建一个新的 map 容器。例如：

**std::map<std::string, int>newMap(myMap);**

由此，通过调用 map 容器的拷贝（复制）构造函数，即可成功创建一个和 myMap 完全一样的 newMap 容器。  
  
**C++ 11 标准中，还为 map 容器增添了移动构造函数。当有临时的 map 对象作为参数，传递给要初始化的 map 容器时，此时就会调用移动构造函数。举个例子：**

**#创建一个会返回临时 map 对象的函数**

**std::map<std::string,int> disMap() {**

**std::map<std::string, int>tempMap{ {"C语言教程",10},{"STL教程",20} };**

**return tempMap;**

**}**

**//调用 map 类模板的移动构造函数创建 newMap 容器**

**std::map<std::string, int>newMap(disMap());**

注意，无论是调用复制构造函数还是调用拷贝构造函数，都必须保证这 2 个容器的类型完全一致。

4) map 类模板还支持取已建 map 容器中指定区域内的键值对，创建并初始化新的 map 容器。例如：

**std::map<std::string, int>myMap{ {"C语言教程",10},{"STL教程",20} };**

**std::map<std::string, int>newMap(++myMap.begin(), myMap.end());**

这里，通过调用 map 容器的双向迭代器，实现了在创建 newMap 容器的同时，将其初始化为包含一个 {"STL教程",20} 键值对的容器。

有关 map 容器迭代器，后续章节会做详细讲解。

5) 当然，在以上几种创建 map 容器的基础上，我们都可以手动修改 map 容器的排序规则。默认情况下，map 容器调用 std::less<T> 规则，根据容器内各键值对的键的大小，对所有键值对做升序排序。  
  
因此，如下 2 行创建 map 容器的方式，其实是等价的：

**std::map<std::string, int>myMap{ {"C语言教程",10},{"STL教程",20} };**

**std::map<std::string, int, std::less<std::string> >myMap{ {"C语言教程",10},{"STL教程",20} };**

以上 2 中创建方式生成的 myMap 容器，其内部键值对排列的顺序为：

<"C语言教程", 10>  
<"STL教程", 20>

下面程序手动修改了 myMap 容器的排序规则，令其作降序排序：

std::map<std::string, int, std::greater<std::string> >myMap{ {"C语言教程",10},{"STL教程",20} };

此时，myMap 容器内部键值对排列的顺序为：

**<"STL教程", 20>  
<"C语言教程", 10>**

在某些特定场景中，我们还需要为 map 容器自定义排序规则，此部分知识后续将利用整整一节做重点讲解。

**二C++ map容器包含的成员方法**

表 1 列出了 map 容器提供的常用成员方法以及各自的功能。

表 1 C++ map容器常用成员方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| begin() | 返回指向容器中第一个（注意，是已排好序的第一个）键值对的双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| end() | 返回指向容器最后一个元素（注意，是已排好序的最后一个）所在位置后一个位置的双向迭代器，通常和 begin() 结合使用。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| rbegin() | 返回指向最后一个（注意，是已排好序的最后一个）元素的反向双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| rend() | 返回指向第一个（注意，是已排好序的第一个）元素所在位置前一个位置的反向双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| find(key) | 在 map 容器中查找键为 key 的键值对，如果成功找到，则返回指向该键值对的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| lower\_bound(key) | 返回一个指向当前 map 容器中第一个大于或等于 key 的键值对的双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| upper\_bound(key) | 返回一个指向当前 map 容器中第一个大于 key 的键值对的迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| equal\_range(key) | 该方法返回一个 pair 对象（包含 2 个双向迭代器），其中 pair.first 和 lower\_bound() 方法的返回值等价，pair.second 和 upper\_bound() 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含的键为 key 的键值对（map 容器键值对唯一，因此该范围最多包含一个键值对）。 |
| empty() | 若容器为空，则返回 true；否则 false。 |
| size() | 返回当前 map 容器中存有键值对的个数。 |
| max\_size() | 返回 map 容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。 |
| operator[] | map容器重载了 [] 运算符，只要知道 map 容器中某个键值对的键的值，就可以向获取数组中元素那样，通过键直接获取对应的值。 |
| at(key) | 找到 map 容器中 key 键对应的值，如果找不到，该函数会引发 out\_of\_range 异常。 |
| insert() | 向 map 容器中插入键值对。 |
| erase() | 删除 map 容器指定位置、指定键（key）值或者指定区域内的键值对。后续章节还会对该方法做重点讲解。 |
| swap() | 交换 2 个 map 容器中存储的键值对，这意味着，操作的 2 个键值对的类型必须相同。 |
| clear() | 清空 map 容器中所有的键值对，即使 map 容器的 size() 为 0。 |
| emplace() | 在当前 map 容器中的指定位置处构造新键值对。其效果和插入键值对一样，但效率更高。 |
| emplace\_hint() | 在本质上和 emplace() 在 map 容器中构造新键值对的方式是一样的，不同之处在于，使用者必须为该方法提供一个指示键值对生成位置的迭代器，并作为该方法的第一个参数。 |
| count(key) | 在当前 map 容器中，查找键为 key 的键值对的个数并返回。注意，由于 map 容器中各键值对的键的值是唯一的，因此该函数的返回值最大为 1。 |

下面的样例演示了表 1 中部分成员方法的用法：

#include <iostream>

#include <map> // map

#include <string> // string

using namespace *std*;

int *main*() {

//创建空 map 容器，默认根据个键值对中键的值，对键值对做降序排序

*std*::*map*<*std*::*string*, *std*::*string*, *std*::*greater*<*std*::*string*>>myMap;

//调用 emplace() 方法，直接向 myMap 容器中指定位置构造新键值对

myMap.*emplace*("C语言教程", "http://c.biancheng.net/c/");

myMap.*emplace*("Python教程", "http://c.biancheng.net/python/");

myMap.*emplace*("STL教程", "http://c.biancheng.net/stl/");

//输出当前 myMap 容器存储键值对的个数

*cout* << "myMap size==" << myMap.*size*() << *endl*;

//判断当前 myMap 容器是否为空

if (!myMap.*empty*()) {

//借助 myMap 容器迭代器，将该容器的键值对逐个输出

for (auto i = myMap.*begin*(); i != myMap.*end*(); ++i) {

*cout* << i->*first* << " " << i->*second* << *endl*;

}

}

return 0;

}

**程序执行结果为：**

**myMap size==3  
STL教程 http://c.biancheng.net/stl/  
Python教程 http://c.biancheng.net/python/  
C语言教程 http://c.biancheng.net/c/**

**第四 C++ STL map容器迭代器用法详解**

无论是前面学习的序列式容器，还是关联式容器，要想实现遍历操作，就必须要用到该类型容器的迭代器。当然，map 容器也不例外。  
 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 标准库为 map 容器配备的是双向迭代器（bidirectional iterator）。这意味着，map 容器迭代器只能进行 ++p、p++、--p、p--、\*p 操作，并且迭代器之间只能使用 == 或者 != 运算符进行比较。  
 值得一提的是，相比序列式容器，map 容器提供了更多的成员方法（如表 1 所示），通过调用它们，我们可以轻松获取具有指定含义的迭代器。

表 1 map 容器迭代器相关成员方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| begin() | 返回指向容器中第一个（注意，是已排好序的第一个）键值对的双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| end() | 返回指向容器最后一个元素（注意，是已排好序的最后一个）所在位置后一个位置的双向迭代器，通常和 begin() 结合使用。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| rbegin() | 返回指向最后一个（注意，是已排好序的最后一个）元素的反向双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| rend() | 返回指向第一个（注意，是已排好序的第一个）元素所在位置前一个位置的反向双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| find(key) | 在 map 容器中查找键为 key 的键值对，如果成功找到，则返回指向该键值对的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| lower\_bound(key) | 返回一个指向当前 map 容器中第一个大于或等于 key 的键值对的双向迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| upper\_bound(key) | 返回一个指向当前 map 容器中第一个大于 key 的键值对的迭代器。如果 map 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| equal\_range(key) | 该方法返回一个 pair 对象（包含 2 个双向迭代器），其中 pair.first 和 lower\_bound() 方法的返回值等价，pair.second 和 upper\_bound() 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含的键为 key 的键值对（map 容器键值对唯一，因此该范围最多包含一个键值对）。 |

表 1 中多数的成员方法，诸如 begin()、end() 等，在学习序列式容器时已经多次使用过，它们的功能如图 2 所示。

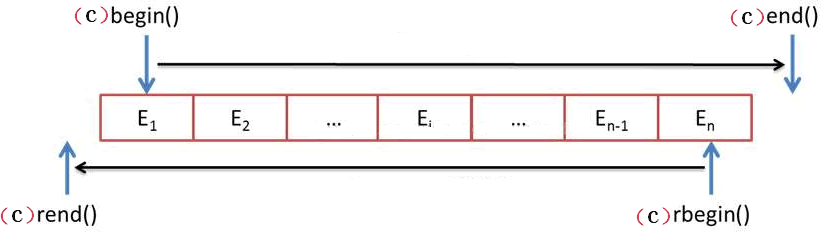


图 2 表 1 部分成员方法的功能示意图

**注意，图中 Ei 表示的是 pair 类对象，即键值对。对于 map 容器来说，每个键值对的键的值都必须保证是唯一的。**

下面程序以 begin()/end() 组合为例，演示了如何遍历 map 容器：

#include <iostream>

#include <map> // pair

#include <string> // string

using namespace *std*;

int *main*() {

//创建并初始化 map 容器

*std*::*map*<*std*::*string*, *std*::*string*>myMap{ {"STL教程","http://c.biancheng.net/stl/"},{"C语言教程","http://c.biancheng.net/c/"} };

//调用 begin()/end() 组合，遍历 map 容器

for (auto iter = myMap.*begin*(); iter != myMap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**C语言教程 http://c.biancheng.net/c/  
STL教程 http://c.biancheng.net/stl/**

**读者可自行尝试使用其他组合（如 cbegin()/cend()、 rbegin()/rend() 等）遍历 map 容器。**

除此之外，map 类模板中还提供了 find() 成员方法，它能帮我们查找指定 key 值的键值对，如果成功找到，则返回一个指向该键值对的双向迭代器；反之，其功能和 end() 方法相同。

举个例子：

#include <iostream>

#include <map> // pair

#include <string> // string

using namespace *std*;

int *main*() {

//创建并初始化 map 容器

*std*::*map*<*std*::*string*, *std*::*string*>myMap{ {"STL教程","http://c.biancheng.net/stl/"},

{"C语言教程","http://c.biancheng.net/c/"},

{"Java教程","http://c.biancheng.net/java/"} };

//查找键为 "Java教程" 的键值对

auto iter = myMap.*find*("Java教程");

//从 iter 开始，遍历 map 容器

for (; iter != myMap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**Java教程 http://c.biancheng.net/java/  
STL教程 http://c.biancheng.net/stl/**

此程序中，创建并初始化的 myMap 容器，默认会根据各键值对中键的值，对各键值对做升序排序，其排序的结果为：

**<"C语言教程","http://c.biancheng.net/c/">  
<"Java教程","http://c.biancheng.net/java/">  
<"STL教程","http://c.biancheng.net/stl/">**

在此基础上，通过调用 find() 方法，我们可以得到一个指向键为 "Java教程" 的键值对的迭代器，由此当使用 [for 循环](http://c.biancheng.net/view/172.html" \t "http://c.biancheng.net/view/_blank)从该迭代器出开始遍历时，就只会遍历到最后 2 个键值对。  
同时，map 类模板中还提供有 lower\_bound(key) 和 upper\_bound(key) 成员方法，它们的功能是类似的，唯一的区别在于：

（1）lower\_bound(key) 返回的是指向第一个键不小于 key 的键值对的迭代器；

（2）upper\_bound(key) 返回的是指向第一个键大于 key 的键值对的迭代器；  
下面程序演示了它们的功能：

#include <iostream>

#include <map> // pair

#include <string> // string

using namespace *std*;

int *main*() {

//创建并初始化 map 容器

*std*::*map*<*std*::*string*, *std*::*string*>myMap{ {"STL教程","http://c.biancheng.net/stl/"},

{"C语言教程","http://c.biancheng.net/c/"},

{"Java教程","http://c.biancheng.net/java/"} };

//找到第一个键的值不小于 "Java教程" 的键值对

auto iter = myMap.*lower\_bound*("Java教程");

*cout* << "lower：" << iter->*first* << " " << iter->*second* << *endl*;

//找到第一个键的值大于 "Java教程" 的键值对

iter = myMap.*upper\_bound*("Java教程");

*cout* << "upper：" << iter->*first* << " " << iter->*second* << *endl*;

return 0;

}

**程序执行结果为：**

**lower：Java教程 http://c.biancheng.net/java/  
upper：STL教程 http://c.biancheng.net/stl/**

**lower\_bound(key) 和 upper\_bound(key) 更多用于 multimap 容器，在 map 容器中很少用到。**  
 equal\_range(key) 成员方法可以看做是 lower\_bound(key) 和 upper\_bound(key) 的结合体，该方法会返回一个 pair 对象，其中的 2 个元素都是迭代器类型，其中 pair.first 实际上就是 lower\_bound(key) 的返回值，而 pair.second 则等同于 upper\_bound(key) 的返回值。  
 显然，equal\_range(key) 成员方法表示的一个范围，位于此范围中的键值对，其键的值都为 key。举个例子：

#include <iostream>

#include <utility> //pair

#include <map> // map

#include <string> // string

using namespace *std*;

int *main*() {

//创建并初始化 map 容器

*std*::*map*<*string*, *string*>myMap{ {"STL教程","http://c.biancheng.net/stl/"},

{"C语言教程","http://c.biancheng.net/c/"},

{"Java教程","http://c.biancheng.net/java/"} };

//创建一个 pair 对象，来接收 equal\_range() 的返回值

*pair* <*std*::*map*<*string*, *string*>::*iterator*, *std*::*map*<*string*, *string*>::*iterator*> myPair = myMap.*equal\_range*("C语言教程");

//通过遍历，输出 myPair 指定范围内的键值对

for (auto iter = myPair.*first*; iter != myPair.*second*; ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**C语言教程 http://c.biancheng.net/c/**

和 lower\_bound(key)、upper\_bound(key) 一样，该方法也更常用于 multimap 容器，因为 map 容器中各键值对的键的值都是唯一的，因此通过 map 容器调用此方法，其返回的范围内最多也只有 1 个键值对。

**第五 C++ STL map获取键对应值的几种方法**

我们知道，map 容器中存储的都是 pair 类型的键值对，但几乎在所有使用 map 容器的场景中，经常要做的不是找到指定的 pair 对象（键值对），而是从该容器中找到某个键对应的值。

**注意，使用 map 容器存储的各个键值对，其键的值都是唯一的，因此指定键对应的值最多有 1 个。**

庆幸的是，map 容器的类模板中提供了以下 2 种方法，可直接获取 map 容器指定键对应的值。  
1) map 类模板中对[ ]运算符进行了重载，这意味着，类似于借助数组下标可以直接访问数组中元素，通过指定的键，我们可以轻松获取 map 容器中该键对应的值。  
  
举个例子：

#include <iostream>

#include <map> // map

#include <string> // string

using namespace *std*;

int *main*() {

//创建并初始化 map 容器

*std*::*map*<*std*::*string*, *std*::*string*>myMap{ {"STL教程","http://c.biancheng.net/stl/"},

{"C语言教程","http://c.biancheng.net/c/"},

{"Java教程","http://c.biancheng.net/java/"} };

*string* cValue = myMap["C语言教程"];

*cout* << cValue << *endl*;

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/c/**

可以看到，在第 11 行代码中，通过指定键的值为 "C语言教程"，借助重载的 [ ] 运算符，就可以在 myMap 容器中直接找到该键对应的值。  
 注意，只有当 map 容器中确实存有包含该指定键的键值对，借助重载的 [ ] 运算符才能成功获取该键对应的值；反之，若当前 map 容器中没有包含该指定键的键值对，则此时使用 [ ] 运算符将不再是访问容器中的元素，而变成了向该 map 容器中增添一个键值对。其中，该键值对的键用 [ ] 运算符中指定的键，其对应的值取决于 map 容器规定键值对中值的数据类型，如果是基本数据类型，则值为 0；如果是 string 类型，其值为 ""，即空字符串（即使用该类型的默认值作为键值对的值）。  
  
举个例子：

#include <iostream>

#include <map> // map

#include <string> // string

using namespace *std*;

int *main*() {

//创建空 map 容器

*std*::*map*<*std*::*string*, int>myMap;

int cValue = myMap["C语言教程"];

for (auto i = myMap.*begin*(); i != myMap.*end*(); ++i) {

*cout* << i->*first* << " " << i->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**C语言教程 0**

显然，对于空的 myMap 容器来说，其内部没有以 "C语言教程" 为键的键值对，这种情况下如果使用 [ ] 运算符获取该键对应的值，其功能就转变成了向该 myMap 容器中添加一个<"C语言教程",0>键值对（由于 myMap 容器规定各个键值对的值的类型为 int，该类型的默认值为 0）。  
 实际上，[ ] 运算符确实有“为 map 容器添加新键值对”的功能，但前提是要保证新添加键值对的键和当前 map 容器中已存储的键值对的键都不一样。例如：

#include <iostream>

#include <map> // map

#include <string> // string

using namespace *std*;

int *main*() {

//创建空 map 容器

*std*::*map*<*string*, *string*>myMap;

myMap["STL教程"] = "http://c.biancheng.net/java/";

myMap["Python教程"] = "http://c.biancheng.net/python/";

myMap["STL教程"] = "http://c.biancheng.net/stl/";

for (auto i = myMap.*begin*(); i != myMap.*end*(); ++i) {

*cout* << i->*first* << " " << i->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**Python教程 http://c.biancheng.net/python/  
STL教程 http://c.biancheng.net/stl/**

注意，程序中第 9 行代码已经为 map 容器添加了一个以 "STL教程" 作为键的键值对，则第 11 行代码的作用就变成了修改该键对应的值，而不再是为 map 容器添加新键值对。

2) 除了借助 [ ] 运算符获取 map 容器中指定键对应的值，还可以使用 at() 成员方法。和前一种方法相比，at() 成员方法也需要根据指定的键，才能从容器中找到该键对应的值；不同之处在于，如果在当前容器中查找失败，该方法不会向容器中添加新的键值对，而是直接抛出 out\_of\_range 异常。  
举个例子：

#include <iostream>

#include <map> // map

#include <string> // string

using namespace *std*;

int *main*() {

//创建并初始化 map 容器

*std*::*map*<*std*::*string*, *std*::*string*>myMap{ {"STL教程","http://c.biancheng.net/stl/"},

{"C语言教程","http://c.biancheng.net/c/"},

{"Java教程","http://c.biancheng.net/java/"} };

*cout* << myMap.*at*("C语言教程") << *endl*;

//下面一行代码会引发 out\_of\_range 异常

//cout << myMap.at("Python教程") << endl;

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/c/**

程序第 12 行代码处，通过 myMap 容器调用  at() 成员方法，可以成功找到键为 "C语言教程" 的键值对，并返回该键对应的值；而第 14 行代码，由于当前 myMap 容器中没有以 "Python教程" 为键的键值对，会导致 at() 成员方法查找失败，并抛出 out\_of\_range 异常。  
 除了可以直接获取指定键对应的值之外，还可以借助 find() 成员方法间接实现此目的。和以上 2 种方式不同的是，该方法返回的是一个迭代器，即如果查找成功，该迭代器指向查找到的键值对；反之，则指向 map 容器最后一个键值对之后的位置（和 end() 成功方法返回的迭代器一样）。  
举个例子：

#include <iostream>

#include <map> // map

#include <string> // string

using namespace *std*;

int *main*() {

//创建并初始化 map 容器

*std*::*map*<*std*::*string*, *std*::*string*>myMap{ {"STL教程","http://c.biancheng.net/stl/"},

{"C语言教程","http://c.biancheng.net/c/"},

{"Java教程","http://c.biancheng.net/java/"} };

*map*< *std*::*string*, *std*::*string* >::*iterator* myIter = myMap.*find*("C语言教程");

*cout* << myIter->*first* << " " << myIter->*second* << *endl*;

return 0;

}

**程序执行结果为：**

**C语言教程 http://c.biancheng.net/c/**

**注意，此程序中如果 find() 查找失败，会导致第 13 行代码运行出错。因为当 find() 方法查找失败时，其返回的迭代器指向的是容器中最后一个键值对之后的位置，即不指向任何有意义的键值对，也就没有所谓的 first 和 second 成员了。**  
 如果以上方法都不适用，我们还可以遍历整个 map 容器，找到包含指定键的键值对，进而获取该键对应的值。比如：

#include <iostream>

#include <map> // map

#include <string> // string

using namespace *std*;

int *main*() {

//创建并初始化 map 容器

*std*::*map*<*std*::*string*, *std*::*string*>myMap{ {"STL教程","http://c.biancheng.net/stl/"},

{"C语言教程","http://c.biancheng.net/c/"},

{"Java教程","http://c.biancheng.net/java/"} };

for (auto iter = myMap.*begin*(); iter != myMap.*end*(); ++iter) {

//调用 string 类的 compare() 方法，找到一个键和指定字符串相同的键值对

if (!iter->*first*.*compare*("C语言教程")) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

}

return 0;

}

**程序执行结果为：**

**C语言教程 http://c.biancheng.net/c/**

**本节所介绍的几种方法中，仅从“在 map 容器存储的键值对中，获取指定键对应的值”的角度出发，更推荐使用 at() 成员方法，因为该方法既简单又安全。**

**第六 C++ STL map insert()插入数据的4种方式**

前面讲过，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) map 类模板中对[ ]运算符进行了重载，即根据使用场景的不同，借助[ ]运算符可以实现不同的操作。举个例子：

#include <iostream>

#include <map> //map

#include <string> //string

using namespace *std*;

int *main*()

{

*std*::*map*<*string*, *string*> mymap{ {"STL教程","http://c.biancheng.net/java/"} };

//获取已存储键值对中，指定键对应的值

*cout* << mymap["STL教程"] << *endl*;

//向 map 容器添加新键值对

mymap["Python教程"] = "http://c.biancheng.net/python/";

//修改 map 容器已存储键值对中，指定键对应的值

mymap["STL教程"] = "http://c.biancheng.net/stl/";

for (auto iter = mymap.*begin*(); iter != mymap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/java/  
Python教程 http://c.biancheng.net/python/  
STL教程 http://c.biancheng.net/stl/**

可以看到，当操作对象为 map 容器中已存储的键值对时，则借助 [ ] 运算符，既可以获取指定键对应的值，还能对指定键对应的值进行修改；反之，若 map 容器内部没有存储以 [ ] 运算符内指定数据为键的键值对，则使用 [ ] 运算符会向当前 map 容器中添加一个新的键值对。  
 实际上，除了使用 [ ] 运算符实现向 map 容器中添加新键值对外，map 类模板中还提供有 insert() 成员方法，该方法专门用来向 map 容器中插入新的键值对。

注意，这里所谓的“插入”，指的是 insert() 方法可以将新的键值对插入到 map 容器中的指定位置，但这与 map 容器会自动对存储的键值对进行排序并不冲突。当使用 insert() 方法向 map 容器的指定位置插入新键值对时，其底层会先将新键值对插入到容器的指定位置，如果其破坏了 map 容器的有序性，该容器会对新键值对的位置进行调整。

**自 C++ 11 标准后，insert() 成员方法的用法大致有以下 4 种。**  
1) 无需指定插入位置，直接将键值对添加到 map 容器中。insert() 方法的语法格式有以下 2 种：

//1、引用传递一个键值对  
pair<iterator,bool> insert (const value\_type& val);  
//2、以右值引用的方式传递键值对  
template <class P>  
    pair<iterator,bool> insert (P&& val);

其中，val 参数表示键值对变量，同时该方法会返回一个 pair 对象，其中 pair.first 表示一个迭代器，pair.second 为一个 bool 类型变量：

如果成功插入 val，则该迭代器指向新插入的 val，bool 值为 true；

如果插入 val 失败，则表明当前 map 容器中存有和 val 的键相同的键值对（用 p 表示），此时返回的迭代器指向 p，bool 值为 false。

**以上 2 种语法格式的区别在于传递参数的方式不同，即无论是局部定义的键值对变量还是全局定义的键值对变量，都采用普通引用传递的方式；而对于临时的键值对变量，则以右值引用的方式传参。有关右值引用，可阅读《[C++右值引用](http://c.biancheng.net/view/439.html" \t "http://c.biancheng.net/view/_blank)》一文做详细了解。**

举个例子：

#include <iostream>

#include <map> //map

#include <string> //string

using namespace *std*;

int *main*()

{

//创建一个空 map 容器

*std*::*map*<*string*, *string*> mymap;

//创建一个真实存在的键值对变量

*std*::*pair*<*string*, *string*> STL = { "STL教程","http://c.biancheng.net/stl/" };

//创建一个接收 insert() 方法返回值的 pair 对象

*std*::*pair*<*std*::*map*<*string*, *string*>::*iterator*, bool> ret;

//插入 STL，由于 STL 并不是临时变量，因此会以第一种方式传参

ret = mymap.*insert*(STL);

*cout* << "ret.iter = <{" << ret.*first*->*first* << ", " << ret.*first*->*second* << "}, " << ret.*second* << ">" << *endl*;

//以右值引用的方式传递临时的键值对变量

ret = mymap.*insert*({ "C语言教程","http://c.biancheng.net/c/" });

*cout* << "ret.iter = <{" << ret.*first*->*first* << ", " << ret.*first*->*second* << "}, " << ret.*second* << ">" << *endl*;

//插入失败样例

ret = mymap.*insert*({ "STL教程","http://c.biancheng.net/java/" });

*cout* << "ret.iter = <{" << ret.*first*->*first* << ", " << ret.*first*->*second* << "}, " << ret.*second* << ">" << *endl*;

return 0;

}

**程序执行结果为：**

**ret.iter = <{STL教程, http://c.biancheng.net/stl/}, 1>  
ret.iter = <{C语言教程, http://c.biancheng.net/c/}, 1>  
ret.iter = <{STL教程, http://c.biancheng.net/stl/}, 0>**

从执行结果中不难看出，程序中共执行了 3 次插入操作，其中成功了 2 次，失败了 1 次：

（1）对于插入成功的 insert() 方法，其返回的 pair 对象中包含一个指向新插入键值对的迭代器和值为 1 的 bool 变量

（2）对于插入失败的 insert() 方法，同样会返回一个 pair 对象，其中包含一个指向 map 容器中键为 "STL教程" 的键值对和值为 0 的 bool 变量。

另外，在程序中的第 21 行代码，还可以使用如下 2 种方式创建临时的键值对变量，它们是等价的：

**//调用 pair 类模板的构造函数**

**ret = mymap.insert(pair<string,string>{ "C语言教程","http://c.biancheng.net/c/" });**

**//调用 make\_pair() 函数**

**ret = mymap.insert(make\_pair("C语言教程", "http://c.biancheng.net/c/"));**

2) 除此之外，insert() 方法还支持向 map 容器的指定位置插入新键值对，该方法的语法格式如下：

//以普通引用的方式传递 val 参数  
iterator insert (const\_iterator position, const value\_type& val);  
//以右值引用的方式传递 val 键值对参数  
template <class P>  
    iterator insert (const\_iterator position, P&& val);

其中 val 为要插入的键值对变量。注意，和第 1 种方式的语法格式不同，这里 insert() 方法返回的是迭代器，而不再是 pair 对象：

（1）如果插入成功，insert() 方法会返回一个指向 map 容器中已插入键值对的迭代器；

（2）如果插入失败，insert() 方法同样会返回一个迭代器，该迭代器指向 map 容器中和 val 具有相同键的那个键值对。  
举个例子：

#include <iostream>

#include <map> //map

#include <string> //string

using namespace *std*;

int *main*()

{

//创建一个空 map 容器

*std*::*map*<*string*, *string*> mymap;

//创建一个真实存在的键值对变量

*std*::*pair*<*string*, *string*> STL = { "STL教程","http://c.biancheng.net/stl/" };

//指定要插入的位置

*std*::*map*<*string*, *string*>::*iterator* it = mymap.*begin*();

//向 it 位置以普通引用的方式插入 STL

auto iter1 = mymap.*insert*(it, STL);

*cout* << iter1->*first* << " " << iter1->*second* << *endl*;

//向 it 位置以右值引用的方式插入临时键值对

auto iter2 = mymap.*insert*(it, *std*::*pair*<*string*, *string*>("C语言教程", "http://c.biancheng.net/c/"));

*cout* << iter2->*first* << " " << iter2->*second* << *endl*;

//插入失败样例

auto iter3 = mymap.*insert*(it, *std*::*pair*<*string*, *string*>("STL教程", "http://c.biancheng.net/java/"));

*cout* << iter3->*first* << " " << iter3->*second* << *endl*;

return 0;

}

**程序执行结果为：**

**STL教程 http://c.biancheng.net/stl/  
C语言教程 http://c.biancheng.net/c/  
STL教程 http://c.biancheng.net/stl/**

再次强调，即便指定了新键值对的插入位置，map 容器仍会对存储的键值对进行排序。也可以说，决定新插入键值对位于 map 容器中位置的，不是 insert() 方法中传入的迭代器，而是新键值对中键的值。  
3) insert() 方法还支持向当前 map 容器中插入其它 map 容器指定区域内的所有键值对，该方法的语法格式如下：

**template <class InputIterator>  
  void insert (InputIterator first, InputIterator last);**

其中 first 和 last 都是迭代器，它们的组合<first,last>可以表示某 map 容器中的指定区域。

举个例子：

#include <iostream>

#include <map> //map

#include <string> //string

using namespace *std*;

int *main*()

{

//创建并初始化 map 容器

*std*::*map*<*std*::*string*, *std*::*string*>mymap{ {"STL教程","http://c.biancheng.net/stl/"},

{"C语言教程","http://c.biancheng.net/c/"},

{"Java教程","http://c.biancheng.net/java/"} };

//创建一个空 map 容器

*std*::*map*<*std*::*string*, *std*::*string*>copymap;

//指定插入区域

*std*::*map*<*string*, *string*>::*iterator* first = ++mymap.*begin*();

*std*::*map*<*string*, *string*>::*iterator* last = mymap.*end*();

//将<first,last>区域内的键值对插入到 copymap 中

copymap.*insert*(first, last);

//遍历输出 copymap 容器中的键值对

for (auto iter = copymap.*begin*(); iter != copymap.*end*(); ++iter) {

*cout* << iter->first << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**Java教程 http://c.biancheng.net/java/  
STL教程 http://c.biancheng.net/stl/**

此程序中，<first,last> 指定的区域是从 mumap 容器第 2 个键值对开始，之后所有的键值对，所以 copymap 容器中包含有 2 个键值对。

4) 除了以上一种格式外，insert() 方法还允许一次向 map 容器中插入多个键值对，其语法格式为：

void insert ({val1, val2, ...});

其中，vali 都表示的是键值对变量。  
举个例子：

#include <iostream>

#include <map> //map

#include <string> //string

using namespace *std*;

int *main*()

{

//创建空的 map 容器

*std*::*map*<*std*::*string*, *std*::*string*>mymap;

//向 mymap 容器中添加 3 个键值对

mymap.*insert*({ {"STL教程", "http://c.biancheng.net/stl/"},

{ "C语言教程","http://c.biancheng.net/c/" },

{ "Java教程","http://c.biancheng.net/java/" } });

for (auto iter = mymap.*begin*(); iter != mymap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**C语言教程 http://c.biancheng.net/c/  
Java教程 http://c.biancheng.net/java/  
STL教程 http://c.biancheng.net/stl/**

**值得一提的是，除了 insert() 方法，map 类模板还提供 emplace() 和 emplace\_hint() 方法，它们也可以完成向 map 容器中插入键值对的操作，且效率还会 insert() 方法高。关于这 2 个方法，会在下一节做详细介绍。**

**第七 C++ map容器operator[]和insert()效率对比（深度剖析）**

通过前面的学习我们知道，map 容器模板类中提供有 operator[ ] 和 insert() 这 2 个成员方法，而值得一提的是，这 2 个方法具有相同的功能，它们既可以实现向 map 容器中添加新的键值对元素，也可以实现更新（修改）map 容器已存储键值对的值。

举个例子（程序一）：

#include <iostream>

#include <map> //map

#include <string> //string

using namespace *std*;

int *main*()

{

*std*::*map*<*string*, *string*> mymap;

//借用 operator[] 添加新键值对

mymap["STL教程"] = "http://c.biancheng.net/java/";

*cout* << "old mymap：" << mymap["STL教程"] << *endl*;

//借用 operator[] 更新某个键对应的值

mymap["STL教程"] = "http://c.biancheng.net/stl/";

*cout* << "new mymap：" << mymap["STL教程"] << *endl*;

//借用insert()添加新键值对

*std*::*pair*<*string*, *string*> STL = { "Java教程","http://c.biancheng.net/python/" };

*std*::*pair*<*std*::*map*<*string*, *string*>::*iterator*, bool> ret;

ret = mymap.*insert*(STL);

*cout* << "old ret.iter = <{" << ret.*first*->*first* << ", " << ret.*first*->*second* << "}, " << ret.*second* << ">" << *endl*;

//借用 insert() 更新键值对

mymap.*insert*(STL).*first*->*second* = "http://c.biancheng.net/java/";

*cout* << "new ret.iter = <" << ret.*first*->*first* << ", " << ret.*first*->*second* << ">" << *endl*;

return 0;

}

**程序执行结果为：**

**old mymap：http://c.biancheng.net/java/  
new mymap：http://c.biancheng.net/stl/  
old ret.iter = <{Java教程, http://c.biancheng.net/python/}, 1>  
new ret.iter = <Java教程, http://c.biancheng.net/java/>**

有关程序中 operator[ ] 和 insert() 成员方法的具体用法，读者可翻阅前面的文章做详细了解，这里不再做过多解释。

显然，map 模板类中 operator[ ] 和 insert() 的功能发生了重叠，这就产生了一个问题，谁的执行效率更高呢？  
 总的来说，读者可记住这样一条结论：当实现“向 map 容器中添加新键值对元素”的操作时，insert() 成员方法的执行效率更高；而在实现“更新 map 容器指定键值对的值”的操作时，operator[ ] 的效率更高。  
  
至于为什么，有兴趣的读者可继续往下阅读。

**一 向map容器中增添元素，insert()效率更高**

首先解释一下，为什么实现向 map 容器中添加新键值对元素，insert() 方法的执行效率比 operator[ ] 更高？回顾程序一中，如下语句完成了向空 mymap 容器添加新的键值对元素：

**mymap["STL教程"] = "http://c.biancheng.net/java/";**

此行代码中，mymap["STL教程"] 实际上是 mymap.operator[ ](“STL教程”) 的缩写（底层调用的 operator[ ] 方法），该方法会返回一个指向 “STL教程” 对应的 value 值的引用。  
 但需要注意的是，由于此时 mymap 容器是空的，并没有 "STL教程" 对应的 value 值。这种情况下，operator[ ] 方法会默认构造一个 string 对象，并将其作为 "STL教程" 对应的 value 值，然后返回一个指向此 string 对象的引用。在此基础上，代码还会将 "http://c.biancheng.net.java/" 赋值给这个 string 对象。  
 也就是说，上面这行代码的执行流程，可以等效为如下程序：

**typedef map<string, string> mstr;**

**//创建要添加的默认键值对元素**

**pair<mstr::iterator, bool>res = mymap.insert(mstr::value\_type("STL教程", string()));**

**//将新键值对的值赋值为指定的值**

**res.first->second = "http://c.biancheng.net/java/";**

**注意，这里的 value\_type(K,T) 指的是 map 容器中存储元素的类型，其实际上就等同于 pair<K,T>。**

可以看到，使用 operator[ ] 添加新键值对元素的流程是，先构造一个有默认值的键值对，然后再为其 value 赋值。  
 那么，为什么不直接构造一个要添加的键值对元素呢，比如：

**mymap.insert(mstr::value\_type("STL教程", "http://c.biancheng.net/java/"));**

此行代码和上面程序的执行效果完全相同，但它省略了创建临时 string 对象的过程以及析构该对象的过程，同时还省略了调用 string 类重载的赋值运算符。由于可见，同样是完成向 map 容器添加新键值对，insert() 方法比 operator[ ] 的执行效率更高。

**二 更新map容器中的键值对，operator[]效率更高**

仍以程序一中的代码为例，如下分别是 operator[ ] 和 insert() 实现更新 mymap 容器中指定键对应的值的代码：

**//operator[]**

**mymap["STL教程"] = "http://c.biancheng.net/stl/";**

**//insert()**

**std::pair<string, string> STL = { "Java教程","http://c.biancheng.net/python/" };**

**mymap.insert(STL).first->second = "http://c.biancheng.net/java/";**

仅仅从语法形式本身来考虑，或许已经促使很多读者选择 operator[ ] 了。接下来，我们再从执行效率的角度对比以上 2 种实现方式。  
 从上面代码可以看到，insert() 方法在进行更新操作之前，需要有一个 pair 类型（也就是 map::value\_type 类型）元素做参数。这意味着，该方法要多构造一个 pair 对象（附带要构造 2 个 string 对象），并且事后还要析构此 pair 对象（附带 2 个 string 对象的析构）。  
 而和 insert() 方法相比，operator[ ] 就不需要使用 pair 对象，自然不需要构造（并析构）任何 pair 对象或者 string 对象。因此，对于更新已经存储在 map 容器中键值对的值，应优先使用 operator[ ] 方法。

**第八 C++ STL map emplace()和emplace\_hint()**

学习 map insert() 方法时提到，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) map 类模板中还提供了 emplace() 和 emplace\_hint() 成员函数，也可以实现向 map 容器中插入新的键值对。本节就来讲解这 2 个成员方法的用法。

**值得一提的是，实现相同的插入操作，无论是用 emplace() 还是 emplace\_hont()，都比 insert() 方法的效率高（后续章节会详细讲解）。**

和 insert() 方法相比，emplace() 和 emplace\_hint() 方法的使用要简单很多，因为它们各自只有一种语法格式。其中，emplace() 方法的语法格式如下：

**template <class... Args>  
  pair<iterator,bool> emplace (Args&&... args);**

参数 (Args&&... args) 指的是，这里只需要将创建新键值对所需的数据作为参数直接传入即可，此方法可以自行利用这些数据构建出指定的键值对。另外，该方法的返回值也是一个 pair 对象，其中 pair.first 为一个迭代器，pair.second 为一个 bool 类型变量：

（1）当该方法将键值对成功插入到 map 容器中时，其返回的迭代器指向该新插入的键值对，同时 bool 变量的值为 true；

（2）当插入失败时，则表明 map 容器中存在具有相同键的键值对，此时返回的迭代器指向此具有相同键的键值对，同时 bool 变量的值为 false。  
下面程序演示 emplace() 方法的具体用法：

#include <iostream>

#include <map> //map

#include <string> //string

using namespace *std*;

int *main*()

{

//创建并初始化 map 容器

*std*::*map*<*string*, *string*>mymap;

//插入键值对

*pair*<*map*<*string*, *string*>::*iterator*, bool> ret = mymap.*emplace*("STL教程", "http://c.biancheng.net/stl/");

*cout* << "1、ret.iter = <{" << ret.*first*->*first* << ", " << ret.*first*->*second* << "}, " << ret.*second* << ">" << *endl*;

//插入新键值对

ret = mymap.*emplace*("C语言教程", "http://c.biancheng.net/c/");

*cout* << "2、ret.iter = <{" << ret.*first*->*first* << ", " << ret.*first*->*second* << "}, " << ret.*second* << ">" << *endl*;

//失败插入的样例

ret = mymap.*emplace*("STL教程", "http://c.biancheng.net/java/");

*cout* << "3、ret.iter = <{" << ret.*first*->*first* << ", " << ret.*first*->*second* << "}, " << ret.*second* << ">" << *endl*;

return 0;

}

**程序执行结果为：**

**1、ret.iter = <{STL教程, http://c.biancheng.net/stl/}, 1>  
2、ret.iter = <{C语言教程, http://c.biancheng.net/c/}, 1>  
3、ret.iter = <{STL教程, http://c.biancheng.net/stl/}, 0>**

可以看到，程序中共执行了 3 次向 map 容器插入键值对的操作，其中前 2 次都成功了，第 3 次由于要插入的键值对的键和 map 容器中已存在的键值对的键相同，因此插入失败。  
  
emplace\_hint() 方法的功能和 emplace() 类似，其语法格式如下：

**template <class... Args>  
  iterator emplace\_hint (const\_iterator position, Args&&... args);**

显然和 emplace() 语法格式相比，有以下 2 点不同：

（1）该方法不仅要传入创建键值对所需要的数据，还需要传入一个迭代器作为第一个参数，指明要插入的位置（新键值对键会插入到该迭代器指向的键值对的前面）；

（2）该方法的返回值是一个迭代器，而不再是 pair 对象。当成功插入新键值对时，返回的迭代器指向新插入的键值对；反之，如果插入失败，则表明 map 容器中存有相同键的键值对，返回的迭代器就指向这个键值对。  
下面程序演示 emplace\_hint() 方法的用法：

#include <iostream>

#include <map> //map

#include <string> //string

using namespace *std*;

int *main*()

{

//创建并初始化 map 容器

*std*::*map*<*string*, *string*>mymap;

//指定在 map 容器插入键值对

*map*<*string*, *string*>::*iterator* iter = mymap.*emplace\_hint*(mymap.*begin*(), "STL教程", "http://c.biancheng.net/stl/");

*cout* << iter->*first* << " " << iter->*second* << *endl*;

iter = mymap.*emplace\_hint*(mymap.*begin*(), "C语言教程", "http://c.biancheng.net/c/");

*cout* << iter->*first* << " " << iter->*second* << *endl*;

//插入失败样例

iter = mymap.*emplace\_hint*(mymap.*begin*(), "STL教程", "http://c.biancheng.net/java/");

*cout* << iter->*first* << " " << iter->*second* << *endl*;

return 0;

}

**程序执行结果为：**

**STL教程 http://c.biancheng.net/stl/  
C语言教程 http://c.biancheng.net/c/  
STL教程 http://c.biancheng.net/stl/**

**注意，和 insert() 方法一样，虽然 emplace\_hint() 方法指定了插入键值对的位置，但 map 容器为了保持存储键值对的有序状态，可能会移动其位置。**

**那么，为什么 emplace() 和 emplace\_hint() 方法的执行效率，比 insert() 高呢？下一节会做详细解释。**

**第九 C++ map容器3种插入键值对的方法，谁的效率更高？**

上一节在学习 C++STL map 容器的 emplace() 和 emplace\_hint() 的基本用法时，还遗留了一个问题，即为什么 emplace() 和 emplace\_hint() 的执行效率会比 insert() 高？  
 原因很简单，它们向 map 容器插入键值对时，底层的实现方式不同：

使用 insert() 向 map 容器中插入键值对的过程是，先创建该键值对，然后再将该键值对复制或者移动到 map 容器中的指定位置；

使用 emplace() 或 emplace\_hint() 插入键值对的过程是，直接在 map 容器中的指定位置构造该键值对。  
 也就是说，向 map 容器中插入键值对时，emplace() 和 emplace\_hint() 方法都省略了移动键值对的过程，因此执行效率更高。下面程序提供了有利的证明：

#include <iostream>

#include <map> //map

#include <string> //string

using namespace *std*;

class testDemo

{

public:

testDemo(int num) :num(num) {

*std*::*cout* << "调用构造函数" << *endl*;

}

testDemo(const testDemo& other) :num(other.num) {

*std*::*cout* << "调用拷贝构造函数" << *endl*;

}

testDemo(testDemo&& other) :num(other.num) {

*std*::*cout* << "调用移动构造函数" << *endl*;

}

private:

int num;

};

int *main*()

{

//创建空 map 容器

*std*::*map*<*std*::*string*, testDemo>mymap;

*cout* << "insert():" << *endl*;

mymap.*insert*({ "http://c.biancheng.net/stl/", testDemo(1) });

*cout* << "emplace():" << *endl*;

mymap.*emplace*("http://c.biancheng.net/stl/:", 1);

*cout* << "emplace\_hint():" << *endl*;

mymap.*emplace\_hint*(mymap.*begin*(), "http://c.biancheng.net/stl/", 1);

return 0;

}

**程序输出结果为：**

**insert():  
调用构造函数  
调用移动构造函数  
调用移动构造函数  
emplace():  
调用构造函数  
emplace\_hint():  
调用构造函数**

分析一下这个程序。首先，我们创建了一个存储 <string,tempDemo> 类型键值对的空 map 容器，接下来分别用 insert()、emplace() 和 emplace\_hint() 方法向该 map 容器中插入相同的键值对。  
 从输出结果可以看出，在使用 insert() 方法向 map 容器插入键值对时，整个插入过程调用了 1 次 tempDemo 类的构造函数，同时还调用了 2 次移动构造函数。实际上，程序第 28 行代码底层的执行过程，可以分解为以下 3 步：

**//构造类对象**

**testDemo val = testDemo(1); //调用 1 次构造函数**

**//构造键值对**

**auto pai = make\_pair("http://c.biancheng.net/stl/", val); //调用 1 次移动构造函数**

**//完成插入操作**

**mymap.insert(pai); //调用 1 次移动构造函数**

而完成同样的插入操作，emplace() 和 emplace\_hint() 方法都只调用了 1 次构造函数，这足以证明，这 2 个方法是在 map 容器内部直接构造的键值对。

因此，在实现向 map 容器中插入键值对时，应优先考虑使用 emplace() 或者 emplace\_hint()。

**第十 C++ STL multimap容器用法完全攻略**

在掌握 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) map 容器的基础上，本节再讲一个和 map 相似的关联式容器，即 multimap 容器。  
 所谓“相似”，指的是 multimap 容器具有和 map 相同的特性，即 multimap 容器也用于存储 pair<const K, T> 类型的键值对（其中 K 表示键的类型，T 表示值的类型），其中各个键值对的键的值不能做修改；并且，该容器也会自行根据键的大小对存储的所有键值对做排序操作。和 map 容器的区别在于，multimap 容器中可以同时存储多（≥2）个键相同的键值对。  
 和 map 容器一样，实现 multimap 容器的类模板也定义在<map>头文件，并位于 std 命名空间中。因此，在使用 multimap 容器前，程序应包含如下代码：

#include <map>  
u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;

注意，第二行代码不是必需的，但若不用，则程序中在使用 multimap 容器时需手动注明 std 命名空间（强烈建议初学者使用）。

multimap 容器类模板的定义如下：

**template < class Key, // 指定键（key）的类型**

**class T, // 指定值（value）的类型**

**class Compare = less<Key>, // 指定排序规则**

**class Alloc = allocator<pair<const Key,T> > // 指定分配器对象的类型**

**> class multimap;**

可以看到，multimap 容器模板有 4 个参数，其中后 2 个参数都设有默认值。

大多数场景中，我们只需要设定前 2 个参数的值，有些场景可能会用到第 3 个参数，但最后一个参数几乎不会用到。

**一 创建C++ multimap容器的方法**

multimap 类模板内部提供有多个构造函数，总的来说，创建 multimap 容器的方式可归为以下 5 种。  
1) 通过调用 multimap 类模板的默认构造函数，可以创建一个空的 multimap 容器：

std::multimap<std::string, std::string>mymultimap;

如果程序中已经默认指定了 std 命令空间，这里可以省略 std::。  
2) 当然，在创建 multimap 容器的同时，还可以进行初始化操作。比如：

//创建并初始化 multimap 容器

**multimap<string, string>mymultimap{ {"C语言教程", "http://c.biancheng.net/c/"},**

**{"[Python](http://c.biancheng.net/python/" \t "http://c.biancheng.net/view/_blank)教程", "http://c.biancheng.net/python/"},**

**{"STL教程", "http://c.biancheng.net/stl/"} };**

注意，使用此方式初始化 multimap 容器时，其底层会先将每一个{key, value}创建成 pair 类型的键值对，然后再用已建好的各个键值对初始化 multimap 容器。  
 实际上，我们完全可以先手动创建好键值对，然后再用其初始化 multimap 容器。下面程序使用了 2 种方式创建 pair 类型键值对，再用其初始化 multimap 容器，它们是完全等价的：

**//借助 pair 类模板的构造函数来生成各个pair类型的键值对**

**multimap<string, string>mymultimap{**

**pair<string,string>{"C语言教程", "http://c.biancheng.net/c/"},**

**pair<string,string>{ "Python教程", "http://c.biancheng.net/python/"},**

**pair<string,string>{ "STL教程", "http://c.biancheng.net/stl/"}**

**};**

**//调用 make\_pair() 函数，生成键值对元素**

**//创建并初始化 multimap 容器**

**multimap<string, string>mymultimap{**

**make\_pair("C语言教程", "http://c.biancheng.net/c/"),**

**make\_pair("Python教程", "http://c.biancheng.net/python/"),**

**make\_pair("STL教程", "http://c.biancheng.net/stl/")**

**};**  
3) 除此之外，通过调用 multimap 类模板的拷贝（复制）构造函数，也可以初始化新的 multimap 容器。例如：

**multimap<string, string>newmultimap(mymultimap);**

由此，就成功创建一个和 mymultimap 完全一样的 newmultimap 容器。  
在 C++ 11 标准中，还为 multimap 类增添了移动构造函数。即当有临时的 multimap 容器作为参数初始化新 multimap 容器时，其底层就会调用移动构造函数来实现初始化操作。举个例子：

**//创建一个会返回临时 multimap 对象的函数**

**multimap<string, string> dismultimap() {**

**multimap<string, string>tempmultimap{ {"C语言教程", "http://c.biancheng.net/c/"},{"Python教程", "http://c.biancheng.net/python/"} };**

**return tempmultimap;**

**}**

**//调用 multimap 类模板的移动构造函数创建 newMultimap 容器**

**multimap<string, string>newmultimap(dismultimap());**

上面程序中，由于 dismultimap() 函数返回的 tempmultimap 容器是一个临时对象，因此在实现初始化 newmultimap 容器时，底层调用的是 multimap 容器的移动构造函数，而不再是拷贝构造函数。

注意，无论是调用复制构造函数还是调用拷贝构造函数，都必须保证这 2 个容器的类型完全一致。

4) multimap 类模板还支持从已有 multimap 容器中，选定某块区域内的所有键值对，用作初始化新 multimap 容器时使用。例如：

**//创建并初始化 multimap 容器**

**multimap<string, string>mymultimap{ {"C语言教程", "http://c.biancheng.net/c/"},**

**{"Python教程", "http://c.biancheng.net/python/"},**

**{"STL教程", "http://c.biancheng.net/stl/"} };**

**multimap<string, string>newmultimap(++mymultimap.begin(), mymultimap.end());**

这里使用了 multimap 容器的迭代器，选取了 mymultimap 容器中的最后 2 个键值对，用于初始化 newmultimap 容器。

multimap 容器迭代器，和 map 容器迭代器的用法完全相同，这里不再赘述。  
5) 前面讲到，multimap 类模板共可以接收 4 个参数，其中第 3 个参数可用来修改 multimap 容器内部的排序规则。默认情况下，此参数的值为std::less<T>，这意味着以下 2 种创建 multimap 容器的方式是等价的：

**multimap<char, int>mymultimap{ {'a',1},{'b',2} };**

**multimap<char, int, std::less<char>>mymultimap{ {'a',1},{'b',2} };**

mymultimap 容器中键值对的存储顺序为：

<a,1>  
<b,2>

下面程序利用了 STL 模板库提供的std::greater<T>排序函数，实现令 multimap 容器对存储的键值对做降序排序：

**multimap<char, int, std::greater<char>>mymultimap{ {'a',1},{'b',2} };**

其内部键值对的存储顺序为：

**<b,2>  
<a,1>**

**在某些特定场景中，我们还可以为 multimap 容器自定义排序规则，此部分知识后续将利用整整一节做重点讲解。**

**二 C++ multimap容器包含的成员方法**

表 1 列出了 multimap 类模板提供的常用成员方法及各自的功能。

表 1 C++ multimap 容器常用成员方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| begin() | 返回指向容器中第一个（注意，是已排好序的第一个）键值对的双向迭代器。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| end() | 返回指向容器最后一个元素（注意，是已排好序的最后一个）所在位置后一个位置的双向迭代器，通常和 begin() 结合使用。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| rbegin() | 返回指向最后一个（注意，是已排好序的最后一个）元素的反向双向迭代器。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| rend() | 返回指向第一个（注意，是已排好序的第一个）元素所在位置前一个位置的反向双向迭代器。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的键值对。 |
| find(key) | 在 multimap 容器中查找首个键为 key 的键值对，如果成功找到，则返回指向该键值对的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| lower\_bound(key) | 返回一个指向当前 multimap 容器中第一个大于或等于 key 的键值对的双向迭代器。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| upper\_bound(key) | 返回一个指向当前 multimap 容器中第一个大于 key 的键值对的迭代器。如果 multimap 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| equal\_range(key) | 该方法返回一个 pair 对象（包含 2 个双向迭代器），其中 pair.first 和 lower\_bound() 方法的返回值等价，pair.second 和 upper\_bound() 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含的键为 key 的键值对。 |
| empty() | 若容器为空，则返回 true；否则 false。 |
| size() | 返回当前 multimap 容器中存有键值对的个数。 |
| max\_size() | 返回 multimap 容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。 |
| insert() | 向 multimap 容器中插入键值对。 |
| erase() | 删除 multimap 容器指定位置、指定键（key）值或者指定区域内的键值对。 |
| swap() | 交换 2 个 multimap 容器中存储的键值对，这意味着，操作的 2 个键值对的类型必须相同。 |
| clear() | 清空 multimap 容器中所有的键值对，使 multimap 容器的 size() 为 0。 |
| emplace() | 在当前 multimap 容器中的指定位置处构造新键值对。其效果和插入键值对一样，但效率更高。 |
| emplace\_hint() | 在本质上和 emplace() 在 multimap 容器中构造新键值对的方式是一样的，不同之处在于，使用者必须为该方法提供一个指示键值对生成位置的迭代器，并作为该方法的第一个参数。 |
| count(key) | 在当前 multimap 容器中，查找键为 key 的键值对的个数并返回。 |

**和 map 容器相比，multimap 未提供 at() 成员方法，也没有重载 [] 运算符。这意味着，map 容器中通过指定键获取指定指定键值对的方式，将不再适用于 multimap 容器。其实这很好理解，因为 multimap 容器中指定的键可能对应多个键值对，而不再是 1 个。**

另外值的一提的是，由于 multimap 容器可存储多个具有相同键的键值对，因此表 1 中的 lower\_bound()、upper\_bound()、equal\_range() 以及 count() 成员方法会经常用到。

下面例子演示了表 1 中部分成员方法的用法：

#include <iostream>

#include <map> //map

using namespace *std*;

int *main*()

{

//创建并初始化 multimap 容器

*multimap*<char, int>mymultimap{ {'a',10},{'b',20},{'b',15}, {'c',30} };

//输出 mymultimap 容器存储键值对的数量

*cout* << mymultimap.*size*() << *endl*;

//输出 mymultimap 容器中存储键为 'b' 的键值对的数量

*cout* << mymultimap.*count*('b') << *endl*;

for (auto iter = mymultimap.*begin*(); iter != mymultimap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**4  
2  
a 10  
b 20  
b 15  
c 30**

注意，只要是 multimap 容器提供的成员方法，map 容器都提供，并且它们的用法是相同的。前面章节中已经对 map 容器提供的成员方法做了详细的讲解，因此这里不再对表 1 中其它的成员方法做详细的介绍。

**第十一 C++ STL set容器完全攻略**

前面章节讲解了 map 容器和 multimap 容器的用法，类似地，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 标准库中还提供有 set 和 multiset 这 2 个容器，它们也属于关联式容器。不过，本节先讲解 set 容器，后续章节再讲解 multiset 容器。  
 和 map、multimap 容器不同，使用 set 容器存储的各个键值对，要求键 key 和值 value 必须相等。  
举个例子，如下有 2 组键值对数据：

**{<'a', 1>, <'b', 2>, <'c', 3>}  
{<'a', 'a'>, <'b', 'b'>, <'c', 'c'>}**

显然，第一组数据中各键值对的键和值不相等，而第二组中各键值对的键和值对应相等。对于 set 容器来说，只能存储第 2 组键值对，而无法存储第一组键值对。  
 基于 set 容器的这种特性，当使用 set 容器存储键值对时，只需要为其提供各键值对中的 value 值（也就是 key 的值）即可。仍以存储上面第 2 组键值对为例，只需要为 set 容器提供 {'a','b','c'} ，该容器即可成功将它们存储起来。  
 通过前面的学习我们知道，map、multimap 容器都会自行根据键的大小对存储的键值对进行排序，set 容器也会如此，只不过 set 容器中各键值对的键 key 和值 value 是相等的，根据 key 排序，也就等价为根据 value 排序。  
 另外，使用 set 容器存储的各个元素的值必须各不相同。更重要的是，从语法上讲 set 容器并没有强制对存储元素的类型做 const 修饰，即 set 容器中存储的元素的值是可以修改的。但是，C++ 标准为了防止用户修改容器中元素的值，对所有可能会实现此操作的行为做了限制，使得在正常情况下，用户是无法做到修改 set 容器中元素的值的。

**对于初学者来说，切勿尝试直接修改 set 容器中已存储元素的值，这很有可能破坏 set 容器中元素的有序性，最正确的修改 set 容器中元素值的做法是：先删除该元素，然后再添加一个修改后的元素。**

值得一提的是，set 容器定义于<set>头文件，并位于 std 命名空间中。因此如果想在程序中使用 set 容器，该程序代码应先包含如下语句：

**#include <set>**

**u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;**

**注意，第二行代码不是必需的，如果不用，则后续程序中在使用 set 容器时，需手动注明 std 命名空间（强烈建议初学者使用）。**  
  
**set 容器的类模板定义如下：**

**template < class T, // 键 key 和值 value 的类型**

**class Compare = less<T>, // 指定 set 容器内部的排序规则**

**class Alloc = allocator<T> // 指定分配器对象的类型**

**> class set;**

**注意，由于 set 容器存储的各个键值对，其键和值完全相同，也就意味着它们的类型相同，因此 set 容器类模板的定义中，仅有第 1 个参数用于设定存储数据的类型。**

**对于 set 类模板中的 3 个参数，后 2 个参数自带默认值，且几乎所有场景中只需使用前 2 个参数，第 3 个参数不会用到。**

**一 创建C++ set容器的几种方法**

常见的创建 set 容器的方法，大致有以下 5 种。  
  
1) 调用默认构造函数，创建空的 set 容器。比如：

**std::set<std::string> myset;**

如果程序中已经默认指定了 std 命令空间，这里可以省略 std::。

由此就创建好了一个 set 容器，该容器采用默认的std::less<T>规则，会对存储的 string 类型元素做升序排序。注意，由于 set 容器支持随时向内部添加新的元素，因此创建空 set 容器的方法是经常使用的。  
  
2) 除此之外，set 类模板还支持在创建 set 容器的同时，对其进行初始化。例如：

**std::set<std::string> myset{"http://c.biancheng.net/java/",**

**"http://c.biancheng.net/stl/",**

**"http://c.biancheng.net/python/"};**

由此即创建好了包含 3 个 string 元素的 myset 容器。由于其采用默认的 std::less<T> 规则，因此其内部存储 string 元素的顺序如下所示：

**"http://c.biancheng.net/java/"  
"http://c.biancheng.net/python/"  
"http://c.biancheng.net/stl/"**

3) set 类模板中还提供了拷贝（复制）构造函数，可以实现在创建新 set 容器的同时，将已有 set 容器中存储的所有元素全部复制到新 set 容器中。  
  
例如，在第 2 种方式创建的 myset 容器的基础上，执行如下代码：

**std::set<std::string> copyset(myset);**

**//等同于**

**//std::set<std::string> copyset = myset**

**该行代码在创建 copyset 容器的基础上，还会将 myset 容器中存储的所有元素，全部复制给 copyset 容器一份。**  
**另外，C++ 11 标准还为 set 类模板新增了移动构造函数，其功能是实现创建新 set 容器的同时，利用临时的 set 容器为其初始化。比如：**

**set<string> retSet() {**

**std::set<std::string> myset{ "http://c.biancheng.net/java/",**

**"http://c.biancheng.net/stl/",**

**"http://c.biancheng.net/python/" };**

**return myset;**

**}**

**std::set<std::string> copyset(retSet());**

**//或者**

**//std::set<std::string> copyset = retSet();**

**注意，由于 retSet() 函数的返回值是一个临时 set 容器，因此在初始化 copyset 容器时，其内部调用的是 set 类模板中的移动构造函数，而非拷贝构造函数。**

显然，无论是调用复制构造函数还是调用拷贝构造函数，都必须保证这 2 个容器的类型完全一致。

4) 在第 3 种方式的基础上，set 类模板还支持取已有 set 容器中的部分元素，来初始化新 set 容器。例如：

s**td::set<std::string> myset{ "http://c.biancheng.net/java/",**

**"http://c.biancheng.net/stl/",**

**"http://c.biancheng.net/python/" };**

**std::set<std::string> copyset(++myset.begin(), myset.end());**

**由此初始化的 copyset 容器，其内部仅存有如下 2 个 string 字符串：**

**"http://c.biancheng.net/python/"  
"http://c.biancheng.net/stl/"**

5) 以上几种方式创建的 set 容器，都采用了默认的std::less<T>规则。其实，借助 set 类模板定义中第 2 个参数，我们完全可以手动修改 set 容器中的排序规则。比如：

**std::set<std::string,std::greater<string> > myset{**

**"http://c.biancheng.net/java/",**

**"http://c.biancheng.net/stl/",**

**"http://c.biancheng.net/python/"};**

**通过选用 std::greater<string> 降序规则，myset 容器中元素的存储顺序为:**

**"http://c.biancheng.net/stl/"  
"http://c.biancheng.net/python/"  
"http://c.biancheng.net/java/"**

**二 C++ STL set容器包含的成员方法**

表 1 列出了 set 容器提供的常用成员方法以及各自的功能。

表 1 C++ set 容器常用成员方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| begin() | 返回指向容器中第一个（注意，是已排好序的第一个）元素的双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| end() | 返回指向容器最后一个元素（注意，是已排好序的最后一个）所在位置后一个位置的双向迭代器，通常和 begin() 结合使用。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| rbegin() | 返回指向最后一个（注意，是已排好序的最后一个）元素的反向双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| rend() | 返回指向第一个（注意，是已排好序的第一个）元素所在位置前一个位置的反向双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| find(val) | 在 set 容器中查找值为 val 的元素，如果成功找到，则返回指向该元素的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| lower\_bound(val) | 返回一个指向当前 set 容器中第一个大于或等于 val 的元素的双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| upper\_bound(val) | 返回一个指向当前 set 容器中第一个大于 val 的元素的迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| equal\_range(val) | 该方法返回一个 pair 对象（包含 2 个双向迭代器），其中 pair.first 和 lower\_bound() 方法的返回值等价，pair.second 和 upper\_bound() 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含的值为 val 的元素（set 容器中各个元素是唯一的，因此该范围最多包含一个元素）。 |
| empty() | 若容器为空，则返回 true；否则 false。 |
| size() | 返回当前 set 容器中存有元素的个数。 |
| max\_size() | 返回 set 容器所能容纳元素的最大个数，不同的操作系统，其返回值亦不相同。 |
| insert() | 向 set 容器中插入元素。 |
| erase() | 删除 set 容器中存储的元素。 |
| swap() | 交换 2 个 set 容器中存储的所有元素。这意味着，操作的 2 个 set 容器的类型必须相同。 |
| clear() | 清空 set 容器中所有的元素，即令 set 容器的 size() 为 0。 |
| emplace() | 在当前 set 容器中的指定位置直接构造新元素。其效果和 insert() 一样，但效率更高。 |
| emplace\_hint() | 在本质上和 emplace() 在 set 容器中构造新元素的方式是一样的，不同之处在于，使用者必须为该方法提供一个指示新元素生成位置的迭代器，并作为该方法的第一个参数。 |
| count(val) | 在当前 set 容器中，查找值为 val 的元素的个数，并返回。注意，由于 set 容器中各元素的值是唯一的，因此该函数的返回值最大为 1。 |

下面程序演示了表 1 中部分成员函数的用法：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建空set容器

*std*::*set*<*std*::*string*> myset;

//空set容器不存储任何元素

*cout* << "1、myset size = " << myset.*size*() << *endl*;

//向myset容器中插入新元素

myset.*insert*("http://c.biancheng.net/java/");

myset.*insert*("http://c.biancheng.net/stl/");

myset.*insert*("http://c.biancheng.net/python/");

*cout* << "2、myset size = " << myset.*size*() << *endl*;

//利用双向迭代器，遍历myset

for (auto iter = myset.*begin*(); iter != myset.*end*(); ++iter) {

*cout* << \*iter << *endl*;

}

return 0;

}

**程序执行结果为：**

**1、myset size = 0  
2、myset size = 3  
http://c.biancheng.net/java/  
http://c.biancheng.net/python/  
http://c.biancheng.net/stl/**

**有关表 1 中其它成员方法的用法，后续章节会做详细讲解。**

**第十二 C++ STL set容器迭代器用法详解**

和 map 容器不同，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 中的 set 容器类模板中未提供 at() 成员函数，也未对 [] 运算符进行重载。因此，要想访问 set 容器中存储的元素，只能借助 set 容器的迭代器。  
 值得一提的是，C++ STL 标准库为 set 容器配置的迭代器类型为双向迭代器。这意味着，假设 p 为此类型的迭代器，则其只能进行 ++p、p++、--p、p--、\*p 操作，并且 2 个双向迭代器之间做比较，也只能使用 == 或者 != 运算符。在 set 容器类模板提供的所有成员函数中，返回迭代器的成员函数如表 1 所示。

表 1 C++ set 容器迭代器方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| begin() | 返回指向容器中第一个（注意，是已排好序的第一个）元素的双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| end() | 返回指向容器最后一个元素（注意，是已排好序的最后一个）所在位置后一个位置的双向迭代器，通常和 begin() 结合使用。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| rbegin() | 返回指向最后一个（注意，是已排好序的最后一个）元素的反向双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| rend() | 返回指向第一个（注意，是已排好序的第一个）元素所在位置前一个位置的反向双向迭代器。通常和 rbegin() 结合使用。如果 set 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| find(val) | 在 set 容器中查找值为 val 的元素，如果成功找到，则返回指向该元素的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| lower\_bound(val) | 返回一个指向当前 set 容器中第一个大于或等于 val 的元素的双向迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| upper\_bound(val) | 返回一个指向当前 set 容器中第一个大于 val 的元素的迭代器。如果 set 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| equal\_range(val) | 该方法返回一个 pair 对象（包含 2 个双向迭代器），其中 pair.first 和 lower\_bound() 方法的返回值等价，pair.second 和 upper\_bound() 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含的值为 val 的元素（set 容器中各个元素是唯一的，因此该范围最多包含一个元素）。 |

**注意，以上成员函数返回的迭代器，指向的只是 set 容器中存储的元素，而不再是键值对。另外，以上成员方法返回的迭代器，无论是 const 类型还是非 const 类型，都不能用于修改 set 容器中的值。**

图 2 演示了表 1 中除最后 4 个成员函数外，其它几个成员函数的具体功能。

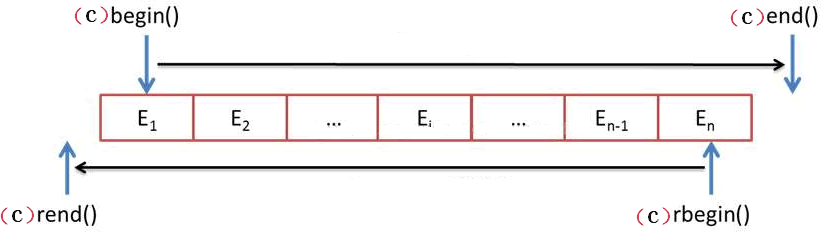


图 2 set容器迭代器功能示意图

**其中，Ei 表示 set 容器中存储的各个元素，它们的值各不相同。**

下面程序以 begin()/end() 为例，演示了如何使用图 2 中相关迭代器遍历 set 容器：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化set容器

*std*::*set*<*std*::*string*> myset{ "http://c.biancheng.net/java/",

"http://c.biancheng.net/stl/",

"http://c.biancheng.net/python/"

};

//利用双向迭代器，遍历myset

for (auto iter = myset.*begin*(); iter != myset.*end*(); ++iter) {

*cout* << \*iter << *endl*;

}

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/java/  
http://c.biancheng.net/python/  
http://c.biancheng.net/stl/**

**再次强调，正如程序第 15 行代码所示的那样，因为 iter 迭代器指向的是 set 容器存储的某个元素，而不是键值对，因此通过 \*iter 可以直接获取该迭代器指向的元素的值。**  
 除此之外，如果只想遍历 set 容器中指定区域内的部分数据，则可以借助 find()、lower\_bound() 以及 upper\_bound() 实现。通过调用它们，可以获取一个指向指定元素的迭代器。  
 **需要特别指出的是，equal\_range(val) 函数的返回值是一个 pair 类型数据，其包含 2 个迭代器，表示 set 容器中和指定参数 val 相等的元素所在的区域，但由于 set 容器中存储的元素各不相等，因此该函数返回的这 2 个迭代器所表示的范围中，最多只会包含 1 个元素。**

举个例子：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化set容器

*std*::*set*<*std*::*string*> myset{ "http://c.biancheng.net/java/",

"http://c.biancheng.net/stl/",

"http://c.biancheng.net/python/"

};

*set*<*string*>::*iterator* iter = myset.*find*("http://c.biancheng.net/python/");

for (; iter != myset.*end*(); ++iter)

{

*cout* << \*iter << *endl*;

}

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/python/  
http://c.biancheng.net/stl/**

**值得一提的是，虽然 C++ STL 标准中，set 类模板中包含 lower\_bound()、upper\_bound()、equal\_range() 这 3 个成员函数，但它们更适用于 multiset 容器，几乎不会用于操作 set 容器。**

**第十三 C++ STL set insert()方法详解**

通过前面的学习，我们已经学会如何创建一个 set 容器。在此基础上，如果想向 set 容器中继续添加元素，可以借助 set 类模板提供的 insert() 方法。  
 为满足不同场景的需要，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 标准的 set 类模板中提供了多种不同语法格式的 insert() 成员方法，它们各自的功能和用法如下所示。  
  
1) 只要给定目标元素的值，insert() 方法即可将该元素添加到 set 容器中，其语法格式如下：

**//普通引用方式传参  
pair<iterator,bool> insert (const value\_type& val);  
//右值引用方式传参  
pair<iterator,bool> insert (value\_type&& val);**

**其中，val 表示要添加的新元素，该方法的返回值为 pair 类型。**

**以上 2 种格式的区别仅在于传递参数的方式不同，即第一种采用普通引用的方式传参，而第二种采用右值引用的方式传参。右值引用为 C++ 11 新添加的一种引用方式，可阅读《[C++ 右值引用](http://c.biancheng.net/view/439.html" \t "http://c.biancheng.net/view/_blank)》一文做详细了解。**

可以看到，以上 2 种语法格式的 insert() 方法，返回的都是 pair 类型的值，其包含 2 个数据，一个迭代器和一个 bool 值：

**（1）当向 set 容器添加元素成功时，该迭代器指向 set 容器新添加的元素，bool 类型的值为 true；**

**（2）如果添加失败，即证明原 set 容器中已存有相同的元素，此时返回的迭代器就指向容器中相同的此元素，同时 bool 类型的值为 false。**

举个例子：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化set容器

*std*::*set*<*std*::*string*> myset;

//准备接受 insert() 的返回值

*pair*<*set*<*string*>::*iterator*, bool> retpair;

//采用普通引用传值方式

*string* str = "http://c.biancheng.net/stl/";

retpair = myset.*insert*(str);

*cout* << "iter->" << \*(retpair.*first*) << " " << "bool = " << retpair.*second* << *endl*;

//采用右值引用传值方式

retpair = myset.*insert*("http://c.biancheng.net/python/");

*cout* << "iter->" << \*(retpair.*first*) << " " << "bool = " << retpair.*second* << *endl*;

return 0;

}

**程序执行结果为：**

**iter->http://c.biancheng.net/stl/ bool = 1  
iter->http://c.biancheng.net/python/ bool = 1**

通过观察输出结果不难看出，程序中两次借助 insert() 方法向 set 容器中添加元素，都成功了。  
2) insert() 还可以指定将新元素插入到 set 容器中的具体位置，其语法格式如下：

**//以普通引用的方式传递 val 值  
iterator insert (const\_iterator position, const value\_type& val);  
//以右值引用的方式传递 val 值  
iterator insert (const\_iterator position, value\_type&& val);**

**以上 2 种语法格式中，insert() 函数的返回值为迭代器：**

**当向 set 容器添加元素成功时，该迭代器指向容器中新添加的元素；**

**当添加失败时，证明原 set 容器中已有相同的元素，该迭代器就指向 set 容器中相同的这个元素。**

举个例子：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化set容器

*std*::*set*<*std*::*string*> myset;

//准备接受 insert() 的返回值

*set*<*string*>::*iterator* iter;

//采用普通引用传值方式

*string* str = "http://c.biancheng.net/stl/";

iter = myset.*insert*(myset.*begin*(), str);

*cout* << "myset size =" << myset.*size*() << *endl*;

//采用右值引用传值方式

iter = myset.*insert*(myset.*end*(), "http://c.biancheng.net/python/");

*cout* << "myset size =" << myset.*size*() << *endl*;

return 0;

}

**程序执行结果为：**

**myset size =1  
myset size =2**

**注意，使用 insert() 方法将目标元素插入到 set 容器指定位置后，如果该元素破坏了容器内部的有序状态，set 容器还会自行对新元素的位置做进一步调整。也就是说，insert() 方法中指定新元素插入的位置，并不一定就是该元素最终所处的位置。**

3) insert() 方法支持向当前 set 容器中插入其它 set 容器指定区域内的所有元素，只要这 2 个 set 容器存储的元素类型相同即可。  
**insert() 方法的语法格式如下：**

**template <class InputIterator>  
  void insert (InputIterator first, InputIterator last);**

**其中 first 和 last 都是迭代器，它们的组合 [first,last) 可以表示另一 set 容器中的一块区域，该区域包括 first 迭代器指向的元素，但不包含 last 迭代器指向的元素。**  
举个例子：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化set容器

*std*::*set*<*std*::*string*> myset{ "http://c.biancheng.net/stl/",

"http://c.biancheng.net/python/",

"http://c.biancheng.net/java/" };

//创建一个同类型的空 set 容器

*std*::*set*<*std*::*string*> otherset;

//利用 myset 初始化 otherset

otherset.*insert*(++myset.*begin*(), myset.*end*());

//输出 otherset 容器中的元素

for (auto iter = otherset.*begin*(); iter != otherset.*end*(); ++iter) {

*cout* << \*iter << *endl*;

}

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/python/  
http://c.biancheng.net/stl/**

**注意，程序第 15 行在初始化 otherset 容器时，选取的是 myset 容器中从第 2 个元素开始（包括此元素）直到容器末尾范围内的所有元素，所以程序输出结果中只有 2 个字符串。**4) 采用如下格式的 insert() 方法，可实现一次向 set 容器中添加多个元素：

**void insert ( {E1, E2,...,En} );**

**其中，Ei 表示新添加的元素。**  
举个例子：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化set容器

*std*::*set*<*std*::*string*> myset;

//向 myset 中添加多个元素

myset.*insert*({ "http://c.biancheng.net/stl/",

"http://c.biancheng.net/python/",

"http://c.biancheng.net/java/" });

for (auto iter = myset.*begin*(); iter != myset.*end*(); ++iter) {

*cout* << \*iter << *endl*;

}

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/java/  
http://c.biancheng.net/python/  
http://c.biancheng.net/stl/**

以上的讲解，即为 set 类模板中 insert() 成员方法的全部用法。指的一提的是，C++ 11 标准的 set 类模板中，还提供有另外 2 个成员方法，分别为 implace() 和 implace\_hint() 方法，借助它们不但能实现向 set 容器添加新元素的功能，其实现效率也比 insert() 成员方法更高。

有关 set 类模板中 implace() 和 implace\_hint() 方法的用法，后续章节会做详细介绍。

**第十四 C++ STL set emplace()和emplace\_hint()**

要知道，set 类模板提供的所有成员方法中，能实现向指定 set 容器中添加新元素的，只有 3 个成员方法，分别为 insert()、emplace() 和 emplace\_hint()。其中 insert() 成员方法的用法已在前面章节做了详细的讲解，本节重点介绍剩下的这 2 个成员方法。  
 emplace() 和 emplace\_hint() 是 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 标准加入到 set 类模板中的，相比具有同样功能的 insert() 方法，完成同样的任务，emplace() 和 emplace\_hint() 的效率会更高。  
**emplace() 方法的语法格式如下：**

**template <class... Args>  
  pair<iterator,bool> emplace (Args&&... args);**

其中，参数 (Args&&... args) 指的是，只需要传入构建新元素所需的数据即可，该方法可以自行利用这些数据构建出要添加的元素。比如，若 set 容器中存储的元素类型为自定义的结构体或者类，则在使用 emplace() 方法向容器中添加新元素时，构造新结构体变量（或者类对象）需要多少个数据，就需要为该方法传入相应个数的数据。  
 另外，该方法的返回值类型为 pair 类型，其包含 2 个元素，一个迭代器和一个 bool 值：

（1）当该方法将目标元素成功添加到 set 容器中时，其返回的迭代器指向新插入的元素，同时 bool 值为 true；

（2）当添加失败时，则表明原 set 容器中已存在相同值的元素，此时返回的迭代器指向容器中具有相同键的这个元素，同时 bool 值为 false。

下面程序演示 emplace() 方法的具体用法：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化 set 容器

*std*::*set*<*string*>myset;

//向 myset 容器中添加元素

*pair*<*set*<*string*, *string*>::*iterator*, bool> ret = myset.*emplace*("http://c.biancheng.net/stl/");

*cout* << "myset size = " << myset.*size*() << *endl*;

*cout* << "ret.iter = <" << \*(ret.*first*) << ", " << ret.*second* << ">" << *endl*;

return 0;

}

**程序执行结果为：**

**myset size = 1  
ret.iter = <http://c.biancheng.net/stl/, 1>**

**显然，从执行结果可以看出，通过调用 emplace() 方法，成功向空 myset 容器中添加了一个元素，并且该方法的返回值中就包含指向新添加元素的迭代器。**  
emplace\_hint() 方法的功能和 emplace() 类似，其语法格式如下：

**template <class... Args>  
  iterator emplace\_hint (const\_iterator position, Args&&... args);**

和 emplace() 方法相比，有以下 2 点不同：

（1）该方法需要额外传入一个迭代器，用来指明新元素添加到 set 容器的具体位置（新元素会添加到该迭代器指向元素的前面）；

（2）返回值是一个迭代器，而不再是 pair 对象。当成功添加元素时，返回的迭代器指向新添加的元素；反之，如果添加失败，则迭代器就指向 set 容器和要添加元素的值相同的元素。

下面程序演示 emplace\_hint() 方法的用法：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化 set 容器

*std*::*set*<*string*>myset;

//在 set 容器的指定位置添加键值对

*set*<*string*>::*iterator* iter = myset.*emplace\_hint*(myset.*begin*(), "http://c.biancheng.net/stl/");

*cout* << "myset size = " << myset.*size*() << *endl*;

*cout* << \*iter << *endl*;

return 0;

}

**程序执行结果为：**

**myset size = 1  
http://c.biancheng.net/stl/**

注意，和 insert() 方法一样，虽然 emplace\_hint() 方法中指定了添加新元素的位置，但 set 容器为了保持数据的有序状态，可能会移动其位置。

**以上内容讲解了 emplace() 和 emplace\_hint() 的用法，至于比 insert() 执行效率高的原因，可参照 map 容器 emplace() 和 emplace\_hint() 比 insert() 效率高的原因，它们是完全一样的，这里不再赘述。**

**第十五C++ STL set删除数据：erase()和clear()方法**

如果想删除 set 容器存储的元素，可以选择用 erase() 或者 clear() 成员方法。  
set 类模板中，erase() 方法有 3 种语法格式，分别如下：

**//删除 set 容器中值为 val 的元素  
size\_type erase (const value\_type& val);  
//删除 position 迭代器指向的元素  
iterator  erase (const\_iterator position);  
//删除 [first,last) 区间内的所有元素  
iterator  erase (const\_iterator first, const\_iterator last);**

其中，第 1 种格式的 erase() 方法，其返回值为一个整数，表示成功删除的元素个数；后 2 种格式的 erase() 方法，返回值都是迭代器，其指向的是 set 容器中删除元素之后的第一个元素。

**注意，如果要删除的元素就是 set 容器最后一个元素，则 erase() 方法返回的迭代器就指向新 set 容器中最后一个元素之后的位置（等价于 end() 方法返回的迭代器）。**

下面程序演示了以上 3 种 erase() 方法的用法：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化 set 容器

*std*::*set*<int>myset{ 1,2,3,4,5 };

*cout* << "myset size = " << myset.*size*() << *endl*;

//1) 调用第一种格式的 erase() 方法

int num = myset.*erase*(2); //删除元素 2，myset={1,3,4,5}

*cout* << "1、myset size = " << myset.*size*() << *endl*;

*cout* << "num = " << num << *endl*;

//2) 调用第二种格式的 erase() 方法

*set*<int>::*iterator* iter = myset.*erase*(myset.*begin*()); //删除元素 1，myset={3,4,5}

*cout* << "2、myset size = " << myset.*size*() << *endl*;

*cout* << "iter->" << \*iter << *endl*;

//3) 调用第三种格式的 erase() 方法

*set*<int>::*iterator* iter2 = myset.*erase*(myset.*begin*(), --myset.*end*());//删除元素 3,4，myset={5}

*cout* << "3、myset size = " << myset.*size*() << *endl*;

*cout* << "iter2->" << \*iter2 << *endl*;

return 0;

}

**程序执行结果为：**

**myset size = 5  
1、myset size = 4  
num = 1  
2、myset size = 3  
iter->3  
3、myset size = 1  
iter2->5**

如果需要删除 set 容器中存储的所有元素，可以使用 clear() 成员方法。该方法的语法格式如下：

void clear();

显然，该方法不需要传入任何参数，也没有任何返回值。  
  
举个例子：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化 set 容器

*std*::*set*<int>myset{ 1,2,3,4,5 };

*cout* << "1、myset size = " << myset.*size*() << *endl*;

//清空 myset 容器

myset.*clear*();

*cout* << "2、myset size = " << myset.*size*() << *endl*;

return 0;

}

**程序执行结果为：**

**1、myset size = 5  
2、myset size = 0**

**第十六 C++ STL multiset容器详解**

前面章节中，对 set 容器做了详细的讲解。回忆一下，set 容器具有以下几个特性：

（1）不再以键值对的方式存储数据，因为 set 容器专门用于存储键和值相等的键值对，因此该容器中真正存储的是各个键值对的值（value）；

（2）set 容器在存储数据时，会根据各元素值的大小对存储的元素进行排序（默认做升序排序）；

（3）存储到 set 容器中的元素，虽然其类型没有明确用 const 修饰，但正常情况下它们的值是无法被修改的；

（4）set 容器存储的元素必须互不相等。  
在此基础上，[C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 标准库中还提供有一个和 set 容器相似的关联式容器，即 multiset 容器。所谓“相似”，是指 multiset 容器遵循 set 容器的前 3 个特性，仅在第 4 条特性上有差异。和 set 容器不同的是，multiset 容器可以存储多个值相同的元素。

**也就是说，multiset 容器和 set 容器唯一的差别在于，multiset 容器允许存储多个值相同的元素，而 set 容器中只能存储互不相同的元素。**

和 set 类模板一样，multiset 类模板也定义在<set>头文件，并位于 std 命名空间中。这意味着，如果想在程序中使用 multiset 容器，该程序代码应包含如下语句：

**#include <set>**

**u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;**

**注意，第二行代码不是必需的，如果不用，则后续程序中在使用 multiset容器时，需手动注明 std 命名空间（强烈建议初学者使用）。**

**multiset 容器类模板的定义如下所示：**

**template < class T, // 存储元素的类型**

**class Compare = less<T>, // 指定容器内部的排序规则**

**class Alloc = allocator<T> > // 指定分配器对象的类型**

**> class multiset;**

**显然，multiset 类模板有 3 个参数，其中后 2 个参数自带有默认值。值得一提的是，在实际使用中，我们最多只需要使用前 2 个参数即可，第 3 个参数不会用到。**

**一 创建C++ multiset容器的方法**

创建 multiset 容器，无疑需要调用 multiset 类模板中的构造函数。值得一提的是，multiset 类模板提供的构造函数，和 set 类模板中提供创建 set 容器的构造函数，是完全相同的。这意味着，创建 set 容器的方式，也同样适用于创建 multiset 容器。  
 考虑到一些读者可能并未系统学习 set 容器，因此这里还是对 multiset 容器的创建做一下详细的介绍。  
multiset 类模板中提供了 5 种构造函数，也就代表有 5 种创建 multiset 容器的方式，分别如下。  
1) 调用默认构造函数，创建空的 multiset 容器。比如：

**std::multiset<std::string> mymultiset;**

**如果程序中已经默认指定了 std 命令空间，这里可以省略 std::。**

由此就创建好了一个 mymultiset 容器，该容器采用默认的std::less<T>规则，会对存储的 string 类型元素做升序排序。

注意，由于 multiset 容器支持随时向内部添加新的元素，因此创建空 multiset 容器的方法比较常用。  
2)除此之外，multiset 类模板还支持在创建 multiset 容器的同时，对其进行初始化。例如：

**std::multiset<std::string> mymultiset{ "http://c.biancheng.net/java/",**

**"http://c.biancheng.net/stl/",**

**"http://c.biancheng.net/python/" };**

由此即创建好了包含 3 个 string 元素的 mymultiset 容器。由于其采用默认的std::less<T>规则，因此其内部存储 string 元素的顺序如下所示：

**"http://c.biancheng.net/java/"  
"http://c.biancheng.net/python/"  
"http://c.biancheng.net/stl/"**

3) multiset 类模板中还提供了拷贝（复制）构造函数，可以实现在创建新 multiset 容器的同时，将已有 multiset 容器中存储的所有元素全部复制到新 multiset 容器中。  
例如，在第 2 种方式创建的 mymultiset 容器的基础上，执行如下代码：

**std::multiset<std::string> copymultiset(mymultiset);**

**//等同于**

**//std::multiset<std::string> copymultiset = mymultiset;**

**该行代码在创建 copymultiset 容器的基础上，还会将 mymultiset 容器中存储的所有元素，全部复制给 copymultiset 容器一份。**另外，C++ 11 标准还为 multiset 类模板新增了移动构造函数，其功能是实现创建新 multiset 容器的同时，利用临时的 multiset 容器为其初始化。比如：

**multiset<string> retMultiset() {**

**std::multiset<std::string> tempmultiset{ "http://c.biancheng.net/java/",**

**"http://c.biancheng.net/stl/",**

**"http://c.biancheng.net/python/" };**

**return tempmultiset;**

**}**

**std::multiset<std::string> copymultiset(retMultiset());**

**//等同于**

**//std::multiset<std::string> copymultiset = retMultiset();**

**注意，由于 retMultiset() 函数的返回值是一个临时 multiset 容器，因此在初始化 copymultiset 容器时，其内部调用的是 multiset 类模板中的移动构造函数，而非拷贝构造函数。**

**显然，无论是调用复制构造函数还是调用拷贝构造函数，都必须保证这 2 个容器的类型完全一致。**

4) 在第 3 种方式的基础上，multiset 类模板还支持取已有 multiset 容器中的部分元素，来初始化新 multiset 容器。例如：

std::multiset<std::string> mymultiset{ "http://c.biancheng.net/java/",

"http://c.biancheng.net/stl/",

"http://c.biancheng.net/python/" };

std::set<std::string> copymultiset(++mymultiset.begin(), mymultiset.end());

以上初始化的 copyset 容器，其内部仅存有如下 2 个 string 字符串：

"http://c.biancheng.net/python/"  
"http://c.biancheng.net/stl/"

5) 以上几种方式创建的 multiset 容器，都采用了默认的std::less<T>规则。其实，借助 multiset 类模板定义中的第 2 个参数，我们完全可以手动修改 multiset 容器中的排序规则。  
  
下面样例中，使用了 STL 标准库提供的 std::greater<T> 排序方法，作为 multiset 容器内部的排序规则：

std::multiset<std::string, std::greater<string> > mymultiset{

"http://c.biancheng.net/java/",

"http://c.biancheng.net/stl/",

"http://c.biancheng.net/python/" };

通过选用std::greater<string>降序规则，mymultiset 容器中元素的存储顺序为:

"http://c.biancheng.net/stl/"  
"http://c.biancheng.net/python/"  
"http://c.biancheng.net/java/"

**二 C++ multiset容器提供的成员方法**

multiset 容器提供的成员方法，和 set 容器提供的完全一样，如表 1 所示。

表 1 C++ multiset 容器常用成员方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| begin() | 返回指向容器中第一个（注意，是已排好序的第一个）元素的双向迭代器。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| end() | 返回指向容器最后一个元素（注意，是已排好序的最后一个）所在位置后一个位置的双向迭代器，通常和 begin() 结合使用。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| rbegin() | 返回指向最后一个（注意，是已排好序的最后一个）元素的反向双向迭代器。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| rend() | 返回指向第一个（注意，是已排好序的第一个）元素所在位置前一个位置的反向双向迭代器。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的反向双向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| crbegin() | 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| crend() | 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改容器内存储的元素值。 |
| find(val) | 在 multiset 容器中查找值为 val 的元素，如果成功找到，则返回指向该元素的双向迭代器；反之，则返回和 end() 方法一样的迭代器。另外，如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| lower\_bound(val) | 返回一个指向当前 multiset 容器中第一个大于或等于 val 的元素的双向迭代器。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| upper\_bound(val) | 返回一个指向当前 multiset 容器中第一个大于 val 的元素的迭代器。如果 multiset 容器用 const 限定，则该方法返回的是 const 类型的双向迭代器。 |
| equal\_range(val) | 该方法返回一个 pair 对象（包含 2 个双向迭代器），其中 pair.first 和 lower\_bound() 方法的返回值等价，pair.second 和 upper\_bound() 方法的返回值等价。也就是说，该方法将返回一个范围，该范围中包含所有值为 val 的元素。 |
| empty() | 若容器为空，则返回 true；否则 false。 |
| size() | 返回当前 multiset 容器中存有元素的个数。 |
| max\_size() | 返回 multiset 容器所能容纳元素的最大个数，不同的操作系统，其返回值亦不相同。 |
| insert() | 向 multiset 容器中插入元素。 |
| erase() | 删除 multiset 容器中存储的指定元素。 |
| swap() | 交换 2 个 multiset 容器中存储的所有元素。这意味着，操作的 2 个 multiset 容器的类型必须相同。 |
| clear() | 清空 multiset 容器中所有的元素，即令 multiset 容器的 size() 为 0。 |
| emplace() | 在当前 multiset 容器中的指定位置直接构造新元素。其效果和 insert() 一样，但效率更高。 |
| emplace\_hint() | 本质上和 emplace() 在 multiset 容器中构造新元素的方式是一样的，不同之处在于，使用者必须为该方法提供一个指示新元素生成位置的迭代器，并作为该方法的第一个参数。 |
| count(val) | 在当前 multiset 容器中，查找值为 val 的元素的个数，并返回。 |

**注意，虽然 multiset 容器和 set 容器拥有的成员方法完全相同，但由于 multiset 容器允许存储多个值相同的元素，因此诸如 count()、find()、lower\_bound()、upper\_bound()、equal\_range()等方法，更常用于 multiset 容器。**

下面程序演示了表 1 中部分成员函数的用法：

#include <iostream>

#include <set>

#include <string>

using namespace *std*;

int *main*() {

*std*::*multiset*<int> mymultiset{ 1,2,2,2,3,4,5 };

*cout* << "multiset size = " << mymultiset.*size*() << *endl*;

*cout* << "multiset count(2) =" << mymultiset.*count*(2) << *endl*;

//向容器中添加元素 8

mymultiset.*insert*(8);

//删除容器中所有值为 2 的元素

int num = mymultiset.*erase*(2);

*cout* << "删除了 " << num << " 个元素 2" << *endl*;

//输出容器中存储的所有元素

for (auto iter = mymultiset.*begin*(); iter != mymultiset.*end*(); ++iter) {

*cout* << \*iter << " ";

}

return 0;

}

**程序执行结果为：**

**multiset size = 7  
multiset count(2) =3  
删除了 3 个元素 2  
1 3 4 5 8**

**注意，表 1 中大多数成员方法的用法，和 set 容器中相应成员方法的用法是完全一样的，只是调用者不同。因此，如果读者想详细了解表 1 中某个成员方法的用法，可以阅读讲解 set 容器相同成员方法的文章。**

**第十七 如何自定义C++ STL关联式容器的排序规则？**

前面在讲解如何创建 map、multimap、set 以及 multiset 容器时，遗留了一个问题，即如何自定义关联式容器中的排序规则？

**实际上，为关联式容器自定义排序规则的方法，已经在 《[STL priority\_queue自定义排序方法](http://c.biancheng.net/view/vip_7728.html" \t "http://c.biancheng.net/view/_blank)》一节中做了详细的讲解。换句话说，为 Priority\_queue 容器适配器自定义排序规则的方法，同样适用于所有关联式容器。**

总的来说，为关联式容器自定义排序规则，有以下 2 种方法。

**一  使用函数对象自定义排序规则**

在掌握此方法之前，读者必须对函数对象有基本的了解，可阅读《[C++函数对象](http://c.biancheng.net/view/354.html" \t "http://c.biancheng.net/view/_blank)》一节。

无论关联式容器中存储的是基础类型（如 int、double、float 等）数据，还是自定义的结构体变量或类对象（包括 string 类），都可以使用函数对象的方式为该容器自定义排序规则。  
  
下面样例以 set 容器为例，演示了如何用函数对象的方式自定义排序规则：

#include <iostream>

#include <set> // set

#include <string> // string

using namespace *std*;

//定义函数对象类

class cmp {

public:

//重载 () 运算符

bool operator ()(const *string*& a, const *string*& b) {

//按照字符串的长度，做升序排序(即存储的字符串从短到长)

return (a.*length*() < b.*length*());

}

};

int *main*() {

//创建 set 容器，并使用自定义的 cmp 排序规则

*std*::*set*<*string*, cmp>myset{ "http://c.biancheng.net/stl/",

"http://c.biancheng.net/python/",

"http://c.biancheng.net/java/" };

//输出容器中存储的元素

for (auto iter = myset.*begin*(); iter != myset.*end*(); ++iter) {

*cout* << \*iter << *endl*;

}

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/stl/  
http://c.biancheng.net/java/  
http://c.biancheng.net/python/**

重点分析一下 6~13 行代码，其定义了一个函数对象类，并在其重载 () 运算符的方法中自定义了新的排序规则，即按照字符串的长度做升序排序。在此基础上，程序第 17 行代码中，通过将函数对象类的类名 cmp 通过 set 类模板的第 2 个参数传递给 myset 容器，该容器内部排序数据的规则，就改为了以字符串的长度为标准做升序排序。

**需要注意的是，此程序中创建的 myset 容器，由于是以字符串的长度为准进行排序，因此其无法存储相同长度的多个字符串。**

另外，C++ 中的 struct 和 class 非常类似（有关两者区别，可阅读《[C++ struct和class到底有什么区别](http://c.biancheng.net/view/2235.html" \t "http://c.biancheng.net/view/_blank)》一文），前者也可以包含成员变量和成员函数。因此上面程序中，函数对象类 cmp 也可以使用 struct 关键字创建：

**//定义函数对象类**

**struct cmp {**

**//重载 () 运算符**

**bool operator ()(const string &a, const string &b) {**

**//按照字符串的长度，做升序排序(即存储的字符串从短到长)**

**return (a.length() < b.length());**

**}**

**};**

值得一提的是，在定义函数对象类时，也可以将其定义为模板类。比如：

**//定义函数对象模板类**

**template <typename T>**

**class cmp {**

**public:**

**//重载 () 运算符**

**bool operator ()(const T &a, const T &b) {**

**//按照值的大小，做升序排序**

**return a < b;**

**}**

**};**

**注意，此方式必须保证 T 类型元素可以直接使用关系运算符（比如这里的 < 运算符）做比较。**

**二 重载关系运算符实现自定义排序**

其实在 STL 标准库中，本就包含几个可供关联式容器使用的排序规则，如表 1 表示。

|  |  |
| --- | --- |
| 排序规则 | 功能 |
| std::less<T> | 底层采用 < 运算符实现升序排序，各关联式容器默认采用的排序规则。 |
| std::greater<T> | 底层采用 > 运算符实现降序排序，同样适用于各个关联式容器。 |
| std::less\_equal<T> | 底层采用 <= 运算符实现升序排序，多用于 multimap 和 multiset 容器。 |
| std::greater\_equal<T> | 底层采用 >= 运算符实现降序排序，多用于 multimap 和 multiset 容器。 |

值得一提的是，表 1 中的这些排序规则，其底层也是采用函数对象的方式实现的。以 **std::less<T> 为例，其底层实现为：**

**template <typename T>**

**struct less {**

**//定义新的排序规则**

**bool operator()(const T &\_lhs, const T &\_rhs) const {**

**return \_lhs < \_rhs;**

**}**

**}**

在此基础上，当关联式容器中存储的数据类型为自定义的结构体变量或者类对象时，通过对现有排序规则中所用的关系运算符进行重载，也能实现自定义排序规则的目的。

注意，当关联式容器中存储的元素类型为结构体指针变量或者类的指针对象时，只能使用函数对象的方式自定义排序规则，此方法不再适用。

举个例子：

#include <iostream>

#include <set> // set

#include <string> // string

using namespace *std*;

//自定义类

class myString {

public:

//定义构造函数，向 myset 容器中添加元素时会用到

myString(*string* tempStr) :str(tempStr) {};

//获取 str 私有对象，由于会被私有对象调用，因此该成员方法也必须为 const 类型

*string* getStr() const;

private:

*string* str;

};

*string* myString::getStr() const {

return this->str;

}

//重载 < 运算符，参数必须都为 const 类型

bool operator <(const myString& stra, const myString& strb) {

//以字符串的长度为标准比较大小

return stra.getStr().*length*() < strb.getStr().*length*();

}

int *main*() {

//创建空 set 容器，仍使用默认的 less<T> 排序规则

*std*::*set*<myString>myset;

//向 set 容器添加元素，这里会调用 myString 类的构造函数

myset.*emplace*("http://c.biancheng.net/stl/");

myset.*emplace*("http://c.biancheng.net/c/");

myset.*emplace*("http://c.biancheng.net/python/");

//

for (auto iter = myset.*begin*(); iter != myset.*end*(); ++iter) {

myString mystr = \*iter;

*cout* << mystr.getStr() << *endl*;

}

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/c/  
http://c.biancheng.net/stl/  
http://c.biancheng.net/python/**

在这个程序中，虽然 myset 容器表面仍采用默认的 std::less<T> 排序规则，但由于我们对其所用的 < 运算符进行了重载，使得 myset 容器内部实则是以字符串的长度为基准，对各个 mystring 类对象进行排序。  
 另外，上面程序以全局函数的形式实现对 < 运算符的重载，还可以使用成员函数或者友元函数的形式实现。其中，当以成员函数的方式重载 < 运算符时，该成员函数必须声明为 const 类型，且参数也必须为 const 类型：

**bool operator <(const myString & tempStr) const {**

**//以字符串的长度为标准比较大小**

**return this->str.length() < tempStr.str.length();**

**}**

至于参数的传值方式是采用按引用传递还是按值传递，都可以（建议采用按引用传递，效率更高）。

**同样，如果以友元函数的方式重载 < 运算符时，要求参数必须使用 const 修饰：**

**//类中友元函数的定义**

**friend bool operator <(const myString &a, const myString &b);**

**//类外部友元函数的具体实现**

**bool operator <(const myString &stra, const myString &strb) {**

**//以字符串的长度为标准比较大小**

**return stra.str.length() < strb.str.length();**

**}**

当然，本节所讲自定义排序规则的方法并不仅仅适用于 set 容器，其它关联式容器（map、multimap、multiset）也同样适用，有兴趣的读者可自行编写代码验证。

**第十八 如何修改关联式容器中键值对的键？**

通过前面的学习，读者已经掌握了所有关联式容器（包括 map、multimap、set 和 multiset）的特性和用法。其中需要指出的是，对于如何修改容器中某个键值对的键，所有关联式容器可以采用同一种解决思路，即先删除该键值对，然后再向容器中添加修改之后的新键值对。  
 那么，是否可以不删除目标键值对，而直接修改它的键呢？接下来就围绕此问题，给读者展开详细的讲解。  
 首先可以明确的是，map 和 multimap 容器只能采用“先删除，再添加”的方式修改某个键值对的键。原因很简单，C++ STL 标准中明确规定，map 和 multimap 容器用于存储类型为 pair<const K, V> 的键值对。显然，只要目标键值对存储在当前容器中，键的值就无法被修改。  
举个例子：

**map<int, int> mymap{ {1,10},{2,20} };**

**//map 容器的键为 const 类型，不能被修改**

**mymap.begin()->first = 100;**

**multimap<int, int> mymultimap{ {10,100},{20,200} };**

**//multimap 容器的键为 const 类型，同样不能被修改**

**mymultimap.begin()->first = 100;**

其中，第 3 行代码试图直接将 mymap 容器中 {1,10} 的键改为 100，同样第 7 行代码试图直接将 mymultimap 容器中 {10,100} 的键改为 100，它们都是不能通过编译的。  
 正如上面例子中演示的那样，直接修改 map 或 multimap 容器中某个键值对的键是行不通的。但对于 set 或者 multiset 容器来说，却是可行的。  
 和 map、multimap 不同，C++ STL 标准中并没有用 const 限定 set 和 multiset 容器中存储元素的类型。换句话说，对于 set<T> 或者 multiset<T> 类型的容器，其存储元素的类型是 T 而不是 const T。  
 事实上，对 set 和 multiset 容器中的元素类型作 const 修饰，是违背常理的。举个例子，假设我们使用 set 容器存储多个学生信息，如下是一个表示学生的类：

**class student {**

**public:**

**student(string name, int id, int age) :name(name), id(id), age(age) {**

**}**

**const int& getid() const {**

**return id;**

**}**

**void setname(const string name){**

**this->name = name;**

**}**

**string getname() const{**

**return name;**

**}**

**void setage(int age){**

**this->age = age;**

**}**

**int getage() const{**

**return age;**

**}**

**private:**

**string name;**

**int id;**

**int age;**

**};**

在创建 set 容器之前，我们还需要为其设计一个排序规则，这里假定以每个学生的 id 做升序排序，其排序规则如下：

**class cmp {**

**public:**

**bool operator ()(const student &stua, const student &stub) {**

**//按照字符串的长度，做升序排序(即存储的字符串从短到长)**

**return stua.getid() < stub.getid();**

**}**

**};**

做完以上所有的准备工作后，就可以创建一个可存储 student 对象的 set 容器了，比如：

**set<student, cmp> myset{ {"zhangsan",10,20},{"lisi",20,21},{"wangwu",15,19} };**

**由此创建的 myset 容器中，存储的数据依次为：**

**{"zhangsan",10,20}  
{"wangwu",15,19}  
{"lisi",20,21}**

注意，set 容器中每个元素也可以看做是键和值相等的键值对，但对于这里的 myset 容器来说，其实每个 student 对象的 id 才是真正的键，其它信息（name 和 age）只不过是和 id 绑定在一起而已。因此，在不破坏 myset 容器中元素的有序性的前提下（即不修改每个学生的 id），学生的其它信息是应该允许修改的，但有一个前提，即 myset 容器中存储的各个 student 对象不能被 const 修饰（这也是 set 容器中的元素类型不能被 const 修饰的原因）。

**总之，set 和 multiset 容器的元素类型没有用 const 修饰。所以从语法的角度分析，我们可以直接修改容器中元素的值，但一定不要修改元素的键。**

例如，在已创建好的 myset 容器的基础上，如下代码尝试修改 myset 容器中某个学生的 name 名字：

**set<student>::iterator iter = mymap.begin();**

**(\*iter).setname("xiaoming");**

**注意，如果读者运行代码会发现，它也是无法通过编译的。**  
虽然 C++ STL 标准没有用 const 修饰 set 或者 multiset 容器中元素的类型，但也做了其它工作来限制用户修改容器的元素。例如上面代码中，\*iter 会调用 operator\*，其返回的是一个 const T& 类型元素。这意味着，C++ STL 标准不允许用户借助迭代器来直接修改 set 或者 multiset 容器中的元素。  
 那么，如何才能正确修改 set 或 multiset 容器中的元素呢？最直接的方式就是借助 const\_cast 运算符，该运算符可以去掉指针或者引用的 const 限定符。

有关 const\_cast 运算符的用法，由于不是本节重点，这里不再做详细讲解，有兴趣的读者可自行查阅相关资料。

比如，我们只需要借助 const\_cast 运算符对上面程序稍作修改，就可以运行成功：

**set<student>::iterator iter = mymap.begin();**

**const\_cast<student&>(\*iter).setname("xiaoming");**

由此，mymap 容器中的 {"zhangsan",10,20} 就变成了 {"xiaoming",10,20}。

再次强调，虽然使用 const\_cast 能直接修改 set 或者 multiset 容器中的元素，但一定不要修改元素的键！如果要修改，只能采用“先删除，再添加”的方式。另外，不要试图以同样的方式修改 map 或者 multimap 容器中键值对的键，这违反了 C++ STL 标准的规定。

**总结**

**总的来说，map 和 multimap 容器中元素的键是无法直接修改的，但借助 const\_cast，我们可以直接修改 set 和 multiset 容器中元素的非键部分。  
为了加深读者的理解，如下是和本节内容相关的完整程序，读者可直接拷贝下来：**

#include<iostream>

#include<set>

#include<string>

using namespace *std*;

class student {

public:

student(*string* name, int id, int age) :name(name), id(id), age(age) {

}

const int& getid() const {

return id;

}

void setname(const *string* name) {

this->name = name;

}

*string* getname() const {

return name;

}

void setage(int age) {

this->age = age;

}

int getage() const {

return age;

}

void display()const {

*cout* << id << " " << name << " " << age << *endl*;

}

private:

*string* name;

int id;

int age;

};

//自定义 myset 容器的排序规则

class cmp {

public:

bool operator ()(const student& stua, const student& stub) {

//按照字符串的长度，做升序排序(即存储的字符串从短到长)

return stua.getid() < stub.getid();

}

};

int *main*() {

*set*<student, cmp> mymap{ {"zhangsan",10,20},{"lisi",20,21},{"wangwu",15,19} };

*set*<student>::*iterator* iter = mymap.*begin*();

//直接将 {"zhangsan",10,20} 中的 "zhangsan" 修改为 "xiaoming"

const\_cast<student&>(\*iter).setname("xiaoming");

while (iter != mymap.*end*()) {

(\*iter).display();

++iter;

}

return 0;

}

**程序执行结果为：**

**10 xiaoming 20  
15 wangwu 19  
20 lisi 21**

**第四章 C++ STL无序关联式容器**

继 map、multimap、set、multiset 关联式容器之后，从本节开始，再讲解一类“特殊”的关联式容器，它们常被称为“无序容器”、“哈希容器”或者“无序关联容器”。

**注意，无序容器是 [C++](http://c.biancheng.net/cplus/" \t "http://c.biancheng.net/view/_blank) 11 标准才正式引入到 [STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 标准库中的，这意味着如果要使用该类容器，则必须选择支持 C++ 11 标准的编译器。**

和关联式容器一样，无序容器也使用键值对（pair 类型）的方式存储数据。不过，本教程将二者分开进行讲解，因为它们有本质上的不同：

（1）关联式容器的底层实现采用的树存储结构，更确切的说是红黑树结构；

（2）无序容器的底层实现采用的是哈希表的存储结构。

C++ STL 底层采用哈希表实现无序容器时，会将所有数据存储到一整块连续的内存空间中，并且当数据存储位置发生冲突时，解决方法选用的是“链地址法”（又称“开链法”）。有关哈希表存储结构，读者可阅读《[哈希表(散列表)详解](http://c.biancheng.net/view/3437.html" \t "http://c.biancheng.net/view/_blank)》一文做详细了解。

基于底层实现采用了不同的[数据结构](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)，因此和关联式容器相比，无序容器具有以下 2 个特点：

（1）无序容器内部存储的键值对是无序的，各键值对的存储位置取决于该键值对中的键，

（2）和关联式容器相比，无序容器擅长通过指定键查找对应的值（平均时间复杂度为 O(1)）；但对于使用迭代器遍历容器中存储的元素，无序容器的执行效率则不如关联式容器。

**第一C++ STL无序容器种类**

和关联式容器一样，无序容器只是一类容器的统称，其包含有 4 个具体容器，分别为unordered\_map、unordered\_multimap、unordered\_set 以及 unordered\_multiset。  
  
表 1 对这 4 种无序容器的功能做了详细的介绍。

表 1 无序容器种类

|  |  |
| --- | --- |
| **无序容器** | **功能** |
| unordered\_map | 存储键值对 <key, value> 类型的元素，其中各个键值对键的值不允许重复，且该容器中存储的键值对是无序的。 |
| unordered\_multimap | 和 unordered\_map 唯一的区别在于，该容器允许存储多个键相同的键值对。 |
| unordered\_set | 不再以键值对的形式存储数据，而是直接存储数据元素本身（当然也可以理解为，该容器存储的全部都是键 key 和值 value 相等的键值对，正因为它们相等，因此只存储 value 即可）。另外，该容器存储的元素不能重复，且容器内部存储的元素也是无序的。 |
| unordered\_multiset | 和 unordered\_set 唯一的区别在于，该容器允许存储值相同的元素。 |

可能读者已经发现，以上 4 种无序容器的名称，仅是在前面所学的 4 种关联式容器名称的基础上，添加了 "unordered\_"。如果读者已经学完了 map、multimap、set 和 multiset 容器不难发现，以 map 和 unordered\_map 为例，其实它们仅有一个区别，即 map 容器内存会对存储的键值对进行排序，而 unordered\_map 不会。

也就是说，C++ 11 标准的 STL 中，在已提供有 4 种关联式容器的基础上，又新增了各自的“unordered”版本（无序版本、哈希版本），提高了查找指定元素的效率。  
 有读者可能会问，既然无序容器和之前所学的关联式容器类似，那么在实际使用中应该选哪种容器呢？总的来说，实际场景中如果涉及大量遍历容器的操作，建议首选关联式容器；反之，如果更多的操作是通过键获取对应的值，则应首选无序容器。  
 为了加深读者对无序容器的认识，这里以 unordered\_map 容器为例，举个例子（不必深究该容器的具体用法）：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建并初始化一个 unordered\_map 容器，其存储的 <string,string> 类型的键值对

*std*::*unordered\_map*<*std*::*string*, *std*::*string*> my\_uMap{

{"C语言教程","http://c.biancheng.net/c/"},

{"Python教程","http://c.biancheng.net/python/"},

{"Java教程","http://c.biancheng.net/java/"} };

//查找指定键对应的值，效率比关联式容器高

*string* str = my\_uMap.*at*("C语言教程");

*cout* << "str = " << str << *endl*;

//使用迭代器遍历哈希容器，效率不如关联式容器

for (auto iter = my\_uMap.*begin*(); iter != my\_uMap.*end*(); ++iter)

{

//pair 类型键值对分为 2 部分

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**str = http://c.biancheng.net/c/  
C语言教程 http://c.biancheng.net/c/  
Python教程 http://c.biancheng.net/python/  
Java教程 http://c.biancheng.net/java/**

**第二C++ STL unordered\_map容器用法详解**

C++ STL 标准库中提供有 4 种无序关联式容器，本节先讲解 unordered\_map 容器。  
unordered\_map 容器，直译过来就是"无序 map 容器"的意思。所谓“无序”，指的是 unordered\_map 容器不会像 map 容器那样对存储的数据进行排序。换句话说，unordered\_map 容器和 map 容器仅有一点不同，即 map 容器中存储的数据是有序的，而 unordered\_map 容器中是无序的。

对于已经学过 map 容器的读者，可以将 unordered\_map 容器等价为无序的 map 容器。

具体来讲，unordered\_map 容器和 map 容器一样，以键值对（pair类型）的形式存储数据，存储的各个键值对的键互不相同且不允许被修改。但由于 unordered\_map 容器底层采用的是哈希表存储结构，该结构本身不具有对数据的排序功能，所以此容器内部不会自行对存储的键值对进行排序。  
 值得一提的是，unordered\_map 容器在<unordered\_map>头文件中，并位于 std 命名空间中。因此，如果想使用该容器，代码中应包含如下语句：

#include <unordered\_map>

using namespace std;

注意，第二行代码不是必需的，但如果不用，则后续程序中在使用此容器时，需手动注明 std 命名空间（强烈建议初学者使用）。

unordered\_map 容器模板的定义如下所示：

**template < class Key, //键值对中键的类型**

**class T, //键值对中值的类型**

**class Hash = hash<Key>, //容器内部存储键值对所用的哈希函数**

**class Pred = equal\_to<Key>, //判断各个键值对键相同的规则**

**class Alloc = allocator< pair<const Key,T> > // 指定分配器对象的类型**

**> class unordered\_map;**

以上 5 个参数中，必须显式给前 2 个参数传值，并且除特殊情况外，最多只需要使用前 4 个参数，各自的含义和功能如表 1 所示。

表 1 unordered\_map 容器模板类的常用参数

|  |  |
| --- | --- |
| **参数** | **含义** |
| <key,T> | 前 2 个参数分别用于确定键值对中键和值的类型，也就是存储键值对的类型。 |
| Hash = hash<Key> | 用于指明容器在存储各个键值对时要使用的哈希函数，默认使用 STL 标准库提供的 hash<key> 哈希函数。注意，默认哈希函数只适用于基本数据类型（包括 string 类型），而不适用于自定义的结构体或者类。 |
| Pred = equal\_to<Key> | 要知道，unordered\_map 容器中存储的各个键值对的键是不能相等的，而判断是否相等的规则，就由此参数指定。默认情况下，使用 STL 标准库中提供的 equal\_to<key> 规则，该规则仅支持可直接用 == 运算符做比较的数据类型。 |

**总的来说，当无序容器中存储键值对的键为自定义类型时，默认的哈希函数 hash 以及比较函数 equal\_to 将不再适用，只能自己设计适用该类型的哈希函数和比较函数，并显式传递给 Hash 参数和 Pred 参数。至于如何实现自定义，后续章节会做详细讲解。**

**一创建C++ unordered\_map容器的方法**

常见的创建 unordered\_map 容器的方法有以下几种  
1) 通过调用 unordered\_map 模板类的默认构造函数，可以创建空的 unordered\_map 容器。比如：

**std::unordered\_map<std::string, std::string> umap;**

由此，就创建好了一个可存储 <string,string> 类型键值对的 unordered\_map 容器。  
2) 当然，在创建 unordered\_map 容器的同时，可以完成初始化操作。比如：

**std::unordered\_map<std::string, std::string> umap{**

**{"Python教程","http://c.biancheng.net/python/"},**

**{"Java教程","http://c.biancheng.net/java/"},**

**{"Linux教程","http://c.biancheng.net/linux/"} };**

通过此方法创建的 umap 容器中，就包含有 3 个键值对元素。  
3) 另外，还可以调用 unordered\_map 模板中提供的复制（拷贝）构造函数，将现有 unordered\_map 容器中存储的键值对，复制给新建 unordered\_map 容器。  
例如，在第二种方式创建好 umap 容器的基础上，再创建并初始化一个 umap2 容器：

**std::unordered\_map<std::string, std::string> umap2(umap);**

由此，umap2 容器中就包含有 umap 容器中所有的键值对。  
除此之外，C++ 11 标准中还向 unordered\_map 模板类增加了移动构造函数，即以右值引用的方式将临时 unordered\_map 容器中存储的所有键值对，全部复制给新建容器。例如：

**//返回临时 unordered\_map 容器的函数**

**std::unordered\_map <std::string, std::string > retUmap(){**

**std::unordered\_map<std::string, std::string>tempUmap{**

**{"Python教程","http://c.biancheng.net/python/"},**

**{"Java教程","http://c.biancheng.net/java/"},**

**{"Linux教程","http://c.biancheng.net/linux/"} };**

**return tempUmap;**

**}**

**//调用移动构造函数，创建 umap2 容器**

**std::unordered\_map<std::string, std::string> umap2(retUmap());**

**注意，无论是调用复制构造函数还是拷贝构造函数，必须保证 2 个容器的类型完全相同。**  
4) 当然，如果不想全部拷贝，可以使用 unordered\_map 类模板提供的迭代器，在现有 unordered\_map 容器中选择部分区域内的键值对，为新建 unordered\_map 容器初始化。例如：

**//传入 2 个迭代器，**

**std::unordered\_map<std::string, std::string> umap2(++umap.begin(),umap.end());**

通过此方式创建的 umap2 容器，其内部就包含 umap 容器中除第 1 个键值对外的所有其它键值对。

**二C++ unordered\_map容器的成员方法**

unordered\_map 既可以看做是关联式容器，更属于自成一脉的无序容器。因此在该容器模板类中，既包含一些在学习关联式容器时常见的成员方法，还有一些属于无序容器特有的成员方法。  
表 2 列出了 unordered\_map 类模板提供的所有常用的成员方法以及各自的功能。

表 2 unordered\_map类模板成员方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| begin() | 返回指向容器中第一个键值对的正向迭代器。 |
| end() | 返回指向容器中最后一个键值对之后位置的正向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。 |
| empty() | 若容器为空，则返回 true；否则 false。 |
| size() | 返回当前容器中存有键值对的个数。 |
| max\_size() | 返回容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。 |
| operator[key] | 该模板类中重载了 [] 运算符，其功能是可以向访问数组中元素那样，只要给定某个键值对的键 key，就可以获取该键对应的值。注意，如果当前容器中没有以 key 为键的键值对，则其会使用该键向当前容器中插入一个新键值对。 |
| at(key) | 返回容器中存储的键 key 对应的值，如果 key 不存在，则会抛出 out\_of\_range 异常。 |
| find(key) | 查找以 key 为键的键值对，如果找到，则返回一个指向该键值对的正向迭代器；反之，则返回一个指向容器中最后一个键值对之后位置的迭代器（如果 end() 方法返回的迭代器）。 |
| count(key) | 在容器中查找以 key 键的键值对的个数。 |
| equal\_range(key) | 返回一个 pair 对象，其包含 2 个迭代器，用于表明当前容器中键为 key 的键值对所在的范围。 |
| emplace() | 向容器中添加新键值对，效率比 insert() 方法高。 |
| emplace\_hint() | 向容器中添加新键值对，效率比 insert() 方法高。 |
| insert() | 向容器中添加新键值对。 |
| erase() | 删除指定键值对。 |
| clear() | 清空容器，即删除容器中存储的所有键值对。 |
| swap() | 交换 2 个 unordered\_map 容器存储的键值对，前提是必须保证这 2 个容器的类型完全相等。 |
| bucket\_count() | 返回当前容器底层存储键值对时，使用桶（一个线性链表代表一个桶）的数量。 |
| max\_bucket\_count() | 返回当前系统中，unordered\_map 容器底层最多可以使用多少桶。 |
| bucket\_size(n) | 返回第 n 个桶中存储键值对的数量。 |
| bucket(key) | 返回以 key 为键的键值对所在桶的编号。 |
| load\_factor() | 返回 unordered\_map 容器中当前的负载因子。负载因子，指的是的当前容器中存储键值对的数量（size()）和使用桶数（bucket\_count()）的比值，即 load\_factor() = size() / bucket\_count()。 |
| max\_load\_factor() | 返回或者设置当前 unordered\_map 容器的负载因子。 |
| rehash(n) | 将当前容器底层使用桶的数量设置为 n。 |
| reserve() | 将存储桶的数量（也就是 bucket\_count() 方法的返回值）设置为至少容纳count个元（不超过最大负载因子）所需的数量，并重新整理容器。 |
| hash\_function() | 返回当前容器使用的哈希函数对象。 |

注意，对于实现互换 2 个相同类型 unordered\_map 容器的键值对，除了可以调用该容器模板类中提供的 swap() 成员方法外，STL 标准库还提供了同名的 swap() 非成员函数。

下面的样例演示了表 2 中部分成员方法的用法：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建空 umap 容器

*unordered\_map*<*string*, *string*> umap;

//向 umap 容器添加新键值对

umap.*emplace*("Python教程", "http://c.biancheng.net/python/");

umap.*emplace*("Java教程", "http://c.biancheng.net/java/");

umap.*emplace*("Linux教程", "http://c.biancheng.net/linux/");

//输出 umap 存储键值对的数量

*cout* << "umap size = " << umap.*size*() << *endl*;

//使用迭代器输出 umap 容器存储的所有键值对

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**umap size = 3  
Python教程 http://c.biancheng.net/python/  
Linux教程 http://c.biancheng.net/linux/  
Java教程 http://c.biancheng.net/java/**

**有关表 2 中其它成员方法的用法，后续章节会做详细讲解。当然，读者也可以自行查询 [C++ STL标准库手册](http://www.cplusplus.com/reference/unordered_map/unordered_map/" \t "http://c.biancheng.net/view/_blank)。**

**第三深度剖析C++无序容器的底层实现机制**

在阅读本节内容之前，读者需了解哈希表存储结构的原理，可猛击《[哈希表（散列表）详解](http://c.biancheng.net/view/3437.html" \t "http://c.biancheng.net/view/_blank)》一节。

在了解哈希表存储结构的基础上，本节将具体分析 C++ STL 无序容器（哈希容器）底层的实现原理。  
 C++ STL 标准库中，不仅是 unordered\_map 容器，所有无序容器的底层实现都采用的是哈希表存储结构。更准确地说，是用“链地址法”（又称“开链法”）解决数据存储位置发生冲突的哈希表，整个存储结构如图 1 所示。

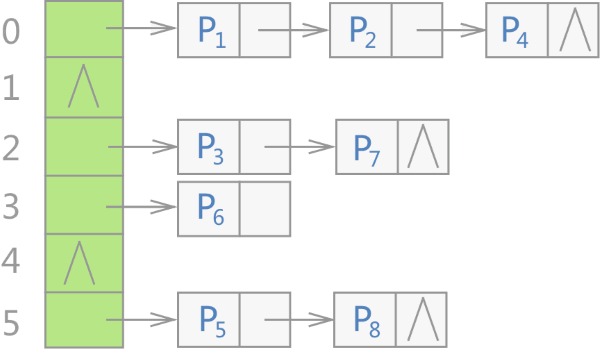


图 1 C++ STL 无序容器存储状态示意图

其中，Pi 表示存储的各个键值对。

可以看到，当使用无序容器存储键值对时，会先申请一整块连续的存储空间，但此空间并不用来直接存储键值对，而是存储各个链表的头指针，各键值对真正的存储位置是各个链表的节点。

**注意，STL 标准库通常选用 vector 容器存储各个链表的头指针。**

不仅如此，在 C++ STL 标准库中，将图 1 中的各个链表称为桶（bucket），每个桶都有自己的编号（从 0 开始）。当有新键值对存储到无序容器中时，整个存储过程分为如下几步：

（1）将该键值对中键的值带入设计好的哈希函数，会得到一个哈希值（一个整数，用 H 表示）；

（2）将 H 和无序容器拥有桶的数量 n 做整除运算（即 H % n），该结果即表示应将此键值对存储到的桶的编号；

（3）建立一个新节点存储此键值对，同时将该节点链接到相应编号的桶上。

另外值得一提的是，哈希表存储结构还有一个重要的属性，称为负载因子（load factor）。该属性同样适用于无序容器，用于衡量容器存储键值对的空/满程序，即负载因子越大，意味着容器越满，即各链表中挂载着越多的键值对，这无疑会降低容器查找目标键值对的效率；反之，负载因子越小，容器肯定越空，但并不一定各个链表中挂载的键值对就越少。  
 举个例子，如果设计的哈希函数不合理，使得各个键值对的键带入该函数得到的哈希值始终相同（所有键值对始终存储在同一链表上）。这种情况下，即便增加桶数是的负载因子减小，该容器的查找效率依旧很差。  
 无序容器中，负载因子的计算方法为：

**负载因子 = 容器存储的总键值对 / 桶数**

默认情况下，无序容器的最大负载因子为 1.0。如果操作无序容器过程中，使得最大复杂因子超过了默认值，则容器会自动增加桶数，并重新进行哈希，以此来减小负载因子的值。需要注意的是，此过程会导致容器迭代器失效，但指向单个键值对的引用或者指针仍然有效。

这也就解释了，为什么我们在操作无序容器过程中，键值对的存储顺序有时会“莫名”的发生变动。  
 C++ STL 标准库为了方便用户更好地管控无序容器底层使用的哈希表存储结构，各个无序容器的模板类中都提供表 2 所示的成员方法。

表 2 无序容器管理哈希表的成员方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| bucket\_count() | 返回当前容器底层存储键值对时，使用桶的数量。 |
| max\_bucket\_count() | 返回当前系统中，unordered\_map 容器底层最多可以使用多少个桶。 |
| bucket\_size(n) | 返回第 n 个桶中存储键值对的数量。 |
| bucket(key) | 返回以 key 为键的键值对所在桶的编号。 |
| load\_factor() | 返回 unordered\_map 容器中当前的负载因子。 |
| max\_load\_factor() | 返回或者设置当前 unordered\_map 容器的最大负载因子。 |
| rehash(n) | 尝试重新调整桶的数量为等于或大于 n 的值。如果 n 大于当前容器使用的桶数，则该方法会是容器重新哈希，该容器新的桶数将等于或大于 n。反之，如果 n 的值小于当前容器使用的桶数，则调用此方法可能没有任何作用。 |
| reserve(n) | 将容器使用的桶数（bucket\_count() 方法的返回值）设置为最适合存储 n 个元素的桶数。 |
| hash\_function() | 返回当前容器使用的哈希函数对象。 |

下面的程序以学过的 unordered\_map 容器为例，演示了表 2 中部分成员方法的用法：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建空 umap 容器

*unordered\_map*<*string*, *string*> umap;

*cout* << "umap 初始桶数: " << umap.*bucket\_count*() << *endl*;

*cout* << "umap 初始负载因子: " << umap.*load\_factor*() << *endl*;

*cout* << "umap 最大负载因子: " << umap.*max\_load\_factor*() << *endl*;

//设置 umap 使用最适合存储 9 个键值对的桶数

umap.*reserve*(9);

*cout* << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << *endl*;

*cout* << "umap 新桶数: " << umap.*bucket\_count*() << *endl*;

*cout* << "umap 新负载因子: " << umap.*load\_factor*() << *endl*;

//向 umap 容器添加 3 个键值对

umap["Python教程"] = "http://c.biancheng.net/python/";

umap["Java教程"] = "http://c.biancheng.net/java/";

umap["Linux教程"] = "http://c.biancheng.net/linux/";

//调用 bucket() 获取指定键值对位于桶的编号

*cout* << "以\"Python教程\"为键的键值对，位于桶的编号为:" << umap.*bucket*("Python教程") << *endl*;

//自行计算某键值对位于哪个桶

auto fn = umap.*hash\_function*();

*cout* << "计算以\"Python教程\"为键的键值对，位于桶的编号为：" << fn("Python教程") % (umap.*bucket\_count*()) << *endl*;

return 0;

}

**程序执行结果为：**

**umap 初始桶数: 8  
umap 初始负载因子: 0  
umap 最大负载因子: 1  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
umap 新桶数: 16  
umap 新负载因子: 0  
以"Python教程"为键的键值对，位于桶的编号为:9  
计算以"Python教程"为键的键值对，位于桶的编号为：9**

从输出结果可以看出，对于空的 umap 容器，初始状态下会分配 8 个桶，并且默认最大负载因子为 1.0，但由于其为存储任何键值对，因此负载因子值为 0。  
 与此同时，程序中调用 reverse() 成员方法，是 umap 容器的桶数改为了 16，其最适合存储 9 个键值对。从侧面可以看出，一旦负载因子大于 1.0（9/8 > 1.0），则容器所使用的桶数就会翻倍式（8、16、32、...）的增加。  
 程序最后还演示了如何手动计算出指定键值对存储的桶的编号，其计算结果和使用 bucket() 成员方法得到的结果是一致的。

**关于表 2 中成员方法的具体语法和用法，都很简单，不再过多赘述，感兴趣的读者可自行翻阅 [C++ STL手册](http://www.cplusplus.com/reference/unordered_map/unordered_map/" \t "http://c.biancheng.net/view/_blank)。**

**第四C++ unordered\_map迭代器的用法**

C++ STL 标准库中，unordered\_map 容器迭代器的类型为前向迭代器（又称正向迭代器）。这意味着，假设 p 是一个前向迭代器，则其只能进行 \*p、p++、++p 操作，且 2 个前向迭代器之间只能用 == 和 != 运算符做比较。  
 在 unordered\_map 容器模板中，提供了表 1 所示的成员方法，可用来获取指向指定位置的前向迭代器。

表 1 C++ unordered\_map迭代器相关成员方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| begin() | 返回指向容器中第一个键值对的正向迭代器。 |
| end() | 返回指向容器中最后一个键值对之后位置的正向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。 |
| find(key) | 查找以 key 为键的键值对，如果找到，则返回一个指向该键值对的正向迭代器；反之，则返回一个指向容器中最后一个键值对之后位置的迭代器（如果 end() 方法返回的迭代器）。 |
| equal\_range(key) | 返回一个 pair 对象，其包含 2 个迭代器，用于表明当前容器中键为 key 的键值对所在的范围。 |

值得一提的是，equal\_range(key) 很少用于 unordered\_map 容器，因为该容器中存储的都是键不相等的键值对，即便调用该成员方法，得到的 2 个迭代器所表示的范围中，最多只包含 1 个键值对。事实上，该成员方法更适用于 unordered\_multimap 容器（该容器后续章节会做详细讲解）。

下面的程序演示了表 1 中部分成员方法的用法。

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap{

{"Python教程","http://c.biancheng.net/python/"},

{"Java教程","http://c.biancheng.net/java/"},

{"Linux教程","http://c.biancheng.net/linux/"} };

*cout* << "umap 存储的键值对包括：" << *endl*;

//遍历输出 umap 容器中所有的键值对

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

*cout* << "<" << iter->*first* << ", " << iter->*second* << ">" << *endl*;

}

//获取指向指定键值对的前向迭代器

*unordered\_map*<*string*, *string*>::*iterator* iter = umap.*find*("Java教程");

*cout* << "umap.find(\"Java教程\") = " << "<" << iter->*first* << ", " << iter->*second* << ">" << *endl*;

return 0;

}

程序执行结果为：

umap 存储的键值对包括：  
<Python教程, http://c.biancheng.net/python/>  
<Linux教程, http://c.biancheng.net/linux/>  
<Java教程, http://c.biancheng.net/java/>  
umap.find("Java教程") = <Java教程, http://c.biancheng.net/java/>

需要注意的是，在操作 unordered\_map 容器过程（尤其是向容器中添加新键值对）中，一旦当前容器的负载因子超过最大负载因子（默认值为 1.0），该容器就会适当增加桶的数量（通常是翻一倍），并自动执行 rehash() 成员方法，重新调整各个键值对的存储位置（此过程又称“重哈希”），此过程很可能导致之前创建的迭代器失效。

**所谓迭代器失效，针对的是那些用于表示容器内某个范围的迭代器，由于重哈希会重新调整每个键值对的存储位置，所以容器重哈希之后，之前表示特定范围的迭代器很可能无法再正确表示该范围。但是，重哈希并不会影响那些指向单个键值对元素的迭代器。**

**举个例子：**

#include <iostream>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<int, int> umap;

//向 umap 容器添加 50 个键值对

for (int i = 1; i <= 50; i++) {

umap.*emplace*(i, i);

}

//获取键为 49 的键值对所在的范围

auto pair = umap.*equal\_range*(49);

//输出 pair 范围内的每个键值对的键的值

for (auto iter = pair.*first*; iter != pair.*second*; ++iter) {

*cout* << iter->*first* << " ";

}

*cout* << *endl*;

//手动调整最大负载因子数

umap.*max\_load\_factor*(3.0);

//手动调用 rehash() 函数重哈希

umap.*rehash*(10);

//重哈希之后，pair 的范围可能会发生变化

for (auto iter = pair.*first*; iter != pair.*second*; ++iter) {

*cout* << iter->*first* << " ";

}

return 0;

}

**程序执行结果为：**

**49  
49 17**

观察输出结果不难发现，之前用于表示键为 49 的键值对所在范围的 2 个迭代器，重哈希之后表示的范围发生了改变。

**经测试，用于遍历整个容器的 begin()/end() 和 cbegin()/cend() 迭代器对，重哈希只会影响遍历容器内键值对的顺序，整个遍历的操作仍然可以顺利完成。**

**第五C++ STL unordered\_map获取元素的4种方法**

通过前面的学习我们知道，unordered\_map 容器以键值对的方式存储数据。为了方便用户快速地从该类型容器提取出目标元素（也就是某个键值对的值），unordered\_map 容器类模板中提供了以下几种方法。  
  
1) unordered\_map 容器类模板中，实现了对 [ ] 运算符的重载，使得我们可以像“利用下标访问普通数组中元素”那样，通过目标键值对的键获取到该键对应的值。

举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap{

{"Python教程","http://c.biancheng.net/python/"},

{"Java教程","http://c.biancheng.net/java/"},

{"Linux教程","http://c.biancheng.net/linux/"} };

//获取 "Java教程" 对应的值

*string* str = umap["Java教程"];

*cout* << str << *endl*;

return 0;

}

**程序输出结果为：**

**http://c.biancheng.net/java/**

需要注意的是，如果当前容器中并没有存储以 [ ] 运算符内指定的元素作为键的键值对，则此时 [ ] 运算符的功能将转变为：向当前容器中添加以目标元素为键的键值对。

举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建空 umap 容器

*unordered\_map*<*string*, *string*> umap;

//[] 运算符在 = 右侧

*string* str = umap["STL教程"];

//[] 运算符在 = 左侧

umap["C教程"] = "http://c.biancheng.net/c/";

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**C教程 http://c.biancheng.net/c/  
STL教程**

可以看到，当使用 [ ] 运算符向 unordered\_map 容器中添加键值对时，分为 2 种情况：

当 [ ] 运算符位于赋值号（=）右侧时，则新添加键值对的键为 [ ] 运算符内的元素，其值为键值对要求的值类型的默认值（string 类型默认值为空字符串）；

当 [ ] 运算符位于赋值号（=）左侧时，则新添加键值对的键为 [ ] 运算符内的元素，其值为赋值号右侧的元素。  
2) unordered\_map 类模板中，还提供有 at() 成员方法，和使用 [ ] 运算符一样，at() 成员方法也需要根据指定的键，才能从容器中找到该键对应的值；不同之处在于，如果在当前容器中查找失败，该方法不会向容器中添加新的键值对，而是直接抛出out\_of\_range异常。  
举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap{

{"Python教程","http://c.biancheng.net/python/"},

{"Java教程","http://c.biancheng.net/java/"},

{"Linux教程","http://c.biancheng.net/linux/"} };

//获取指定键对应的值

*string* str = umap.*at*("Python教程");

*cout* << str << *endl*;

//执行此语句会抛出 out\_of\_range 异常

//cout << umap.at("GO教程");

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/python/**

此程序中，第 13 行代码用于获取 umap 容器中键为“Python教程”对应的值，由于 umap 容器确实有符合条件的键值对，因此可以成功执行；而第 17 行代码，由于当前 umap 容器没有存储以“Go教程”为键的键值对，因此执行此语句会抛出 out\_of\_range 异常。  
  
3) [ ] 运算符和 at() 成员方法基本能满足大多数场景的需要。除此之外，还可以借助 unordered\_map 模板中提供的 find() 成员方法。  
 和前面方法不同的是，通过 find() 方法得到的是一个正向迭代器，该迭代器的指向分以下 2 种情况：

（1）当 find() 方法成功找到以指定元素作为键的键值对时，其返回的迭代器就指向该键值对；

（2）当 find() 方法查找失败时，其返回的迭代器和 end() 方法返回的迭代器一样，指向容器中最后一个键值对之后的位置。  
举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap{

{"Python教程","http://c.biancheng.net/python/"},

{"Java教程","http://c.biancheng.net/java/"},

{"Linux教程","http://c.biancheng.net/linux/"} };

//查找成功

*unordered\_map*<*string*, *string*>::*iterator* iter = umap.*find*("Python教程");

*cout* << iter->*first* << " " << iter->*second* << *endl*;

//查找失败

*unordered\_map*<*string*, *string*>::*iterator* iter2 = umap.*find*("GO教程");

if (iter2 == umap.*end*()) {

*cout* << "当前容器中没有以\"GO教程\"为键的键值对";

}

return 0;

}

**程序执行结果为：**

**Python教程 http://c.biancheng.net/python/  
当前容器中没有以"GO教程"为键的键值对**4) 除了 find() 成员方法之外，甚至可以借助 begin()/end() 或者 cbegin()/cend()，通过遍历整个容器中的键值对来找到目标键值对。  
  
举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap{

{"Python教程","http://c.biancheng.net/python/"},

{"Java教程","http://c.biancheng.net/java/"},

{"Linux教程","http://c.biancheng.net/linux/"} };

//遍历整个容器中存储的键值对

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

//判断当前的键值对是否就是要找的

if (!iter->*first*.*compare*("Java教程")) {

*cout* << iter->*second* << *endl*;

break;

}

}

return 0;

}

**程序执行结果为：**

**http://c.biancheng.net/java/**

**以上 4 种方法中，前 2 种方法基本能满足多数场景的需要，建议初学者首选 at() 成员方法！**

**第六C++ unordered\_map insert()用法精讲**

为了方便用户向已建 unordered\_map 容器中添加新的键值对，该容器模板中提供了 insert() 方法，本节就对此方法的用法做详细的讲解。  
 unordered\_map 模板类中，提供了多种语法格式的 insert() 方法，根据功能的不同，可划分为以下几种用法。  
1) insert() 方法可以将 pair 类型的键值对元素添加到 unordered\_map 容器中，其语法格式有 2 种：

**//以普通方式传递参数  
pair<iterator,bool> insert ( const value\_type& val );  
//以右值引用的方式传递参数  
template <class P>  
    pair<iterator,bool> insert ( P&& val );**

**有关右值引用，可阅读《[C++右值引用详解](http://c.biancheng.net/view/439.html" \t "http://c.biancheng.net/view/_blank)》一文，这里不再做具体解释。**

以上 2 种格式中，参数 val 表示要添加到容器中的目标键值对元素；该方法的返回值为 pair类型值，内部包含一个 iterator 迭代器和 bool 变量：

（1）当 insert() 将 val 成功添加到容器中时，返回的迭代器指向新添加的键值对，bool 值为 True；

（2）当 insert() 添加键值对失败时，意味着当前容器中本就存储有和要添加键值对的键相等的键值对，这种情况下，返回的迭代器将指向这个导致插入操作失败的迭代器，bool 值为 False。  
举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建空 umap 容器

*unordered\_map*<*string*, *string*> umap;

//构建要添加的键值对

*std*::*pair*<*string*, *string*>mypair("STL教程", "http://c.biancheng.net/stl/");

//创建接收 insert() 方法返回值的pair类型变量

*std*::*pair*<*unordered\_map*<*string*, *string*>::*iterator*, bool> ret;

//调用 insert() 方法的第一种语法格式

ret = umap.*insert*(mypair);

*cout* << "bool = " << ret.*second* << *endl*;

*cout* << "iter -> " << ret.*first*->*first* << " " << ret.*first*->*second* << *endl*;

//调用 insert() 方法的第二种语法格式

ret = umap.*insert*(*std*::*make\_pair*("Python教程", "http://c.biancheng.net/python/"));

*cout* << "bool = " << ret.*second* << *endl*;

*cout* << "iter -> " << ret.*first*->*first* << " " << ret.*first*->*second* << *endl*;

return 0;

}

**程序执行结果为：**

**bool = 1  
iter -> STL教程 http://c.biancheng.net/stl/  
bool = 1  
iter -> Python教程 http://c.biancheng.net/python/**

从输出结果很容易看出，两次添加键值对的操作，insert() 方法返回值中的 bool 变量都为 1，表示添加成功，此时返回的迭代器指向的是添加成功的键值对。  
  
2) 除此之外，insert() 方法还可以指定新键值对要添加到容器中的位置，其语法格式如下：

**//以普通方式传递 val 参数  
iterator insert ( const\_iterator hint, const value\_type& val );  
//以右值引用方法传递 val 参数  
template <class P>  
    iterator insert ( const\_iterator hint, P&& val );**

以上 2 种语法格式中，hint 参数为迭代器，用于指定新键值对要添加到容器中的位置；val 参数指的是要添加容器中的键值对；方法的返回值为迭代器：

（1）如果 insert() 方法成功添加键值对，该迭代器指向新添加的键值对；

（2）如果 insert() 方法添加键值对失败，则表示容器中本就包含有相同键的键值对，该方法返回的迭代器就指向容器中键相同的键值对；

**注意，以上 2 种语法格式中，虽然通过 hint 参数指定了新键值对添加到容器中的位置，但该键值对真正存储的位置，并不是 hint 参数说了算，最终的存储位置仍取决于该键值对的键的值。**

举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建空 umap 容器

*unordered\_map*<*string*, *string*> umap;

//构建要添加的键值对

*std*::*pair*<*string*, *string*>mypair("STL教程", "http://c.biancheng.net/stl/");

//创建接收 insert() 方法返回值的迭代器类型变量

*unordered\_map*<*string*, *string*>::*iterator* iter;

//调用第一种语法格式

iter = umap.*insert*(umap.*begin*(), mypair);

*cout* << "iter -> " << iter->*first* << " " << iter->*second* << *endl*;

//调用第二种语法格式

iter = umap.*insert*(umap.*begin*(), *std*::*make\_pair*("Python教程", "http://c.biancheng.net/python/"));

*cout* << "iter -> " << iter->*first* << " " << iter->*second* << *endl*;

return 0;

}

**程序输出结果为：**

**iter -> STL教程 http://c.biancheng.net/stl/  
iter -> Python教程 http://c.biancheng.net/python/**

3) insert() 方法还支持将某一个 unordered\_map 容器中指定区域内的所有键值对，复制到另一个 unordered\_map 容器中，其语法格式如下：

**template <class InputIterator>  
    void insert ( InputIterator first, InputIterator last );**

其中 first 和 last 都为迭代器，[first, last)表示复制其它 unordered\_map 容器中键值对的区域。  
  
举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建并初始化 umap 容器

*unordered\_map*<*string*, *string*> umap{ {"STL教程","http://c.biancheng.net/stl/"},

{"Python教程","http://c.biancheng.net/python/"},

{"Java教程","http://c.biancheng.net/java/"} };

//创建一个空的 unordered\_map 容器

*unordered\_map*<*string*, *string*> otherumap;

//指定要拷贝 umap 容器中键值对的范围

*unordered\_map*<*string*, *string*>::*iterator* first = ++umap.*begin*();

*unordered\_map*<*string*, *string*>::*iterator* last = umap.*end*();

//将指定 umap 容器中 [first,last) 区域内的键值对复制给 otherumap 容器

otherumap.*insert*(first, last);

//遍历 otherumap 容器中存储的键值对

for (auto iter = otherumap.*begin*(); iter != otherumap.*end*(); ++iter) {

*cout* << iter->first << " " << iter->*second* << *endl*;

}

return 0;

}

**程序输出结果为：**

**Python教程 http://c.biancheng.net/python/  
Java教程 http://c.biancheng.net/java/**

4) 除了以上 3 种方式，insert() 方法还支持一次向 unordered\_map 容器添加多个键值对，其语法格式如下：

void insert ( initializer\_list<value\_type> il );

其中，il 参数指的是可以用初始化列表的形式指定多个键值对元素。  
举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建空的 umap 容器

*unordered\_map*<*string*, *string*> umap;

//向 umap 容器同时添加多个键值对

umap.*insert*({ {"STL教程","http://c.biancheng.net/stl/"},

{"Python教程","http://c.biancheng.net/python/"},

{"Java教程","http://c.biancheng.net/java/"} });

//遍历输出 umap 容器中存储的键值对

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序输出结果为：**

**STL教程 http://c.biancheng.net/stl/  
Python教程 http://c.biancheng.net/python/  
Java教程 http://c.biancheng.net/java/**

**总的来说，unordered\_map 模板类提供的 insert() 方法，有以上 4 种用法，读者可以根据实际场景的需要自行选择使用哪一种。**

**第七C++ unordered\_map emplace()和emplace\_hint()方法**

和前面学的 map、set 等容器一样，C++ 11 标准也为 unordered\_map 容器新增了 emplace() 和 emplace\_hint() 成员方法，本节将对它们的用法做详细的介绍。  
 我们知道，实现向已有 unordered\_map 容器中添加新键值对，可以通过调用 insert() 方法，但其实还有更好的方法，即使用 emplace() 或者 emplace\_hint() 方法，它们完成“向容器中添加新键值对”的效率，要比 insert() 方法高。

**至于为什么 emplace()、emplace\_hint() 执行效率会比 insert() 方法高，可阅读《[为什么emplace()、emplace\_hint()执行效率比insert()高](http://c.biancheng.net/view/7183.html" \t "http://c.biancheng.net/view/_blank)》一文，虽然此文的讲解对象为 map 容器，但就这 3 个方法来说，unordered\_map 容器和 map 容器是一样的。**

**一unordered\_map emplace()方法**

emplace() 方法的用法很简单，其语法格式如下：

**template <class... Args>  
    pair<iterator, bool> emplace ( Args&&... args );**

其中，参数 args 表示可直接向该方法传递创建新键值对所需要的 2 个元素的值，其中第一个元素将作为键值对的键，另一个作为键值对的值。也就是说，该方法无需我们手动创建键值对，其内部会自行完成此工作。  
 另外需要注意的是，该方法的返回值为 pair 类型值，其包含一个迭代器和一个 bool 类型值：

（1）当 emplace() 成功添加新键值对时，返回的迭代器指向新添加的键值对，bool 值为 True；

（2）当 emplace() 添加新键值对失败时，说明容器中本就包含一个键相等的键值对，此时返回的迭代器指向的就是容器中键相同的这个键值对，bool 值为 False。

举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap;

//定义一个接受 emplace() 方法的 pair 类型变量

*pair*<*unordered\_map*<*string*, *string*>::*iterator*, bool> ret;

//调用 emplace() 方法

ret = umap.*emplace*("STL教程", "http://c.biancheng.net/stl/");

//输出 ret 中包含的 2 个元素的值

*cout* << "bool =" << ret.*second* << *endl*;

*cout* << "iter ->" << ret.*first*->*first* << " " << ret.*first*->*second* << *endl*;

return 0;

}

**程序执行结果为：**

**bool =1  
iter ->STL教程 http://c.biancheng.net/stl/**

**通过执行结果中 bool 变量的值为 1 可以得知，emplace() 方法成功将新键值对添加到了 umap 容器中。**

**二unordered\_map emplace\_hint()方法**

**emplace\_hint() 方法的语法格式如下：**

**template <class... Args>  
    iterator emplace\_hint ( const\_iterator position, Args&&... args );**

和 emplace() 方法相同，emplace\_hint() 方法内部会自行构造新键值对，因此我们只需向其传递构建该键值对所需的 2 个元素（第一个作为键，另一个作为值）即可。不同之处在于：

（1）emplace\_hint() 方法的返回值仅是一个迭代器，而不再是 pair 类型变量。当该方法将新键值对成功添加到容器中时，返回的迭代器指向新添加的键值对；反之，如果添加失败，该迭代器指向的是容器中和要添加键值对键相同的那个键值对。

（2）emplace\_hint() 方法还需要传递一个迭代器作为第一个参数，该迭代器表明将新键值对添加到容器中的位置。需要注意的是，新键值对添加到容器中的位置，并不是此迭代器说了算，最终仍取决于该键值对的键的值。

**可以这样理解，emplace\_hint() 方法中传入的迭代器，仅是给 unordered\_map 容器提供一个建议，并不一定会被容器采纳。**

举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap;

//定义一个接受 emplace\_hint() 方法的迭代器

*unordered\_map*<*string*, *string*>::*iterator* iter;

//调用 empalce\_hint() 方法

iter = umap.*emplace\_hint*(umap.*begin*(), "STL教程", "http://c.biancheng.net/stl/");

//输出 emplace\_hint() 返回迭代器 iter 指向的键值对的内容

*cout* << "iter ->" << iter->*first* << " " << iter->*second* << *endl*;

return 0;

}

**程序执行结果为：**

**iter ->STL教程 http://c.biancheng.net/stl/**

**第八C++ STL unordered\_map删除元素：erase()和clear()**

C++ STL 标准库为了方便用户可以随时删除 unordered\_map 容器中存储的键值对，unordered\_map 容器类模板中提供了以下 2 个成员方法：

（1）erase()：删除 unordered\_map 容器中指定的键值对；

（2）clear()：删除 unordered\_map 容器中所有的键值对，即清空容器。  
本节就对以上 2 个成员方法的用法做详细的讲解。

**一unordered\_map erase()方法**

为了满足不同场景删除 unordered\_map 容器中键值对的需要，此容器的类模板中提供了 3 种语法格式的 erase() 方法。  
  
1) erase() 方法可以接受一个正向迭代器，并删除该迭代器指向的键值对。该方法的语法格式如下：

**iterator erase ( const\_iterator position );**

其中 position 为指向容器中某个键值对的迭代器，该方法会返回一个指向被删除键值对之后位置的迭代器。

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap{

{"STL教程", "http://c.biancheng.net/stl/"},

{"Python教程", "http://c.biancheng.net/python/"},

{"Java教程", "http://c.biancheng.net/java/"} };

//输出 umap 容器中存储的键值对

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

*cout* << "erase:" << *endl*;

//定义一个接收 erase() 方法的迭代器

*unordered\_map*<*string*, *string*>::*iterator* ret;

//删除容器中第一个键值对

ret = umap.*erase*(umap.*begin*());

//输出 umap 容器中存储的键值对

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

*cout* << "ret = " << ret->*first* << " " << ret->*second* << *endl*;

return 0;

}

**程序执行结果为：**

**STL教程 http://c.biancheng.net/stl/  
Python教程 http://c.biancheng.net/python/  
Java教程 http://c.biancheng.net/java/  
erase:  
Python教程 http://c.biancheng.net/python/  
Java教程 http://c.biancheng.net/java/  
ret = Python教程 http://c.biancheng.net/python/**

可以看到，通过给 erase() 方法传入指向容器中第一个键值对的迭代器，该方法可以将容器中第一个键值对删除，同时返回一个指向被删除键值对之后位置的迭代器。

**注意，如果erase()方法删除的是容器存储的最后一个键值对，则该方法返回的迭代器，将指向容器中最后一个键值对之后的位置（等同于 end() 方法返回的迭代器）。**

2) 我们还可以直接将要删除键值对的键作为参数直接传给 erase() 方法，该方法会自行去 unordered\_map 容器中找和给定键相同的键值对，将其删除。erase() 方法的语法格式如下：

**size\_type erase ( const key\_type& k );**

其中，k 表示目标键值对的键的值；该方法会返回一个整数，其表示成功删除的键值对的数量。  
举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap{

{"STL教程", "http://c.biancheng.net/stl/"},

{"Python教程", "http://c.biancheng.net/python/"},

{"Java教程", "http://c.biancheng.net/java/"} };

//输出 umap 容器中存储的键值对

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

int delNum = umap.*erase*("Python教程");

*cout* << "delNum = " << delNum << *endl*;

//再次输出 umap 容器中存储的键值对

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**STL教程 http://c.biancheng.net/stl/  
Python教程 http://c.biancheng.net/python/  
Java教程 http://c.biancheng.net/java/  
delNum = 1  
STL教程 http://c.biancheng.net/stl/  
Java教程 http://c.biancheng.net/java/**

**通过输出结果可以看到，通过将 "Python教程" 传给 erase() 方法，就成功删除了 umap 容器中键为 "Python教程" 的键值对。**3) 除了支持删除 unordered\_map 容器中指定的某个键值对，erase() 方法还支持一次删除指定范围内的所有键值对，其语法格式如下：

iterator erase ( const\_iterator first, const\_iterator last );

其中 first 和 last 都是正向迭代器，[first, last) 范围内的所有键值对都会被 erase() 方法删除；同时，该方法会返回一个指向被删除的最后一个键值对之后一个位置的迭代器。  
  
举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap{

{"STL教程", "http://c.biancheng.net/stl/"},

{"Python教程", "http://c.biancheng.net/python/"},

{"Java教程", "http://c.biancheng.net/java/"} };

//first 指向第一个键值对

*unordered\_map*<*string*, *string*>::*iterator* first = umap.*begin*();

//last 指向最后一个键值对

*unordered\_map*<*string*, *string*>::*iterator* last = --umap.*end*();

//删除[fist,last)范围内的键值对

auto ret = umap.*erase*(first, last);

//输出 umap 容器中存储的键值对

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

*cout* << iter->first << " " << iter->*second* << *endl*;

}

*cout* << "ret -> " << ret->first << " " << ret->*second* << *endl*;

return 0;

}

**程序执行结果为：**

**Java教程 http://c.biancheng.net/java/  
ret -> Java教程 http://c.biancheng.net/java/**

可以看到，通过 erase() 方法删除 [first,last) 范围内所有的键值对之后，umap 容器仅剩最后一个键值对，同时该方法返回的迭代器指向的也是该键值对（它是被删除最后一个键值对之后的第一个键值对）。

**二unordered\_map clear()方法**

在个别场景中，可能需要一次性删除 unordered\_map 容器中存储的所有键值对，可以使用 clear() 方法，其语法格式如下：

**void clear()**

举个例子：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建 umap 容器

*unordered\_map*<*string*, *string*> umap{

{"STL教程", "http://c.biancheng.net/stl/"},

{"Python教程", "http://c.biancheng.net/python/"},

{"Java教程", "http://c.biancheng.net/java/"} };

//输出 umap 容器中存储的键值对

for (auto iter = umap.*begin*(); iter != umap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

//删除容器内所有键值对

umap.*clear*();

*cout* << "umap size = " << umap.*size*() << *endl*;

return 0;

}

**程序执行结果为：**

**STL教程 http://c.biancheng.net/stl/  
Python教程 http://c.biancheng.net/python/  
Java教程 http://c.biancheng.net/java/  
umap size = 0**

**显然，通过调用 clear() 方法，原本包含 3 个键值对的 umap 容器，变成了空容器。**

**注意，虽然使用 erase() 方法的第 3 种语法格式，可能实现删除 unordered\_map 容器内所有的键值对，但更推荐使用 clear() 方法。**

**第九C++ STL unordered\_multimap容器精讲**

C++ STL 标准库中，除了提供有 unordered\_map 无序关联容器，还提供有和 unordered\_map 容器非常相似的 unordered\_multimap 无序关联容器。  
 和 unordered\_map 容器一样，unordered\_multimap 容器也以键值对的形式存储数据，且底层也采用哈希表结构存储各个键值对。两者唯一的不同之处在于，unordered\_multimap 容器可以存储多个键相等的键值对，而 unordered\_map 容器不行。

**《深度剖析C++ STL无序容器底层原理》一文提到，无序容器中存储的各个键值对，都会哈希存到各个桶（本质为链表）中。而对于 unordered\_multimap 容器来说，其存储的所有键值对中，键相等的键值对会被哈希到同一个桶中存储。**

另外值得一提得是，STL 标准库中实现 unordered\_multimap 容器的模板类并没有定义在以自己名称命名的头文件中，而是和 unordered\_map 容器一样，定义在<unordered\_map>头文件，且位于 std 命名空间中。因此，在使用 unordered\_multimap 容器之前，程序中应包含如下 2 行代码：

#include <unordered\_map>

using namespace std;

注意，第二行代码不是必需的，但如果不用，则后续程序中在使用此容器时，需手动注明 std 命名空间（强烈建议初学者使用）。

**unordered\_multimap 容器模板的定义如下所示：**

**template < class Key, //键（key）的类型**

**class T, //值（value）的类型**

**class Hash = hash<Key>, //底层存储键值对时采用的哈希函数**

**class Pred = equal\_to<Key>, //判断各个键值对的键相等的规则**

**class Alloc = allocator< pair<const Key,T> > // 指定分配器对象的类型**

**> class unordered\_multimap;**

**以上 5 个参数中，必须显式给前 2 个参数传值，且除极个别的情况外，最多只使用前 4 个参数，它们各自的含义和功能如表 1 所示。**

表 1 unordered\_multimap 容器模板类的常用参数

|  |  |
| --- | --- |
| 参数 | 含义 |
| <key,T> | 前 2 个参数分别用于确定键值对中键和值的类型，也就是存储键值对的类型。 |
| Hash = hash<Key> | 用于指明容器在存储各个键值对时要使用的哈希函数，默认使用 STL 标准库提供的 hash<key> 哈希函数。注意，默认哈希函数只适用于基本数据类型（包括 string 类型），而不适用于自定义的结构体或者类。 |
| Pred = equal\_to<Key> | unordered\_multimap 容器可以存储多个键相等的键值对，而判断是否相等的规则，由此参数指定。默认情况下，使用 STL 标准库中提供的 equal\_to<key> 规则，该规则仅支持可直接用 == 运算符做比较的数据类型。 |

注意，当 unordered\_multimap 容器中存储键值对的键为自定义类型时，默认的哈希函数 hash<key> 以及比较函数 equal\_to<key> 将不再适用，这种情况下，需要我们自定义适用的哈希函数和比较函数，并分别显式传递给 Hash 参数和 Pred 参数。

关于给 unordered\_multimap 容器自定义哈希函数和比较函数的方法，后续章节会做详细讲解。

**一创建C++ unordered\_multimap容器的方法**

常见的创建 unordered\_map 容器的方法有以下几种。  
  
1) 利用 unordered\_multimap 容器类模板中的默认构造函数，可以创建空的 unordered\_multimap 容器。比如：

**std::unordered\_multimap<std::string, std::string>myummap;**

如果程序中已经默认指定了 std 命令空间，这里可以省略所有的 std::。

由此，就创建好了一个可存储 <string, string> 类型键值对的 unordered\_multimap 容器，只不过当前容器是空的，即没有存储任何键值对。  
2) 当然，在创建空 unordered\_multimap 容器的基础上，可以完成初始化操作。比如：

**unordered\_multimap<string, string>myummap{**

**{"Python教程","http://c.biancheng.net/python/"},**

**{"Java教程","http://c.biancheng.net/java/"},**

**{"Linux教程","http://c.biancheng.net/linux/"} };**

**通过此方法创建的 myummap 容器中，就包含有 3 个键值对。**  
  
3) 另外，unordered\_multimap 模板中还提供有复制（拷贝）构造函数，可以实现在创建 unordered\_multimap 容器的基础上，用另一 unordered\_multimap 容器中的键值对为其初始化。  
 例如，在第二种方式创建好 myummap 容器的基础上，再创建并初始化一个 myummap2 容器：

**unordered\_multimap<string, string>myummap2(myummap);**

由此，刚刚创建好的 myummap2 容器中，就包含有 myummap 容器中所有的键值对。  
除此之外，C++ 11 标准中还向 unordered\_multimap 模板类增加了移动构造函数，即以右值引用的方式将临时 unordered\_multimap 容器中存储的所有键值对，全部复制给新建容器。例如：

**//返回临时 unordered\_multimap 容器的函数**

**std::unordered\_multimap <std::string, std::string > retUmmap() {**

**std::unordered\_multimap<std::string, std::string>tempummap{**

**{"Python教程","http://c.biancheng.net/python/"},**

**{"Java教程","http://c.biancheng.net/java/"},**

**{"Linux教程","http://c.biancheng.net/linux/"} };**

**return tempummap;**

**}**

**//创建并初始化 myummap 容器**

**std::unordered\_multimap<std::string, std::string> myummap(retummap());**

**注意，无论是调用复制构造函数还是拷贝构造函数，必须保证 2 个容器的类型完全相同。**  
4) 当然，如果不想全部拷贝，可以使用 unordered\_multimap 类模板提供的迭代器，在现有 unordered\_multimap 容器中选择部分区域内的键值对，为新建 unordered\_multimap 容器初始化。例如：

**//传入 2 个迭代器，**

**std::unordered\_multimap<std::string, std::string> myummap2(++myummap.begin(), myummap.end());**

通过此方式创建的 myummap2 容器，其内部就包含 myummap 容器中除第 1 个键值对外的所有其它键值对。

**二C++ unordered\_multimap容器的成员方法**

和 unordered\_map 容器相比，unordered\_multimap 容器的类模板中没有重载 [ ] 运算符，也没有提供 at() 成员方法，除此之外它们完全一致。

**没有提供 [ ] 运算符和 at() 成员方法，意味着 unordered\_multimap 容器无法通过指定键获取该键对应的值，因为该容器允许存储多个键相等的键值对，每个指定的键可能对应多个不同的值。**

unordered\_multimap 类模板提供的成员方法如表 2 所示。

表 2 unordered\_multimap类模板成员方法

|  |  |
| --- | --- |
| 成员方法 | 功能 |
| begin() | 返回指向容器中第一个键值对的正向迭代器。 |
| end() | 返回指向容器中最后一个键值对之后位置的正向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过在其基础上增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。 |
| cend() | 和 end() 功能相同，只不过在其基础上，增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。 |
| empty() | 若容器为空，则返回 true；否则 false。 |
| size() | 返回当前容器中存有键值对的个数。 |
| max\_size() | 返回容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。 |
| find(key) | 查找以 key 为键的键值对，如果找到，则返回一个指向该键值对的正向迭代器；反之，则返回一个指向容器中最后一个键值对之后位置的迭代器（如果 end() 方法返回的迭代器）。 |
| count(key) | 在容器中查找以 key 键的键值对的个数。 |
| equal\_range(key) | 返回一个 pair 对象，其包含 2 个迭代器，用于表明当前容器中键为 key 的键值对所在的范围。 |
| emplace() | 向容器中添加新键值对，效率比 insert() 方法高。 |
| emplace\_hint() | 向容器中添加新键值对，效率比 insert() 方法高。 |
| insert() | 向容器中添加新键值对。 |
| erase() | 删除指定键值对。 |
| clear() | 清空容器，即删除容器中存储的所有键值对。 |
| swap() | 交换 2 个 unordered\_multimap 容器存储的键值对，前提是必须保证这 2 个容器的类型完全相等。 |
| bucket\_count() | 返回当前容器底层存储键值对时，使用桶（一个线性链表代表一个桶）的数量。 |
| max\_bucket\_count() | 返回当前系统中，unordered\_multimap 容器底层最多可以使用多少桶。 |
| bucket\_size(n) | 返回第 n 个桶中存储键值对的数量。 |
| bucket(key) | 返回以 key 为键的键值对所在桶的编号。 |
| load\_factor() | 返回 unordered\_multimap 容器中当前的负载因子。负载因子，指的是的当前容器中存储键值对的数量（size()）和使用桶数（bucket\_count()）的比值，即 load\_factor() = size() / bucket\_count()。 |
| max\_load\_factor() | 返回或者设置当前 unordered\_multimap 容器的负载因子。 |
| rehash(n) | 将当前容器底层使用桶的数量设置为 n。 |
| reserve() | 将存储桶的数量（也就是 bucket\_count() 方法的返回值）设置为至少容纳count个元（不超过最大负载因子）所需的数量，并重新整理容器。 |
| hash\_function() | 返回当前容器使用的哈希函数对象。 |

注意，对于实现互换 2 个相同类型 unordered\_multimap 容器的键值对，除了可以调用该容器模板类中提供的 swap() 成员方法外，STL 标准库还提供了同名的 swap() 非成员函数。  
下面的样例演示了表 2 中部分成员方法的用法：

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace *std*;

int *main*()

{

//创建空容器

*std*::*unordered\_multimap*<*std*::*string*, *std*::*string*> myummap;

//向空容器中连续添加 5 个键值对

myummap.*emplace*("Python教程", "http://c.biancheng.net/python/");

myummap.*emplace*("STL教程", "http://c.biancheng.net/stl/");

myummap.*emplace*("Java教程", "http://c.biancheng.net/java/");

myummap.*emplace*("C教程", "http://c.biancheng.net");

myummap.*emplace*("C教程", "http://c.biancheng.net/c/");

//输出 muummap 容器存储键值对的个数

*cout* << "myummmap size = " << myummap.*size*() << *endl*;

//利用迭代器输出容器中存储的所有键值对

for (auto iter = myummap.*begin*(); iter != myummap.*end*(); ++iter) {

*cout* << iter->*first* << " " << iter->*second* << *endl*;

}

return 0;

}

**程序执行结果为：**

**myummmap size = 5  
Python教程 http://c.biancheng.net/python/  
C教程 http://c.biancheng.net  
C教程 http://c.biancheng.net/c/  
STL教程 http://c.biancheng.net/stl/  
Java教程 http://c.biancheng.net/java/**

值得一提的是，unordered\_multimap 模板提供的所有成员方法的用法，都和 unordered\_map 提供的同名成员方法的用法完全相同（仅是调用者发生了改变），由于在讲解 unordered\_map 容器时，已经对大部分成员方法的用法做了详细的讲解，后续不再做重复性地赘述。

**第十C++ STL unordered\_set容器完全攻略**

我们知道，C++ 11 为 STL 标准库增添了 4 种无序（哈希）容器，前面已经对 unordered\_map 和 unordered\_multimap 容器做了详细的介绍，本节再讲解一种无序容器，即 unordered\_set 容器。

unordered\_set 容器，可直译为“无序 set 容器”，即 unordered\_set 容器和 set 容器很像，唯一的区别就在于 set 容器会自行对存储的数据进行排序，而 unordered\_set 容器不会。

总的来说，unordered\_set 容器具有以下几个特性：

（1）不再以键值对的形式存储数据，而是直接存储数据的值；

（2）容器内部存储的各个元素的值都互不相等，且不能被修改。

（3）不会对内部存储的数据进行排序（这和该容器底层采用哈希表结构存储数据有关，可阅读《C++ STL无序容器底层实现原理》一文做详细了解）；

**对于 unordered\_set 容器不以键值对的形式存储数据，读者也可以这样认为，即 unordered\_set 存储的都是键和值相等的键值对，为了节省存储空间，该类容器在实际存储时选择只存储每个键值对的值。**

另外，实现 unordered\_set 容器的模板类定义在<unordered\_set>头文件，并位于 std 命名空间中。这意味着，如果程序中需要使用该类型容器，则首先应该包含如下代码：

#include <unordered\_set>

using namespace std;

注意，第二行代码不是必需的，但如果不用，则程序中只要用到该容器时，必须手动注明 std 命名空间（强烈建议初学者使用）。

**unordered\_set 容器的类模板定义如下：**

**template < class Key, //容器中存储元素的类型**

**class Hash = hash<Key>, //确定元素存储位置所用的哈希函数**

**class Pred = equal\_to<Key>, //判断各个元素是否相等所用的函数**

**class Alloc = allocator<Key> //指定分配器对象的类型**

**> class unordered\_set;**

可以看到，以上 4 个参数中，只有第一个参数没有默认值，这意味着如果我们想创建一个 unordered\_set 容器，至少需要手动传递 1 个参数。事实上，在 99% 的实际场景中最多只需要使用前 3 个参数（各自含义如表 1 所示），最后一个参数保持默认值即可。

表 1 unordered\_set模板类定义

|  |  |
| --- | --- |
| **参数** | **含义** |
| Key | 确定容器存储元素的类型，如果读者将 unordered\_set 看做是存储键和值相同的键值对的容器，则此参数则用于确定各个键值对的键和值的类型，因为它们是完全相同的，因此一定是同一数据类型的数据。 |
| Hash = hash<Key> | 指定 unordered\_set 容器底层存储各个元素时，所使用的哈希函数。需要注意的是，默认哈希函数 hash<Key> 只适用于基本数据类型（包括 string 类型），而不适用于自定义的结构体或者类。 |
| Pred = equal\_to<Key> | unordered\_set 容器内部不能存储相等的元素，而衡量 2 个元素是否相等的标准，取决于该参数指定的函数。 默认情况下，使用 STL 标准库中提供的 equal\_to<key> 规则，该规则仅支持可直接用 == 运算符做比较的数据类型。 |

**注意，如果 unordered\_set 容器中存储的元素为自定义的数据类型，则默认的哈希函数 hash<key> 以及比较函数 equal\_to<key> 将不再适用，只能自己设计适用该类型的哈希函数和比较函数，并显式传递给 Hash 参数和 Pred 参数。至于如何实现自定义，后续章节会做详细讲解。**

**一创建C++ unordered\_set容器**

前面介绍了如何创建 unordered\_map 和 unordered\_multimap 容器，值得一提的是，创建它们的所有方式完全适用于 unordereded\_set 容器。不过，考虑到一些读者可能尚未学习其它无序容器，因此这里还是讲解一下创建 unordered\_set 容器的几种方法。

1) 通过调用 unordered\_set 模板类的默认构造函数，可以创建空的 unordered\_set 容器。比如：

**std::unordered\_set<std::string> uset;**

**如果程序已经引入了 std 命名空间，这里可以省略所有的 std::。**

由此，就创建好了一个可存储 string 类型值的 unordered\_set 容器，该容器底层采用默认的哈希函数 hash<Key> 和比较函数 equal\_to<Key>。

2) 当然，在创建 unordered\_set 容器的同时，可以完成初始化操作。比如：

**std::unordered\_set<std::string> uset{ "http://c.biancheng.net/c/",**

**"http://c.biancheng.net/java/",**

**"http://c.biancheng.net/linux/" };**

**通过此方法创建的 uset 容器中，就包含有 3 个 string 类型元素。**

3) 还可以调用 unordered\_set 模板中提供的复制（拷贝）构造函数，将现有 unordered\_set 容器中存储的元素全部用于为新建 unordered\_set 容器初始化。

**例如，在第二种方式创建好 uset 容器的基础上，再创建并初始化一个 uset2 容器：**

**std::unordered\_set<std::string> uset2(uset);**

**由此，umap2 容器中就包含有 umap 容器中所有的元素。**

除此之外，C++ 11 标准中还向 unordered\_set 模板类增加了移动构造函数，即以右值引用的方式，利用临时 unordered\_set 容器中存储的所有元素，给新建容器初始化。例如：

**//返回临时 unordered\_set 容器的函数**

**std::unordered\_set <std::string> retuset() {**

**std::unordered\_set<std::string> tempuset{ "http://c.biancheng.net/c/",**

**"http://c.biancheng.net/java/",**

**"http://c.biancheng.net/linux/" };**

**return tempuset;**

**}**

**//调用移动构造函数，创建 uset 容器**

**std::unordered\_set<std::string> uset(retuset());**

**注意，无论是调用复制构造函数还是拷贝构造函数，必须保证 2 个容器的类型完全相同。**

4) 当然，如果不想全部拷贝，可以使用 unordered\_set 类模板提供的迭代器，在现有 unordered\_set 容器中选择部分区域内的元素，为新建 unordered\_set 容器初始化。例如：

**//传入 2 个迭代器，**

**std::unordered\_set<std::string> uset2(++uset.begin(),uset.end());**

通过此方式创建的 uset2 容器，其内部就包含 uset 容器中除第 1 个元素外的所有其它元素。

**二C++ unordered\_set容器的成员方法**

unordered\_set 类模板中，提供了如表 2 所示的成员方法。

表 2 unordered\_set 类模板成员方法

|  |  |
| --- | --- |
| **成员方法** | **功能** |
| begin() | 返回指向容器中第一个元素的正向迭代器。 |
| end(); | 返回指向容器中最后一个元素之后位置的正向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过其返回的是 const 类型的正向迭代器。 |
| cend() | 和 end() 功能相同，只不过其返回的是 const 类型的正向迭代器。 |
| empty() | 若容器为空，则返回 true；否则 false。 |
| size() | 返回当前容器中存有元素的个数。 |
| max\_size() | 返回容器所能容纳元素的最大个数，不同的操作系统，其返回值亦不相同。 |
| find(key) | 查找以值为 key 的元素，如果找到，则返回一个指向该元素的正向迭代器；反之，则返回一个指向容器中最后一个元素之后位置的迭代器（如果 end() 方法返回的迭代器）。 |
| count(key) | 在容器中查找值为 key 的元素的个数。 |
| equal\_range(key) | 返回一个 pair 对象，其包含 2 个迭代器，用于表明当前容器中值为 key 的元素所在的范围。 |
| emplace() | 向容器中添加新元素，效率比 insert() 方法高。 |
| emplace\_hint() | 向容器中添加新元素，效率比 insert() 方法高。 |
| insert() | 向容器中添加新元素。 |
| erase() | 删除指定元素。 |
| clear() | 清空容器，即删除容器中存储的所有元素。 |
| swap() | 交换 2 个 unordered\_map 容器存储的元素，前提是必须保证这 2 个容器的类型完全相等。 |
| bucket\_count() | 返回当前容器底层存储元素时，使用桶（一个线性链表代表一个桶）的数量。 |
| max\_bucket\_count() | 返回当前系统中，unordered\_map 容器底层最多可以使用多少桶。 |
| bucket\_size(n) | 返回第 n 个桶中存储元素的数量。 |
| bucket(key) | 返回值为 key 的元素所在桶的编号。 |
| load\_factor() | 返回 unordered\_map 容器中当前的负载因子。负载因子，指的是的当前容器中存储元素的数量（size()）和使用桶数（bucket\_count()）的比值，即 load\_factor() = size() / bucket\_count()。 |
| max\_load\_factor() | 返回或者设置当前 unordered\_map 容器的负载因子。 |
| rehash(n) | 将当前容器底层使用桶的数量设置为 n。 |
| reserve() | 将存储桶的数量（也就是 bucket\_count() 方法的返回值）设置为至少容纳count个元（不超过最大负载因子）所需的数量，并重新整理容器。 |
| hash\_function() | 返回当前容器使用的哈希函数对象。 |

**注意，此容器模板类中没有重载 [ ] 运算符，也没有提供 at() 成员方法。不仅如此，由于 unordered\_set 容器内部存储的元素值不能被修改，因此无论使用那个迭代器方法获得的迭代器，都不能用于修改容器中元素的值。**

另外，对于实现互换 2 个相同类型 unordered\_set 容器的所有元素，除了调用表 2 中的 swap() 成员方法外，还可以使用 STL 标准库提供的 swap() 非成员函数，它们具有相同的名称，用法也相同（都只需要传入 2 个参数即可），仅是调用方式上有差别。

下面的样例演示了表 2 中部分成员方法的用法：

#include <iostream>

#include <string>

#include <unordered\_set>

using namespace *std*;

int *main*()

{

//创建一个空的unordered\_set容器

*std*::*unordered\_set*<*std*::*string*> uset;

//给 uset 容器添加数据

uset.*emplace*("http://c.biancheng.net/java/");

uset.*emplace*("http://c.biancheng.net/c/");

uset.*emplace*("http://c.biancheng.net/python/");

//查看当前 uset 容器存储元素的个数

*cout* << "uset size = " << uset.*size*() << *endl*;

//遍历输出 uset 容器存储的所有元素

for (auto iter = uset.*begin*(); iter != uset.*end*(); ++iter) {

*cout* << \*iter << *endl*;

}

return 0;

}

**程序执行结果为：**

**uset size = 3**

**http://c.biancheng.net/java/**

**http://c.biancheng.net/c/**

**http://c.biancheng.net/python/**

注意，表 2 中绝大多数成员方法的用法，都和 unordered\_map 容器提供的同名成员方法相同，读者可翻阅前面的文章做详细了解，当然也可以到 C++

STL标准库官网查询。

**第十一C++ STL unordered\_multiset容器详解**

前面章节详细地介绍了 unordered\_set 容器的特定和用法，在此基础上，本节再介绍一个类似的 C++ STL 无序容器，即 unordered\_multiset 容器。  
 所谓“类似”，指的是 unordered\_multiset 容器大部分的特性都和 unordered\_set 容器相同，包括：

（1）unordered\_multiset 不以键值对的形式存储数据，而是直接存储数据的值；

（2）该类型容器底层采用的也是哈希表存储结构（可阅读《[C++ STL无序容器底层实现原理](http://c.biancheng.net/view/7235.html" \t "http://c.biancheng.net/view/_blank)》一文做详细了解），它不会对内部存储的数据进行排序；

（3）unordered\_multiset 容器内部存储的元素，其值不能被修改。  
**和 unordered\_set 容器不同的是，unordered\_multiset 容器可以同时存储多个值相同的元素，且这些元素会存储到哈希表中同一个桶（本质就是链表）上。**

**读者可以这样认为，unordered\_multiset 除了能存储相同值的元素外，它和 unordered\_set 容器完全相同。**

另外值得一提的是，实现 unordered\_multiset 容器的模板类并没有定义在以该容器名命名的文件中，而是和 unordered\_set 容器共用同一个<unordered\_set>头文件，并且也位于 std 命名空间。因此，如果程序中需要使用该类型容器，应包含如下代码：

#include <unordered\_set>

using namespace std;

注意，第二行代码不是必需的，但如果不用，则程序中只要用到该容器时，必须手动注明 std 命名空间（强烈建议初学者使用）。

**unordered\_multiset 容器类模板的定义如下：**

**template < class Key, //容器中存储元素的类型**

**class Hash = hash<Key>, //确定元素存储位置所用的哈希函数**

**class Pred = equal\_to<Key>, //判断各个元素是否相等所用的函数**

**class Alloc = allocator<Key> //指定分配器对象的类型**

**> class unordered\_multiset;**

需要说明的是，在 99% 的实际场景中，最多只需要使用前 3 个参数（各自含义如表 1 所示），最后一个参数保持默认值即可。

表 1 unordered\_multiset 模板类定义

|  |  |
| --- | --- |
| 参数 | 含义 |
| Key | 确定容器存储元素的类型，如果读者将 unordered\_multiset 看做是存储键和值相同的键值对的容器，则此参数则用于确定各个键值对的键和值的类型，因为它们是完全相同的，因此一定是同一数据类型的数据。 |
| Hash = hash<Key> | 指定 unordered\_multiset 容器底层存储各个元素时所使用的哈希函数。需要注意的是，默认哈希函数 hash<Key> 只适用于基本数据类型（包括 string 类型），而不适用于自定义的结构体或者类。 |
| Pred = equal\_to<Key> | 用于指定 unordered\_multiset 容器判断元素值相等的规则。默认情况下，使用 STL 标准库中提供的 equal\_to<key> 规则，该规则仅支持可直接用 == 运算符做比较的数据类型。 |

**总之，如果 unordered\_multiset 容器中存储的元素为自定义的数据类型，则默认的哈希函数 hash<key> 以及比较函数 equal\_to<key> 将不再适用，只能自己设计适用该类型的哈希函数和比较函数，并显式传递给 Hash 参数和 Pred 参数。至于如何实现自定义，后续章节会做详细讲解。**

**一创建C++ unordered\_multiset容器**

考虑到不同场景的需要，unordered\_multiset 容器模板类共提供了以下 4 种创建 unordered\_multiset 容器的方式。  
  
1) 调用 unordered\_multiset 模板类的默认构造函数，可以创建空的 unordered\_multiset 容器。比如：

**std::unordered\_multiset<std::string> umset;**

如果程序已经引入了 std 命名空间，这里可以省略所有的 std::。

由此，就创建好了一个可存储 string 类型值的 unordered\_multiset 容器，该容器底层采用默认的哈希函数 hash<Key> 和比较函数 equal\_to<Key>。  
  
2) 当然，在创建 unordered\_multiset 容器的同时，可以进行初始化操作。比如：

std::unordered\_multiset<std::string> umset{ "http://c.biancheng.net/c/",

"http://c.biancheng.net/java/",

"http://c.biancheng.net/linux/" };

通过此方法创建的 umset 容器中，内部存有 3 个 string 类型元素。  
  
3) 还可以调用 unordered\_multiset 模板中提供的复制（拷贝）构造函数，将现有 unordered\_multiset 容器中存储的元素全部用于为新建 unordered\_multiset 容器初始化。  
  
例如，在第二种方式创建好 umset 容器的基础上，再创建并初始化一个 umset2 容器：

std::unordered\_multiset<std::string> umset2(umset);

由此，umap2 容器中就包含有 umap 容器中所有的元素。  
  
除此之外，C++ 11 标准中还向 unordered\_multiset 模板类增加了移动构造函数，即以右值引用的方式，利用临时 unordered\_multiset 容器中存储的所有元素，给新建容器初始化。例如：

//返回临时 unordered\_multiset 容器的函数

std::unordered\_multiset <std::string> retumset() {

std::unordered\_multiset<std::string> tempumset{ "http://c.biancheng.net/c/",

"http://c.biancheng.net/java/",

"http://c.biancheng.net/linux/" };

return tempumset;

}

//调用移动构造函数，创建 umset 容器

std::unordered\_multiset<std::string> umset(retumset());

注意，无论是调用复制构造函数还是拷贝构造函数，必须保证 2 个容器的类型完全相同。  
4) 当然，如果不想全部拷贝，可以使用 unordered\_multiset 类模板提供的迭代器，在现有 unordered\_multiset 容器中选择部分区域内的元素，为新建 unordered\_multiset 容器初始化。例如：

//传入 2 个迭代器，

std::unordered\_multiset<std::string> umset2(++umset.begin(), umset.end());

通过此方式创建的 umset2 容器，其内部就包含 umset 容器中除第 1 个元素外的所有其它元素。

**二C++ unordered\_multimap容器的成员方法**

值得一提的是，unordered\_multiset 模板类中提供的成员方法，无论是种类还是数量，都和 unordered\_set 类模板一样，如表 2 所示。

表 2 unordered\_set 类模板成员方法

|  |  |
| --- | --- |
| 成员方法 | 功能 |
| begin() | 返回指向容器中第一个元素的正向迭代器。 |
| end(); | 返回指向容器中最后一个元素之后位置的正向迭代器。 |
| cbegin() | 和 begin() 功能相同，只不过其返回的是 const 类型的正向迭代器。 |
| cend() | 和 end() 功能相同，只不过其返回的是 const 类型的正向迭代器。 |
| empty() | 若容器为空，则返回 true；否则 false。 |
| size() | 返回当前容器中存有元素的个数。 |
| max\_size() | 返回容器所能容纳元素的最大个数，不同的操作系统，其返回值亦不相同。 |
| find(key) | 查找以值为 key 的元素，如果找到，则返回一个指向该元素的正向迭代器；反之，则返回一个指向容器中最后一个元素之后位置的迭代器（如果 end() 方法返回的迭代器）。 |
| count(key) | 在容器中查找值为 key 的元素的个数。 |
| equal\_range(key) | 返回一个 pair 对象，其包含 2 个迭代器，用于表明当前容器中值为 key 的元素所在的范围。 |
| emplace() | 向容器中添加新元素，效率比 insert() 方法高。 |
| emplace\_hint() | 向容器中添加新元素，效率比 insert() 方法高。 |
| insert() | 向容器中添加新元素。 |
| erase() | 删除指定元素。 |
| clear() | 清空容器，即删除容器中存储的所有元素。 |
| swap() | 交换 2 个 unordered\_multimap 容器存储的元素，前提是必须保证这 2 个容器的类型完全相等。 |
| bucket\_count() | 返回当前容器底层存储元素时，使用桶（一个线性链表代表一个桶）的数量。 |
| max\_bucket\_count() | 返回当前系统中，容器底层最多可以使用多少桶。 |
| bucket\_size(n) | 返回第 n 个桶中存储元素的数量。 |
| bucket(key) | 返回值为 key 的元素所在桶的编号。 |
| load\_factor() | 返回容器当前的负载因子。所谓负载因子，指的是的当前容器中存储元素的数量（size()）和使用桶数（bucket\_count()）的比值，即 load\_factor() = size() / bucket\_count()。 |
| max\_load\_factor() | 返回或者设置当前 unordered\_map 容器的负载因子。 |
| rehash(n) | 将当前容器底层使用桶的数量设置为 n。 |
| reserve() | 将存储桶的数量（也就是 bucket\_count() 方法的返回值）设置为至少容纳count个元（不超过最大负载因子）所需的数量，并重新整理容器。 |
| hash\_function() | 返回当前容器使用的哈希函数对象。 |

注意，和 unordered\_set 容器一样，unordered\_multiset 模板类也没有重载 [ ] 运算符，没有提供 at() 成员方法。不仅如此，无论是由哪个成员方法返回的迭代器，都不能用于修改容器中元素的值。  
  
另外，对于互换 2 个相同类型 unordered\_multiset 容器存储的所有元素，除了调用表 2 中的 swap() 成员方法外，STL 标准库也提供了 swap() 非成员函数。  
  
下面的样例演示了表 2 中部分成员方法的用法：

#include <iostream>

#include <string>

#include <unordered\_set>

using namespace *std*;

int *main*()

{

//创建一个空的unordered\_multiset容器

*std*::*unordered\_multiset*<*std*::*string*> umset;

//给 uset 容器添加数据

umset.*emplace*("http://c.biancheng.net/java/");

umset.*emplace*("http://c.biancheng.net/c/");

umset.*emplace*("http://c.biancheng.net/python/");

umset.*emplace*("http://c.biancheng.net/c/");

//查看当前 umset 容器存储元素的个数

*cout* << "umset size = " << umset.*size*() << *endl*;

//遍历输出 umset 容器存储的所有元素

for (auto iter = umset.*begin*(); iter != umset.*end*(); ++iter) {

*cout* << \*iter << *endl*;

}

return 0;

}

程序执行结果为：

umset size = 4  
http://c.biancheng.net/java/  
http://c.biancheng.net/c/  
http://c.biancheng.net/c/  
http://c.biancheng.net/python/

注意，表 2 中绝大多数成员方法的用法，都和 unordered\_map 容器提供的同名成员方法相同，读者可翻阅前面的文章做详细了解，当然也可以到[C++ STL标准库](http://www.cplusplus.com/reference/unordered_set/unordered_multiset/" \t "http://c.biancheng.net/view/_blank)官网查询。

**第十二如何自定义C++ STL无序容器的哈希函数和比较规则？**

前面在讲解 unordered\_map、unordered\_multimap、unordered\_set 以及 unordered\_multiset 这 4 种无序关联式容器（哈希容器）时，遗留过一个共性问题，即如何给无序容器自定义一个哈希函数和比较规则？  
 注意，虽然每种无序容器都指定了默认的 hash<key> 哈希函数和 equal\_to<key> 比较规则，但它们仅适用于存储基本类型（比如 int、double、float、string 等）数据的无序容器。换句话说，如果无序容器存储的数据类型为自定义的结构体或类，则 STL 标准库提供的 hash<key> 和 equal\_to<key> 将不再适用。

**一C++无序容器自定义哈希函数**

我们知道，无序容器以键值对的方式存储数据（unordered\_set 和 unordered\_multiset 容器可以看做存储的是键和值相等的键值对），且底层采用哈希表结构存储各个键值对。在此存储结构中，哈希函数的功能是根据各个键值对中键的值，计算出一个哈希值（本质就是一个整数），哈希表可以根据该值判断出该键值对具体的存储位置。  
 简单地理解哈希函数，它可以接收一个元素，并通过内部对该元素做再加工，最终会得出一个整形值并反馈回来。需要注意的是，哈希函数只是一个称谓，其本体并不是普通的函数形式，而是一个函数对象类。因此，如果我们想自定义个哈希函数，就需要自定义一个函数对象类。

**关于什么函数对象类，可阅读《[C++函数对象详解](http://c.biancheng.net/view/354.html" \t "http://c.biancheng.net/view/_blank)》一节做详细了解，由于不是本节重点，这里不再赘述。**

举个例子，假设有如下一个 Person 类：

class Person {

public:

Person(string name, int age) :name(name), age(age) {};

string getName() const;

int getAge() const;

private:

string name;

int age;

};

string Person::getName() const {

return this->name;

}

int Person::getAge() const {

return this->age;

}

在此基础上，假设我们想创建一个可存储 Person 类对象的 unordered\_set 容器，考虑到 Person 为自定义的类型，因此默认的 hash<key> 哈希函数不再适用，这时就需要以函数对象类的方式自定义一个哈希函数。比如：

**class hash\_fun {**

**public:**

**int operator()(const Person &A) const {**

**return A.getAge();**

**}**

**};**

注意，重载 ( ) 运算符时，其参数必须为 const 类型，且该方法也必须用 const 修饰。

可以看到，我们利用 hash\_fun 函数对象类的 ( ) 运算符重载方法，自定义了适用于 Person 类对象的哈希函数。该哈希函数每接收一个 Person 类对象，都会返回该对象的 age 成员变量的值。

**事实上，默认的 hash<key> 哈希函数，其底层也是以函数对象类的形式实现的。**

由此，在创建存储 Person 类对象的 unordered\_set 容器时，可以将 hash\_fun 作为参数传递给该容器模板类中的 Pred 参数：

**std::unordered\_set<Person, hash\_fun> myset；**

但是，此时创建的 myset 容器还无法使用，因为该容器使用的是默认的 std::equal\_to<key> 比较规则，但此规则并不适用于该容器。

**二C++无序容器自定义比较规则**

和哈希函数一样，无论创建哪种无序容器，都需要为其指定一种可比较容器中各个元素是否相等的规则。  
 值得一提的是，默认情况下无序容器使用的 std::equal\_to<key> 比较规则，其本质也是一个函数对象类，底层实现如下：

**template<class T>**

**class equal\_to**

**{**

**public:**

**bool operator()(const T& \_Left, const T& \_Right) const{**

**return (\_Left == \_Right);**

**}**

**};**

可以看到，该规则在底层实现过程中，直接用 == 运算符比较容器中任意 2 个元素是否相等，这意味着，如果容器中存储的元素类型，支持直接用 == 运算符比较是否相等，则该容器可以使用默认的 std::equal\_to<key> 比较规则；反之，就不可以使用。  
 显然，对于我们上面创建的 myset 容器，其内部存储的是 Person 类对象，不支持直接使用 == 运算符做比较。这种情况下，有以下 2 种方式可以解决此问题：

（1）在 Person 类中重载 == 运算符，这会使得 std::equal\_to<key> 比较规则中使用的 == 运算符变得合法，myset 容器就可以继续使用 std::equal\_to<key> 比较规则；

（2）以函数对象类的方式，自定义一个适用于 myset 容器的比较规则。

**1) 重载==运算符**

如果选用第一种解决方式，仍以 Python 类为例，在此类的外部添加如下语句：

**bool operator==(const Person &A, const Person &B) {**

**return (A.getAge() == B.getAge());**

**}**

**注意，这里在重载 == 运算符时，2 个参数必须用 const 修饰。**

可以看到，通过此方式重载的运算符，当 std::equal\_to<key> 函数对象类中直接比较 2 个 Person 类对象时，实际上是在比较这 2 个对象的 age 成员变量是否相等。换句话说，此时的 std::equal\_to<key> 规则的含义为：只要 2 个 Person对象的 age 成员变量相等，就认为这 2 个 Person 对象是相等的。  
  
重载 == 运算符之后，就能以如下方式创建 myset 容器：

**std::unordered\_set<Person, hash\_fun> myset{ {"zhangsan", 40},{"zhangsan", 40},{"lisi", 40},{"lisi", 30} };**

**注意，虽然这里给 myset 容器初始化了 4 个 Person 对象，但由于比较规则以各个类对象的 age 值为准，myset 容器会认为前 3 个 Person 对象是相等的，因此最终 myset 容器只会存储 {"zhangsan", 40} 和 {"lisi", 30}。**

**2) 以函数对象类的方式自定义比较规则**

除此之外，还可以完全舍弃 std::equal\_to<key>，以函数对象类的方式自定义一个比较规则。比如：

**class mycmp {**

**public:**

**bool operator()(const Person &A, const Person &B) const {**

**return (A.getName() == B.getName()) && (A.getAge() == B.getAge());**

**}**

**};**

在 mycmp 规则的基础上，我们可以像如下这样创建 myset 容器：

**std::unordered\_set<Person, hash\_fun, mycmp> myset{ {"zhangsan", 40},{"zhangsan", 40},{"lisi", 40},{"lisi", 30} };**

由此创建的 myset 容器，虽然初始化了 4 个 Person 对象，但 myset 容器根据 mycmp 比较规则，可以识别出前 2 个是相等的，因此最终该容器内部存有  {"zhangsan", 40}、{"lisi", 40} 和 {"lisi", 30} 这 3 个 Person 对象。

**总结**

总的来说，当无序容器中存储的是基本类型（int、double、float、string）数据时，自定义哈希函数和比较规则，都只能以函数对象类的方式实现。  
 而当无序容器中存储的是用结构体或类自定义类型的数据时，自定义哈希函数的方式仍只有一种，即使用函数对象类的形式；而自定义比较规则的方式有两种，要么也以函数对象类的方式，要么仍使用默认的 std::equal\_to<key> 规则，但前提是必须重载 == 运算符。  
如下是本节的完整代码，读者可直接拷贝下来，加深对本节知识的理解：

#include <iostream>

#include <string>

#include <unordered\_set>

using namespace *std*;

class Person {

public:

Person(*string* name, int age) :name(name), age(age) {};

*string* getName() const;

int getAge() const;

private:

*string* name;

int age;

};

*string* Person::getName() const {

return this->name;

}

int Person::getAge() const {

return this->age;

}

//自定义哈希函数

class hash\_fun {

public:

int operator()(const Person& A) const {

return A.getAge();

}

};

//重载 == 运算符，myset 可以继续使用默认的 equal\_to<key> 规则

bool operator==(const Person& A, const Person& B) {

return (A.getAge() == B.getAge());

}

//完全自定义比较规则，弃用 equal\_to<key>

class mycmp {

public:

bool operator()(const Person& A, const Person& B) const {

return (A.getName() == B.getName()) && (A.getAge() == B.getAge());

}

};

int *main*()

{

//使用自定义的 hash\_fun 哈希函数，比较规则仍选择默认的 equal\_to<key>,前提是必须重载 == 运算符

*std*::*unordered\_set*<Person, hash\_fun> myset1{ {"zhangsan", 40},{"zhangsan", 40},{"lisi", 40},{"lisi", 30} };

//使用自定义的 hash\_fun 哈希函数，以及自定义的 mycmp 比较规则

*std*::*unordered\_set*<Person, hash\_fun, mycmp> myset2{ {"zhangsan", 40},{"zhangsan", 40},{"lisi", 40},{"lisi", 30} };

*cout* << "myset1:" << *endl*;

for (auto iter = myset1.*begin*(); iter != myset1.*end*(); ++iter) {

*cout* << iter->getName() << " " << iter->getAge() << *endl*;

}

*cout* << "myset2:" << *endl*;

for (auto iter = myset2.*begin*(); iter != myset2.*end*(); ++iter) {

*cout* << iter->getName() << " " << iter->getAge() << *endl*;

}

return 0;

}

**程序执行结果为：**

**myset1:  
zhangsan 40  
lisi 30  
myset2:  
lisi 40  
zhangsan 40  
lisi 30**

**第十三C++ STL容器这么多，怎样选出最适合的？**

到此为止，本教程已经讲解了 C++ STL 标准库中所有容器的特性、功能以及用法，但考虑到一些读者可能在纠结“什么场景中选用哪个容器”这个问题，本节将带领大家系统回顾一下所学的这些容器，并给出一个解决此问题的思路。  
 值得一提的是，虽然 STL 标准库还有迭代器、算法、函数对象等，但容器仍是大多数 C++ 程序员关注的焦点。首先，和普通数组相比，容器支持动态扩容和收缩，还可以自行管理存储的元素（例如排序），同时还提供有诸多成员方法，大大提高了开发效率等等。其次，每个容器的底层实现，都采用的是精心挑选的数据结构，这意味着在使用这些容器时，不用担心它们的执行效率。  
总的来说，C++ STL 标准库（以 C++ 11 为准）提供了以下几种容器供我们选择：

（1）序列式容器：array、vector、deque、list 和 forward\_list；

（2）关联式容器：map、multimap、set 和 multiset；

（3）无序关联式容器：unordered\_map、unordered\_multimap、unordered\_set 和 unordered\_multiset；

（4）容器适配器：stack、queue 和 priority\_queue。

**注意，容器适配器本质上也属于容器，关于以上各个容器适配器，后续章节会做详细讲解。**

上面是依据容器类型进行分类的。实际上，每个容器所具有的特性都和其底层选用的存储结构息息相关。根据容器底层采用的是连续的存储空间，还是分散的存储空间（以链表或者树作为存储结构），还可以将上面容器分为如下两类：

1、采用连续的存储空间：array、vector、deque；

2、采用分散的存储空间：list、forward\_list 以及所有的关联式容器和哈希容器。

**注意，这里将 deque 容器归为使用连续存储空间的这一类，是存在争议的。因为 deque 容器底层采用一段一段的连续空间存储元素，但是各段存储空间之间并不一定是紧挨着的。关于 deque 容器的底层存储结构（可阅读《[C++ STL deque底层实现原理](http://c.biancheng.net/view/vip_7714.html" \t "http://c.biancheng.net/view/_blank)》一节详细了解），读者理解即可，这里不必深究。**

既然 C++ STL 标准库提供了这么多种容器，在实际场景中我们应该如何选择呢？  
要想选择出适用于该特定场景的最佳容器，需要综合考虑多种实际因素，例如：

（1）是否需要在容器的指定位置插入新元素？如果需要，则只能选择序列式容器，而关联式容器和哈希容器是不行的；

（2）是否对容器中各元素的存储位置有要求？如果没有，则可以考虑使用哈希容器，反之就要避免使用哈希容器；

（3）是否需要使用指定类型的迭代器？举个例子，如果必须是随机访问迭代器，则只能选择 array、vector、deque；如果必须是双向迭代器，则可以考虑 list 序列式容器以及所有的关联式容器；如果必须是前向迭代器，则可以考虑 forward\_list 序列式容器以及所有的哈希容器；

（4）当发生新元素的插入或删除操作时，是否要避免移动容器中的其它元素？如果是，则要避开 array、vector、deque，选择其它容器；

（5）容器中查找元素的效率是否为关键的考虑因素？如果是，则应优先考虑哈希容器。

**当然，以上问题并没有涵盖所有的情形，只是起到一个抛砖引玉的作用。在实际场景中，我们需要考虑更多的因素（例如对比各个容器解决当前问题所需的时间复杂度），经过层层筛选，最终找到适合该场景的那个容器。**

# **第五章 C++（STL）容器适配器**

容器适配器是一个封装了序列容器的类模板，它在一般序列容器的基础上提供了一些不同的功能。之所以称作适配器类，是因为它可以通过适配容器现有的接口来提供不同的功能。  
本章将介绍 3 种容器适配器，分别是 stack、queue、priority\_queue：

（1）stack<T>：是一个封装了 deque<T> 容器的适配器类模板，默认实现的是一个后入先出（Last-In-First-Out，LIFO）的压入栈。stack<T> 模板定义在头文件 stack 中。

（2）queue<T>：是一个封装了 deque<T> 容器的适配器类模板，默认实现的是一个先入先出（First-In-First-Out，LIFO）的队列。可以为它指定一个符合确定条件的基础容器。queue<T> 模板定义在头文件 queue 中。

（3）priority\_queue<T>：是一个封装了 vector<T> 容器的适配器类模板，默认实现的是一个会对元素排序，从而保证最大元素总在队列最前面的队列。priority\_queue<T> 模板定义在头文件 queue 中。

适配器类在基础序列容器的基础上实现了一些自己的操作，显然也可以添加一些自己的操作。它们提供的优势是简化了公共接口，而且提高了代码的可读性。本章我们会详细地探讨这些适配器的应用。

**第一什么是适配器，C++ STL容器适配器详解**

在详解什么是容器适配器之前，初学者首先要理解适配器的含义。  
 其实，容器适配器中的“适配器”，和生活中常见的电源适配器中“适配器”的含义非常接近。我们知道，无论是电脑、手机还是其它电器，充电时都无法直接使用 220V 的交流电，为了方便用户使用，各个电器厂商都会提供一个适用于自己产品的电源线，它可以将 220V 的交流电转换成适合电器使用的低压直流电。  
 从用户的角度看，电源线扮演的角色就是将原本不适用的交流电变得适用，因此其又被称为电源适配器。  
再举一个例子，假设一个代码模块 A，它的构成如下所示：

**class A{**

**public:**

**void f1(){}**

**void f2(){}**

**void f3(){}**

**void f4(){}**

**};**

**现在我们需要设计一个模板 B，但发现，其实只需要组合一下模块 A 中的 f1()、f2()、f3()，就可以实现模板 B 需要的功能。其中 f1() 单独使用即可，而 f2() 和 f3() 需要组合起来使用，如下所示：**

**class B{**

**private:**

**A \* a;**

**public:**

**void g1(){**

**a->f1();**

**}**

**void g2(){**

**a->f2();**

**a->f3();**

**}**

**};**

可以看到，就如同是电源适配器将不适用的交流电变得适用一样，模板 B 将不适合直接拿来用的模板 A 变得适用了，因此我们可以将模板 B 称为 B 适配器。  
 容器适配器也是同样的道理，简单的理解容器适配器，其就是将不适用的序列式容器（包括 vector、deque 和 list）变得适用。容器适配器的底层实现和模板 A、B 的关系是完全相同的，即通过封装某个序列式容器，并重新组合该容器中包含的成员函数，使其满足某些特定场景的需要。

**容器适配器本质上还是容器，只不过此容器模板类的实现，利用了大量其它基础容器模板类中已经写好的成员函数。当然，如果必要的话，容器适配器中也可以自创新的成员函数。**

**需要注意的是，[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 中的容器适配器，其内部使用的基础容器并不是固定的，用户可以在满足特定条件的多个基础容器中自由选择。**

**一STL容器适配器的种类**

STL 提供了 3 种容器适配器，分别为 stack 栈适配器、queue 队列适配器以及 priority\_queue 优先权队列适配器。其中，各适配器所使用的默认基础容器以及可供用户选择的基础容器，如表 1 所示。

表 1 STL 容器适配器及其基础容器

|  |  |  |
| --- | --- | --- |
| 容器适配器 | 基础容器筛选条件 | 默认使用的基础容器 |
| stack | 基础容器需包含以下成员函数：  empty()  size()  back()  push\_back()  pop\_back()  满足条件的基础容器有 vector、deque、list。 | deque |
| queue | 基础容器需包含以下成员函数：  empty()  size()  front()  back()  push\_back()  pop\_front()  满足条件的基础容器有 deque、list。 | deque |
| priority\_queue | 基础容器需包含以下成员函数：  empty()  size()  front()  push\_back()  pop\_back()  满足条件的基础容器有vector、deque。 | vector |

**不同场景下，由于不同的序列式容器其底层采用的[数据结构](http://c.biancheng.net/data_structure/" \t "http://c.biancheng.net/view/_blank)不同，因此容器适配器的执行效率也不尽相同。但通常情况下，使用默认的基础容器即可。当然，我们也可以手动修改，具体的修改容器适配器基础容器的方法，后续讲解具体的容器适配器会详细介绍。**

**第二 C++ stack（STL stack）容器适配器用法详解**

stack 栈适配器是一种单端开口的容器（如图 1 所示），实际上该容器模拟的就是栈存储结构，即无论是向里存数据还是从中取数据，都只能从这一个开口实现操作。

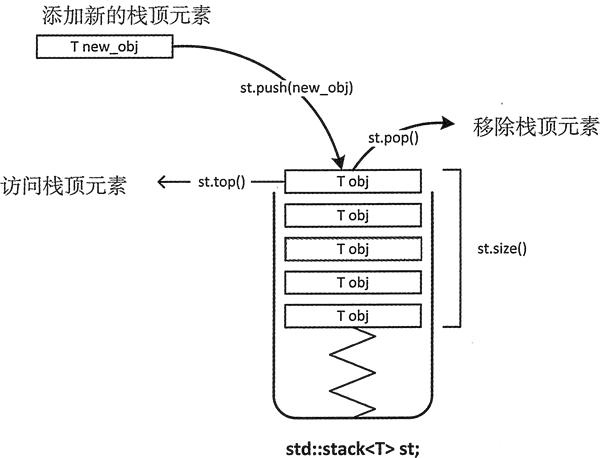


图 1 stack 适配器示意图

如图 1 所示，stack 适配器的开头端通常称为栈顶。由于数据的存和取只能从栈顶处进行操作，因此对于存取数据，stack 适配器有这样的特性，即每次只能访问适配器中位于最顶端的元素，也只有移除 stack 顶部的元素之后，才能访问位于栈中的元素。

**栈中存储的元素满足“后进先出（简称LIFO）”的准则，stack 适配器也同样遵循这一准则。**

**一stack容器适配器的创建**

由于 stack 适配器以模板类 stack<T,Container=deque<T>>（其中 T 为存储元素的类型，Container 表示底层容器的类型）的形式位于<stack>头文件中，并定义在 std 命名空间里。因此，在创建该容器之前，程序中应包含以下 2 行代码：

#include <stack>

u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;

std 命名空间也可以在使用 stack 适配器时额外注明。

创建 stack 适配器，大致分为如下几种方式。  
1) 创建一个不包含任何元素的 stack 适配器，并采用默认的 deque 基础容器：

**std::stack<int> values;**

**上面这行代码，就成功创建了一个可存储 int 类型元素，底层采用 deque 基础容器的 stack 适配器。**2) 上面提到，stack<T,Container=deque<T>> 模板类提供了 2 个参数，通过指定第二个模板类型参数，我们可以使用出 deque 容器外的其它序列式容器，只要该容器支持 empty()、size()、back()、push\_back()、pop\_back() 这 5 个成员函数即可。  
 在介绍适配器时提到，序列式容器中同时包含这 5 个成员函数的，有 vector、deque 和 list 这 3 个容器。因此，stack 适配器的基础容器可以是它们 3 个中任何一个。例如，下面展示了如何定义一个使用 list 基础容器的 stack 适配器：

**std::stack<std::string, std::list<int>> values;**  
3) 可以用一个基础容器来初始化 stack 适配器，只要该容器的类型和 stack 底层使用的基础容器类型相同即可。例如：

**std::list<int> values {1, 2, 3};**

**std::stack<int,std::list<int>> my\_stack (values);**

**注意，初始化后的 my\_stack 适配器中，栈顶元素为 3，而不是 1。另外在第 2 行代码中，stack 第 2 个模板参数必须显式指定为 list<int>（必须为 int 类型，和存储类型保持一致），否则 stack 底层将默认使用 deque 容器，也就无法用 lsit 容器的内容来初始化 stack 适配器。**  
4) 还可以用一个 stack 适配器来初始化另一个 stack 适配器，只要它们存储的元素类型以及底层采用的基础容器类型相同即可。例如：

**std::list<int> values{ 1, 2, 3 };**

**std::stack<int, std::list<int>> my\_stack1(values);**

**std::stack<int, std::list<int>> my\_stack=my\_stack1;**

**//std::stack<int, std::list<int>> my\_stack(my\_stack1);**  
可以看到，和使用基础容器不同，使用 stack 适配器给另一个 stack 进行初始化时，有 2 种方式，使用哪一种都可以。

注意，第 3、4 种初始化方法中，my\_stack 适配器的数据是经过拷贝得来的，也就是说，操作 my\_stack 适配器，并不会对 values 容器以及 my\_stack1 适配器有任何影响；反过来也是如此。

**二stack容器适配器支持的成员函数**

和其他序列容器相比，stack 是一类存储机制简单、提供成员函数较少的容器。表 1 列出了 stack 容器支持的全部成员函数。

表 1 stack容器适配器支持的成员函数

|  |  |
| --- | --- |
| 成员函数 | 功能 |
| empty() | 当 stack 栈中没有元素时，该成员函数返回 true；反之，返回 false。 |
| size() | 返回 stack 栈中存储元素的个数。 |
| top() | 返回一个栈顶元素的引用，类型为 T&。如果栈为空，程序会报错。 |
| push(const T& val) | 先复制 val，再将 val 副本压入栈顶。这是通过调用底层容器的 push\_back() 函数完成的。 |
| push(T&& obj) | 以移动元素的方式将其压入栈顶。这是通过调用底层容器的有右值引用参数的 push\_back() 函数完成的。 |
| pop() | 弹出栈顶元素。 |
| emplace(arg...) | arg... 可以是一个参数，也可以是多个参数，但它们都只用于构造一个对象，并在栈顶直接生成该对象，作为新的栈顶元素。 |
| swap(stack<T> & other\_stack) | 将两个 stack 适配器中的元素进行互换，需要注意的是，进行互换的 2 个 stack 适配器中存储的元素类型以及底层采用的基础容器类型，都必须相同。 |

下面这个例子中演示了表 1 中部分成员函数的用法：

#include <iostream>

#include <stack>

#include <list>

using namespace *std*;

int *main*()

{

//构建 stack 容器适配器

*list*<int> values{ 1, 2, 3 };

*stack*<int, *list*<int>> my\_stack(values);

//查看 my\_stack 存储元素的个数

*cout* << "size of my\_stack: " << my\_stack.*size*() << *endl*;

//将 my\_stack 中存储的元素依次弹栈，直到其为空

while (!my\_stack.*empty*())

{

*cout* << my\_stack.*top*() << *endl*;

//将栈顶元素弹栈

my\_stack.*pop*();

}

return 0;

}

**运行结果为：**

**size of my\_stack: 3  
3  
2  
1**

**表 1 中其它成员函数的用法也非常简单，这里不再给出具体示例，后续章节用法会做具体介绍。**

**第三stack容器适配器实现计算器（含实现代码）**

前面章节中，已经对 stack 容器适配器及其用法做了详细的讲解。本节将利用 stack 适配器实现一个简单的计算机程序，此计算机支持基本的加（+）、 减（-）、乘（\*）、除（/）、幂（^）运算。  
这里，先给大家展示出完整的实现代码，读者可先自行思考该程序的实现流程。当然，后续也会详细的讲解：

#include <iostream>

#include <cmath> // pow()

#include <stack> // stack<T>

#include <algorithm> // remove()

#include <stdexcept> // runtime\_error

#include <string> // string

using *std*::*string*;

// 返回运算符的优先级，值越大，优先级越高

inline *size\_t* precedence(const char op)

{

if (op == '+' || op == '-')

return 1;

if (op == '\*' || op == '/')

return 2;

if (op == '^')

return 3;

throw *std*::*runtime\_error*{ *string* {"表达中包含无效的运算符"} + op };

}

// 计算

double execute(*std*::*stack*<char>& ops, *std*::*stack*<double>& operands)

{

double result{};

double rhs{ operands.*top*() }; // 得到右操作数

operands.*pop*();

double lhs{ operands.*top*() }; // 得到做操作数

operands.*pop*();

switch (ops.*top*()) // 根据两个操作数之间的运算符，执行相应计算

{

case '+':

result = lhs + rhs;

break;

case '-':

result = lhs - rhs;

break;

case '\*':

result = lhs \* rhs;

break;

case '/':

result = lhs / rhs;

break;

case '^':

result = *std*::*pow*(lhs, rhs);

break;

default:

throw *std*::*runtime\_error*{ *string*{"invalid operator: "} + ops.*top*() };

}

ops.*pop*(); //计算完成后，该运算符要弹栈

operands.*push*(result);//将新计算出来的结果入栈

return result;

}

int *main*()

{

*std*::*stack*<double> operands; //存储表达式中的运算符

*std*::*stack*<char> operators; //存储表达式中的数值

*string* exp; //接受用户输入的表达式文本

try

{

while (true)

{

*std*::*cout* << "输入表达式(按Enter结束):" << *std*::*endl*;

*std*::*getline*(*std*::*cin*, exp, '\n');

if (exp.*empty*()) break;

//移除用户输入表达式中包含的无用的空格

exp.*erase*(*std*::*remove*(*std*::*begin*(exp), *std*::*end*(exp), ' '), *std*::*end*(exp));

*size\_t* index{};

//每个表达式必须以数字开头,index表示该数字的位数

operands.*push*(*std*::*stod*(exp, &index)); // 将表达式中第一个数字进栈

*std*::*cout* << index << *std*::*endl*;

while (true)

{

operators.*push*(exp[index++]); // 将运算符进栈

*size\_t* i{};

operands.*push*(*std*::*stod*(exp.*substr*(index), &i)); //将运算符后的数字也进栈,并将数字的位数赋值给 i。

index += i; //更新 index

if (index == exp.*length*())

{

while (!operators.*empty*()) //如果 operators不为空，表示还没有计算完

execute(operators, operands);

break;

}

//如果表达式还未遍历完，但子表达式中的运算符优先级比其后面的运算符优先级大，就先计算当前的子表达式的值

while (!operators.*empty*() && precedence(exp[index]) <= precedence(operators.*top*()))

execute(operators, operands);

}

*std*::*cout* << "result = " << operands.*top*() << *std*::*endl*;

}

}

catch (const *std*::*exception*& e)

{

*std*::*cerr* << e.*what*() << *std*::*endl*;

}

*std*::*cout* << "计算结束" << *std*::*endl*;

return 0;

}

**下面是一些示例输出：**

**输入表达式(按Enter结束):  
5\*2-3  
result = 7  
输入表达式(按Enter结束):  
4+4\*2  
result = 12  
输入表达式(按Enter结束):↙   <--键入Enter  
计算结束**

**一 计算器程序的实现流程**

了解一个程序的功能，通常是从 main() 函数开始。因此，下面从 main() 函数开始，给大家讲解程序的整个实现过程。  
 首先，我们创建 2 个 stack 适配器，operands 负责将表达式中的运算符逐个压栈，operators 负责将表达式的数值逐个压栈，同时还需要一个 string 类型的 exp，用于接收用户输入的表达式。  
 正如上面代码中所有看到的，所有的实现代码都包含在一个由 try 代码块包裹着的 while 循环中，这样既可以实现用户可以多次输入表达式的功能（当输入的表达式为一个空字符串时，循环结束），还可以捕获程序运行过程中抛出的任何异常（在 catch 代码块中，调用异常对象的成员函数 what() 会将错误信息输出到标准错误流中）。  
 当用户输入完要计算的表达式之后，由于整个表达式是以字符串的形式接收的，考虑到字符串中可能掺杂空格，影响后续对字符串的处理，因此又必须借助 remove() 函数来移除输入表达式中的多余空格（第 70 行代码处）。  
 得到统一格式的表达式之后，接下来才是实现计算功能的核心，其实现思路为：  
1) 因为所有的运算符都需要两个操作数，所以有效的输入表达式格式为“操作数 运算符 操作数 运算符 操作数...”，即序列的第一个和最后一个元素肯定都是操作数，每对操作数之间有一个运算符。由于有效表达式总是以操作数开头，所以第一个操作数在分析表达式的嵌套循环之前被提取出来。  
2) 在循环中，输入字符串的运算符会被压入 operators 栈。在确认没有到达字符串末尾后，再从 exp 提取第二个操作数。这时 stod() 的第一个参数是从 index 开始的 exp 字符串，它是被压入 operators 栈的运算符后的所有字符。此时字符串中第一个运算符的索引为 i，因为 i 是相对于 index 的，所以我们会将 index 加上 i 的值，使它指向操作数后的一个运算符（如果是 exp 中的最后一个操作数，它会指向字符串末尾的下一个位置）。  
3) 当 index 的值超过 exp 的最后一个字符时，会执行 operators 容器中剩下的运算符。如果没有到达字符串末尾，operators 容器也不为空，我们会比较 operators 栈顶运算符和 exp 中下一个运算符的优先级。如果栈顶运算符的优先级高于下一个运算符，就先执行栈顶的运算符。否则，就不执行栈顶运算符，在下一次循环开始时，将下一个运算符压入 operators 栈。通过这种方式，就可以正确计算出带优先级的表达式的值。  
以“5-2\*3+1”为例，以上程序的计算过程如下：  
1) 取  5 和 2 进 operands 栈容器，同时它们之间的 - 运算符进 operators 栈容器，判断后续是否还有表达式，显然还有“\*3+1”，这种情况下，取 operators 栈顶运算符 - 和后续的 \* 运算符做优先级比较，由于 \* 的优先级更高，此时继续将后续的 \* 和 3 分别进栈；

**此时，operands 中从栈顶依次存储的是 3、2、5，operators 容器中从栈顶依次存储的是 \*、-。**

2) 继续判断后续是否还有表达式，由于还有“+1”，则取 operators 栈顶运算符 \* 和 + 运算符做优先级比较，显然前者的优先级更高，此时将 operands 栈顶的 2 个元素（2 和 3）取出并弹栈，同时将 operators 栈顶元素（\*）取出并弹栈，计算它们组成的表达式 2\*3，并将计算结果再入 operands 栈。

**计算到这里，operands 中从栈顶依次存储的是 6、5，operators 中从栈顶依次存储的是 -。**

3) 由于 operator 容器不空，因此继续取新的栈顶运算符“-”和“+”做优先级比较，由于它们的优先级是相同的，因为继续将 operands 栈顶的 2 个元素（5 和 6）取出并弹栈，同时将 operators 栈顶元素（-） 取出并弹栈，计算它们组成的表达式“5-6”，并将计算结果 -1 再入 operands 栈。

**此时，operands 中从栈顶依次存储的是 -1，operator 为空。**

4）由于此时 operator 栈为空，因此将后续“+1”表达式中的 1 和 + 分别进栈。由于后续再无其他表达式，此时就可以直接取 operands 位于栈顶的 2 个元素（-1 和 1），和 operator 的栈顶运算符（+），执行 -1+1 运算，并将计算结果再入 operands 栈。  
  
**通过以上几步，最终“5-2\*3+1”的计算结果 0 位于 operands 的栈顶。**

**第四C++ STL queue容器适配器详解**

和 stack 栈容器适配器不同，queue 容器适配器有 2 个开口，其中一个开口专门用来输入数据，另一个专门用来输出数据，如图 1 所示。

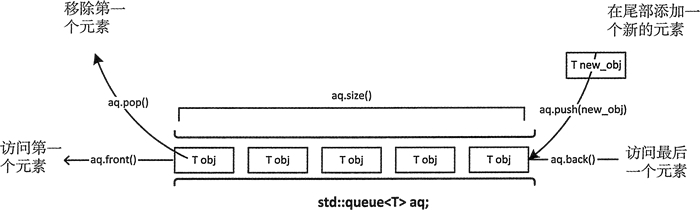


图 1 queue容器适配器

这种存储结构最大的特点是，最先进入 queue 的元素，也可以最先从 queue 中出来，即用此容器适配器存储数据具有“先进先出（简称 "FIFO" ）”的特点，因此 queue 又称为队列适配器。

**其实，[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) queue 容器适配器模拟的就是队列这种存储结构，因此对于任何需要用队列进行处理的序列来说，使用 queue 容器适配器都是好的选择。**

**一queue容器适配器的创建**

queue 容器适配器以模板类 queue<T,Container=deque<T>>（其中 T 为存储元素的类型，Container 表示底层容器的类型）的形式位于<queue>头文件中，并定义在 std 命名空间里。因此，在创建该容器之前，程序中应包含以下 2 行代码：

#include <queue>

u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;

创建 queue 容器适配器的方式大致可分为以下几种。  
1) 创建一个空的 queue 容器适配器，其底层使用的基础容器选择默认的 deque 容器：

std::queue<int> values;

通过此行代码，就可以成功创建一个可存储 int 类型元素，底层采用 deque 容器的 queue 容器适配器。  
2) 当然，也可以手动指定 queue 容器适配器底层采用的基础容器类型。通过学习 《[STL容器适配器详解](http://c.biancheng.net/view/6967.html" \t "http://c.biancheng.net/view/_blank)》一节我们知道，queue 容器适配器底层容器可以选择 deque 和 list。

**作为 queue 容器适配器的基础容器，其必须提供 front()、back()、push\_back()、pop\_front()、empty() 和 size() 这几个成员函数，符合条件的序列式容器仅有 deque 和 list。**

例如，下面创建了一个使用 list 容器作为基础容器的空 queue 容器适配器：

std::queue<int, std::list<int>> values;

注意，在手动指定基础容器的类型时，其存储的数据类型必须和 queue 容器适配器存储的元素类型保持一致。  
3) 可以用基础容器来初始化 queue 容器适配器，只要该容器类型和 queue 底层使用的基础容器类型相同即可。例如：

**std::deque<int> values{1,2,3};**

**std::queue<int> my\_queue(values);**

**由于 my\_queue 底层采用的是 deque 容器，和 values 类型一致，且存储的也都是 int 类型元素，因此可以用 values 对 my\_queue 进行初始化。**  
4) 还可以直接通过 queue 容器适配器来初始化另一个 queue 容器适配器，只要它们存储的元素类型以及底层采用的基础容器类型相同即可。例如：

**std::deque<int> values{1,2,3};**

**std::queue<int> my\_queue1(values);**

**std::queue<int> my\_queue(my\_queue1);**

**//或者使用**

**//std::queue<int> my\_queue = my\_queue1;**

注意，和使用基础容器不同，使用 queue 适配器给另一个 queue 进行初始化时，有 2 种方式，使用哪一种都可以。

**值得一提的是，第 3、4 种初始化方法中 my\_queue 容器适配器的数据是经过拷贝得来的，也就是说，操作 my\_queue 容器适配器中的数据，并不会对 values 容器以及 my\_queue1 容器适配器有任何影响；反过来也是如此。**

**queue容器适配器支持的成员函数**

**queue 容器适配器和 stack 有一些成员函数相似，但在一些情况下，工作方式有些不同。表 2 罗列了 queue 容器支持的全部成员函数。**

表 2 queue容器适配器支持的成员函数

|  |  |
| --- | --- |
| 成员函数 | 功能 |
| empty() | 如果 queue 中没有元素的话，返回 true。 |
| size() | 返回 queue 中元素的个数。 |
| front() | 返回 queue 中第一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。 |
| back() | 返回 queue 中最后一个元素的引用。如果 queue 是常量，就返回一个常引用；如果 queue 为空，返回值是未定义的。 |
| push(const T& obj) | 在 queue 的尾部添加一个元素的副本。这是通过调用底层容器的成员函数 push\_back() 来完成的。 |
| emplace() | 在 queue 的尾部直接添加一个元素。 |
| push(T&& obj) | 以移动的方式在 queue 的尾部添加元素。这是通过调用底层容器的具有右值引用参数的成员函数 push\_back() 来完成的。 |
| pop() | 删除 queue 中的第一个元素。 |
| swap(queue<T> &other\_queue) | 将两个 queue 容器适配器中的元素进行互换，需要注意的是，进行互换的 2 个 queue 容器适配器中存储的元素类型以及底层采用的基础容器类型，都必须相同。 |

和 stack 一样，queue 也没有迭代器，因此访问元素的唯一方式是遍历容器，通过不断移除访问过的元素，去访问下一个元素。

下面这个例子中演示了表 2 中部分成员函数的用法：

#include <iostream>

#include <queue>

#include <list>

using namespace *std*;

int *main*()

{

//构建 queue 容器适配器

*std*::*deque*<int> values{ 1,2,3 };

*std*::*queue*<int> my\_queue(values);//{1,2,3}

//查看 my\_queue 存储元素的个数

*cout* << "size of my\_queue: " << my\_queue.*size*() << *endl*;

//访问 my\_queue 中的元素

while (!my\_queue.*empty*())

{

*cout* << my\_queue.*front*() << *endl*;

//访问过的元素出队列

my\_queue.*pop*();

}

return 0;

}

**运行结果为：**

**size of my\_queue: 3  
1  
2  
3**

表 2 中其它成员函数的用法也非常简单，这里不再给出具体示例，后续章节用法会做具体介绍。

**第五C++ queue容器适配器模拟超市结账环节**

前面章节介绍了 queue 容器适配器的具有用法，本节将利用 queue 模拟超市中结账环节运转的程序。  
 在超市营业过程中，结账队列的长度是超市运转的关键因素。它会影响超市可容纳的顾客数，因为太长的队伍会使顾客感到气馁，从而放弃排队，这和医院可用病床数会严重影响应急处理设施的运转，是同样的道理。  
 首先，我们要在头文件 Customer.h 中定义一个类来模拟顾客：

#ifndef CUSTOMER\_H

#define CUSTOMER\_H

class Customer

{

protected:

*size\_t* service\_t{}; //顾客结账需要的时间

public:

explicit Customer(*size\_t* st = 10) :service\_t{ st } {}

//模拟随着时间的变化，顾客结账所需时间也会减短

Customer& time\_decrement()

{

if (service\_t > 0)

--service\_t;

return \*this;

}

bool done() const { return service\_t == 0; }

};

#endif

这里只有一个成员变量 service\_t，用来记录顾客结账需要的时间。每个顾客的结账时间都不同。每过一分钟，会调用一次 time\_decrement() 函数，这个函数会减少 service\_t 的值，它可以反映顾客结账所花费的时间。当 service\_t 的值为 0 时，成员函数 done() 返回 true。  
 超市的每个结账柜台都有一队排队等待的顾客。Checkout.h 中定义的 Checkout 类如下：

#ifndef CHECKOUT\_H

#define CHECKOUT\_H

#include <queue> // For queue container

#include "Customer.h"

class Checkout

{

private:

*std*::*queue*<Customer> customers; //该队列等到结账的顾客数量

public:

void add(const Customer& customer) { customers.*push*(customer); }

*size\_t* qlength() const { return customers.*size*(); }

void time\_increment()

{

if (!customers.*empty*())

{

//有顾客正在等待结账，如果顾客结账了，就出队列

if (customers.*front*().time\_decrement().*done*())

customers.*pop*();

}

}

bool operator<(const Checkout& other) const { return qlength() < other.qlength(); }

bool operator>(const Checkout& other) const { return qlength() > other.qlength(); }

};

#endif

可以看到，queue 容器是 Checkout 唯一的成员变量，用来保存等待结账的 Customer 对象。成员函数 add() 可以向队列中添加新顾客。只能处理队列中的第一个元素。 每过一分钟，调用一次 Checkout 对象的成员函数 time\_increment(}，它会调用第一个 Customer 对象的成员函数 time\_decrement() 来减少剩余的等待时间，然后再调用成员函数 done()。如果 done() 返回 true，表明顾客结账完成，因此把他从队列中移除。Checkout 对象的比较运算符可以比较队列的长度。  
 为了模拟超市结账，我们需要有随机数生成的功能。因此打算使用 <random> 头文件中的一个非常简单的工具，但不打算深入解释它。我们会在教程后面的章节深入探讨 random 头文件中的内容。程序使用了一个 uniform\_int\_distribution() 类型的实例。顾名思义，它定义的整数值在最大值和最小值之间均匀分布。在均匀分布中，所有这个范围内的值都可能相等。可以在 10 和 100 之间定义如下分布：

**std::uniform\_int\_distribution<> d {10, 100};**

这里只定义了分布对象 d，它指定了整数值分布的范围。为了获取这个范围内的随机数，我们需要使用一个随机数生成器，然后把它作为参数传给 d 的调用运算符，从而返回一个随机整数。 random 头文件中定义了几种随机数生成器。这里我们使用最简单的一个，可以按如下方式定义：

**std::random\_device random\_number\_engine;**

为了在 d 分布范围内生成随机数，我们可以这样写：

**auto value = d(random\_number\_engine);**

**value 的值在 d 分布范围内。**  
  
完整模拟器的源文件如下：

#include <iostream> // For standard streams

#include <iomanip> // For stream manipulators

#include <vector> // For vector container

#include <string> // For string class

#include <numeric> // For accumulate()

#include <algorithm> // For min\_element & max\_element

#include <random> // For random number generation

#include "Customer.h"

#include "Checkout.h"

using *std*::*string*;

using distribution = *std*::*uniform\_int\_distribution*<>;

// 以横向柱形图的方式输出每个服务时间出现的次数

void histogram(const *std*::*vector*<int>& v, int min)

{

*string* bar(60, '\*');

for (*size\_t* i{}; i < v.*size*(); ++i)

{

*std*::*cout* << *std*::*setw*(3) << i + min << " " //结账等待时间为 index + min

<< *std*::*setw*(4) << v[i] << " " //输出出现的次数

<< bar.*substr*(0, v[i])

<< (v[i] > static\_cast<int>(bar.*size*()) ? "..." : "")

<< *std*::*endl*;

}

}

int *main*()

{

*std*::*random\_device* random\_n;

//设置最大和最小的结账时间，以分钟为单位

int service\_t\_min{ 2 }, service\_t\_max{ 15 };

distribution service\_t\_d{ service\_t\_min, service\_t\_max };

//设置在超市开业时顾客的人数

int min\_customers{ 15 }, max\_customers{ 20 };

distribution n\_1st\_customers\_d{ min\_customers, max\_customers };

// 设置顾客到达的最大和最小的时间间隔

int min\_arr\_interval{ 1 }, max\_arr\_interval{ 5 };

distribution arrival\_interval\_d{ min\_arr\_interval, max\_arr\_interval };

*size\_t* n\_checkouts{};

*std*::*cout* << "输入超市中结账柜台的数量：";

*std*::*cin* >> n\_checkouts;

if (!n\_checkouts)

{

*std*::*cout* << "结账柜台的数量必须大于 0，这里将默认设置为 1" << *std*::*endl*;

n\_checkouts = 1;

}

*std*::*vector*<Checkout> checkouts{ n\_checkouts };

*std*::*vector*<int> service\_times(service\_t\_max - service\_t\_min + 1);

//等待超市营业的顾客人数

int count{ n\_1st\_customers\_d(random\_n) };

*std*::*cout* << "等待超市营业的顾客人数：" << count << *std*::*endl*;

int added{};

int service\_t{};

while (added++ < count)

{

service\_t = service\_t\_d(random\_n);

*std*::*min\_element*(*std*::*begin*(checkouts), *std*::*end*(checkouts))->*add*(Customer(service\_t));

++service\_times[service\_t - service\_t\_min];

}

*size\_t* time{};

const *size\_t* total\_time{ 600 }; // 设置超市持续营业的时间

*size\_t* longest\_q{}; // 等待结账最长队列的长度

// 新顾客到达的时间

int new\_cust\_interval{ arrival\_interval\_d(random\_n) };

//模拟超市运转的过程

while (time < total\_time)

{

++time; //时间增长

// 新顾客到达

if (--new\_cust\_interval == 0)

{

service\_t = service\_t\_d(random\_n); // 设置顾客结账所需要的时间

*std*::*min\_element*(*std*::*begin*(checkouts), *std*::*end*(checkouts))->*add*(Customer(service\_t));

++service\_times[service\_t - service\_t\_min]; // 记录结账需要等待的时间

//记录最长队列的长度

for (auto& checkout : checkouts)

longest\_q = *std*::*max*(longest\_q, checkout.qlength());

new\_cust\_interval = arrival\_interval\_d(random\_n);

}

// 更新每个队列中第一个顾客的结账时间

for (auto& checkout : checkouts)

checkout.time\_increment();

}

*std*::*cout* << "最大的队列长度为：" << longest\_q << *std*::*endl*;

*std*::*cout* << "\n各个结账时间出现的次数：:\n";

histogram(service\_times, service\_t\_min);

*std*::*cout* << "\n总的顾客数："

<< *std*::*accumulate*(*std*::*begin*(service\_times), *std*::*end*(service\_times), 0)

<< *std*::*endl*;

return 0;

}

直接使用 using 指令可以减少代码输入，简化代码。顾客结账信息记录在 vector 中。结账时间减去 service\_times 的最小值可以用来索引需要自增的 vector 元素，这导致 vector 的第一个元素会记录下最少结账时间出现的次数。histogram() 函数会以水平条形图的形式生成每个服务时间出现次数的柱状图。  
 程序中 checkouts 的值为 600，意味着将模拟开业时间设置为 600 分钟，也可以用参数输入这个时间。main() 函数生成了顾客结账时间，超市开门时等在门外的顾客数，以及顾客到达时间间隔的分布对象。我们可以轻松地将这个程序扩展为每次到达的顾客数是一个处于一定范围内的随机数。  
 通过调用 min\_element() 算法可以找到最短的 Checkout 对象队列，因此顾客总是可以被分配到最短的结账队列。在这次模拟开始前，当超市开门营业时，在门外等待的顾客的初始序列被添加到 Checkout 对象中，然后结账时间记录被更新。  
 模拟在 while 循环中进行，在每次循环中，time 都会增加 1 分钟。在下一个顾客到达期间，new\_cust\_interval 会在每次循环中减小，直到等于 0。用新的随机结账时间生成新的顾客，然后把它加到最短的 Checkout 对象队列中。这个时候也会更新变量 longest\_q，因为在添加新顾客后，可能出现新的最长队列。然后调用每个 Checkout 对象的 time\_increment() 函数来处理队列中的第一个顾客。  
**下面是一些示例输出：**

**输入超级中结账柜台的数量：2  
等待超市营业的顾客人数：20  
最大的队列长度为：43  
  
各个结账时间出现的次数：  
  2   13 \*\*\*\*\*\*\*\*\*\*\*\*\*  
  3   20 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  4   11 \*\*\*\*\*\*\*\*\*\*\*  
  5   16 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  6   12 \*\*\*\*\*\*\*\*\*\*\*\*  
  7   18 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  8   17 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  9   18 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
10   10 \*\*\*\*\*\*\*\*\*\*  
11   22 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
12   19 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
13   13 \*\*\*\*\*\*\*\*\*\*\*\*\*  
14   16 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
15   18 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
总的顾客数：223**

**这里有 2 个结账柜台，最长队列的长度达到 43，已经长到会让顾客放弃付款。**  
以上代码还可以做更多改进，让模拟更加真实，例如，均匀分配并不符合实际，顾客通常成群结队到来。可以增加一些其他的因素，比如收银员休息时间、某个收银员生病工作状态不佳，这些都会导致顾客不选择这个柜台结账。

**第六C++ STL priority\_queue容器适配器详解**

priority\_queue 容器适配器模拟的也是队列这种存储结构，即使用此容器适配器存储元素只能“从一端进（称为队尾），从另一端出（称为队头）”，且每次只能访问 priority\_queue 中位于队头的元素。  
 但是，priority\_queue 容器适配器中元素的存和取，遵循的并不是 “First in,First out”（先入先出）原则，而是“First in，Largest out”原则。直白的翻译，指的就是先进队列的元素并不一定先出队列，而是优先级最大的元素最先出队列。

注意，“First in，Largest out”原则是笔者为了总结 priority\_queue 存取元素的特性自创的一种称谓，仅为了方便读者理解。

那么，priority\_queue 容器适配器中存储的元素，优先级是如何评定的呢？很简单，每个 priority\_queue 容器适配器在创建时，都制定了一种排序规则。根据此规则，该容器适配器中存储的元素就有了优先级高低之分。  
 举个例子，假设当前有一个 priority\_queue 容器适配器，其制定的排序规则是按照元素值从大到小进行排序。根据此规则，自然是 priority\_queue 中值最大的元素的优先级最高。  
 priority\_queue 容器适配器为了保证每次从队头移除的都是当前优先级最高的元素，每当有新元素进入，它都会根据既定的排序规则找到优先级最高的元素，并将其移动到队列的队头；同样，当 priority\_queue 从队头移除出一个元素之后，它也会再找到当前优先级最高的元素，并将其移动到队头。  
 基于 priority\_queue 的这种特性，因此该容器适配器有被称为优先级队列。

**priority\_queue 容器适配器“First in，Largest out”的特性，和它底层采用堆结构存储数据是分不开的。有关该容器适配器的底层实现，后续章节会进行深度剖析。**

**[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 中，priority\_queue 容器适配器的定义如下：**

**template <typename T,**

**typename Container=std::vector<T>,**

**typename Compare=std::less<T> >**

**class priority\_queue{**

**//......**

**}**

可以看到，priority\_queue 容器适配器模板类最多可以传入 3 个参数，它们各自的含义如下：

（1）typename T：指定存储元素的具体类型；

（2）typename Container：指定 priority\_queue 底层使用的基础容器，默认使用 vector 容器。

**作为 priority\_queue 容器适配器的底层容器，其必须包含 empty()、size()、front()、push\_back()、pop\_back() 这几个成员函数，[STL](http://c.biancheng.net/stl/" \t "http://c.biancheng.net/view/_blank) 序列式容器中只有 vector 和 deque 容器符合条件。**

（3）typename Compare：指定容器中评定元素优先级所遵循的排序规则，默认使用std::less<T>按照元素值从大到小进行排序，还可以使用std::greater<T>按照元素值从小到大排序，但更多情况下是使用自定义的排序规则。

**其中，std::less<T> 和 std::greater<T> 都是以函数对象的方式定义在 <function> 头文件中。关于如何自定义排序规则，后续章节会做详细介绍。**

**一创建priority\_queue的几种方式**

由于 priority\_queue 容器适配器模板位于<queue>头文件中，并定义在 std 命名空间里，因此在试图创建该类型容器之前，程序中需包含以下 2 行代码：

#include <queue>

u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g namespace std;  
创建 priority\_queue 容器适配器的方法，大致有以下几种。  
1) 创建一个空的 priority\_queue 容器适配器，第底层采用默认的 vector 容器，排序方式也采用默认的 std::less<T> 方法：

std::priority\_queue<int> values;  
2) 可以使用普通数组或其它容器中指定范围内的数据，对 priority\_queue 容器适配器进行初始化：

//使用普通数组

int values[]{4,1,3,2};

std::priority\_queue<int>copy\_values(values,values+4);//{4,2,3,1}

//使用序列式容器

std::array<int,4>values{ 4,1,3,2 };

std::priority\_queue<int>copy\_values(values.begin(),values.end());//{4,2,3,1}

注意，以上 2 种方式必须保证数组或容器中存储的元素类型和 priority\_queue 指定的存储类型相同。另外，用来初始化的数组或容器中的数据不需要有序，priority\_queue 会自动对它们进行排序。  
3) 还可以手动指定 priority\_queue 使用的底层容器以及排序规则，比如：

int values[]{ 4,1,2,3 };

std::priority\_queue<int, std::deque<int>, std::greater<int> >copy\_values(values, values+4);//{1,3,2,4}

事实上，std::less<T> 和 std::greater<T> 适用的场景是有限的，更多场景中我们会使用自定义的排序规则。

由于自定义排序规则的方式不只一种，因此这部分知识将在后续章节做详细介绍。

priority\_queue提供的成员函数

priority\_queue 容器适配器提供了表 2 所示的这些成员函数。

表 2 priority\_queue 提供的成员函数

|  |  |
| --- | --- |
| 成员函数 | 功能 |
| empty() | 如果 priority\_queue 为空的话，返回 true；反之，返回 false。 |
| size() | 返回 priority\_queue 中存储元素的个数。 |
| top() | 返回 priority\_queue 中第一个元素的引用形式。 |
| push(const T& obj) | 根据既定的排序规则，将元素 obj 的副本存储到 priority\_queue 中适当的位置。 |
| push(T&& obj) | 根据既定的排序规则，将元素 obj 移动存储到 priority\_queue 中适当的位置。 |
| emplace(Args&&... args) | Args&&... args 表示构造一个存储类型的元素所需要的数据（对于类对象来说，可能需要多个数据构造出一个对象）。此函数的功能是根据既定的排序规则，在容器适配器适当的位置直接生成该新元素。 |
| pop() | 移除 priority\_queue 容器适配器中第一个元素。 |
| swap(priority\_queue<T>& other) | 将两个 priority\_queue 容器适配器中的元素进行互换，需要注意的是，进行互换的 2 个 priority\_queue 容器适配器中存储的元素类型以及底层采用的基础容器类型，都必须相同。 |

和 queue 一样，priority\_queue 也没有迭代器，因此访问元素的唯一方式是遍历容器，通过不断移除访问过的元素，去访问下一个元素。

下面的程序演示了表 2 中部分成员函数的具体用法：

#include <iostream>

#include <queue>

#include <array>

#include <functional>

using namespace *std*;

int *main*()

{

//创建一个空的priority\_queue容器适配器

*std*::*priority\_queue*<int>values;

//使用 push() 成员函数向适配器中添加元素

values.*push*(3);//{3}

values.*push*(1);//{3,1}

values.*push*(4);//{4,1,3}

values.*push*(2);//{4,2,3,1}

//遍历整个容器适配器

while (!values.*empty*())

{

//输出第一个元素并移除。

*std*::*cout* << values.*top*() << " ";

values.*pop*();//移除队头元素的同时，将剩余元素中优先级最大的移至队头

}

return 0;

}

**运行结果为：**

**4 3 2 1**

**表 2 中其它成员函数的用法也非常简单，这里不再给出具体示例，后续章节用法会做具体介绍。**

**第七 priority\_queue容器适配器实现自定义排序**

前面讲解 priority\_queue 容器适配器时，还遗留一个问题，即当 <function> 头文件提供的排序方式（std::less<T> 和 std::greater<T>）不再适用时，如何自定义一个满足需求的排序规则。  
 首先，无论 priority\_queue 中存储的是基础数据类型（int、double 等），还是 string 类对象或者自定义的类对象，都可以使用函数对象的方式自定义排序规则。例如：

#include<iostream>

#include<queue>

using namespace *std*;

//函数对象类

template <typename T>

class cmp

{

public:

//重载 () 运算符

bool operator()(T a, T b)

{

return a > b;

}

};

int *main*()

{

int a[] = { 4,2,3,5,6 };

*priority\_queue*<int, *vector*<int>, cmp<int> > pq(a, a + 5);

while (!pq.*empty*())

{

*cout* << pq.*top*() << " ";

pq.*pop*();

}

return 0;

}

**运行结果为：**

**2 3 4 5 6**

注意，C++ 中的 struct 和 class 非常类似，前者也可以包含成员变量和成员函数，因此上面程序中，函数对象类 cmp 也可以使用 struct 关键字创建：

**struct cmp**

**{**

**//重载 () 运算符**

**bool operator()(T a, T b)**

**{**

**return a > b;**

**}**

**};**

可以看到，通过在 cmp 类（结构体）重载的 () 运算符中自定义排序规则，并将其实例化后作为 priority\_queue 模板的第 3 个参数传入，即可实现为 priority\_queue 容器适配器自定义比较函数。  
 除此之外，当 priority\_queue 容器适配器中存储的数据类型为结构体或者类对象（包括 string 类对象）时，还可以通过重载其 > 或者 < 运算符，间接实现自定义排序规则的目的。

**注意，此方式仅适用于 priority\_queue 容器中存储的为类对象或者结构体变量，也就是说，当存储类型为类的指针对象或者结构体指针变量时，此方式将不再适用，而只能使用函数对象的方式。**

要想彻底理解这种方式的实现原理，首先要搞清楚 std::less<T> 和 std::greater<T> 各自的底层实现。实际上，<function> 头文件中的 std::less<T> 和 std::greater<T> ，各自底层实现采用的都是函数对象的方式。比如，std::less<T> 的底层实现代码为：

**template <typename T>**

**struct less {**

**//定义新的排序规则**

**bool operator()(const T &\_lhs, const T &\_rhs) const {**

**return \_lhs < \_rhs;**

**}**

**};**

**std::greater<T> 的底层实现代码为：**

**template <typename T>**

**struct greater {**

**bool operator()(const T &\_lhs, const T &\_rhs) const {**

**return \_lhs > \_rhs;**

**}**

**};**

可以看到，std::less<T> 和 std::greater<T> 底层实现的唯一不同在于，前者使用 < 号实现从大到小排序，后者使用 > 号实现从小到大排序。  
 那么，是否可以通过重载 < 或者 > 运算符修改 std::less<T> 和 std::greater<T> 的排序规则，从而间接实现自定义排序呢？答案是肯定的，举个例子：

#include<queue>

#include<iostream>

using namespace *std*;

class node {

public:

node(int x = 0, int y = 0) :x(x), y(y) {}

int x, y;

};

//新的排序规则为：先按照 x 值排序，如果 x 相等，则按 y 的值排序

bool operator < (const node& a, const node& b) {

if (a.x > b.x) return 1;

else if (a.x == b.x)

if (a.y >= b.y) return 1;

return 0;

}

int *main*() {

//创建一个 priority\_queue 容器适配器，其使用默认的 vector 基础容器以及 less 排序规则。

*priority\_queue*<node> pq;

pq.*push*(node(1, 2));

pq.*push*(node(2, 2));

pq.*push*(node(3, 4));

pq.*push*(node(3, 3));

pq.*push*(node(2, 3));

*cout* << "x y" << *endl*;

while (!pq.*empty*()) {

*cout* << pq.*top*().x << " " << pq.*top*().y << *endl*;

pq.*pop*();

}

return 0;

}

**输出结果为：**

**x y  
1 2  
2 2  
2 3  
3 3  
3 4**

可以看到，通过重载 < 运算符，使得 std::less<T> 变得适用了。

**读者还可以自行尝试，通过重载 > 运算符，赋予 std::greater<T> 和之前不同的排序方式。**

当然，也可以以友元函数或者成员函数的方式重载 > 或者 < 运算符。需要注意的是，以成员函数的方式重载 > 或者 < 运算符时，该成员函数必须声明为 const 类型，且参数也必须为 const 类型，至于参数的传值方式是采用按引用传递还是按值传递，都可以（建议采用按引用传递，效率更高）。  
例如，将上面程序改为以成员函数的方式重载 < 运算符：

class node {

public:

node(int x = 0, int y = 0) :x(x), y(y) {}

int x, y;

bool operator < (const node &b) const{

if ((\*this).x > b.x) return 1;

else if ((\*this).x == b.x)

if ((\*this).y >= b.y) return 1;

return 0;

}

};

同样，在以友元函数的方式重载 < 或者 > 运算符时，要求参数必须使用 const 修饰。例如，将上面程序改为以友元函数的方式重载 < 运算符。例如：

[纯文本复制](http://c.biancheng.net/view/vip_7728.html)

class node {

public:

node(int x = 0, int y = 0) :x(x), y(y) {}

int x, y;

friend bool operator < (const node &a, const node &b);

};

//新的排序规则为：先按照 x 值排序，如果 x 相等，则按 y 的值排序

bool operator < (const node &a, const node &b){

if (a.x > b.x) return 1;

else if (a.x == b.x)

if (a.y >= b.y) return 1;

return 0;

}

总的来说，以函数对象的方式自定义 priority\_queue 的排序规则，适用于任何情况；而以重载 > 或者 < 运算符间接实现 priority\_queue 自定义排序的方式，仅适用于 priority\_queue 中存储的是结构体变量或者类对象（包括 string 类对象）。

**第八深度剖析priority\_queue容器的底层实现**

priority\_queue 优先级队列之所以总能保证优先级最高的元素位于队头，最重要的原因是其底层采用堆数据结构存储结构。  
 有读者可能会问，priority\_queue 底层不是采用 vector 或 deque 容器存储数据吗，这里又说使用堆结构存储数据，它们之间不冲突吗？显然，它们之间是不冲突的。  
 首先，vector 和 deque 是用来存储元素的容器，而堆是一种数据结构，其本身无法存储数据，只能依附于某个存储介质，辅助其组织数据存储的先后次序。其次，priority\_queue 底层采用 vector 或者 deque 作为基础容器，这毋庸置疑。但由于 vector 或 deque 容器并没有提供实现 priority\_queue 容器适配器 “First in,Largest out” 特性的功能，因此 STL 选择使用堆来重新组织 vector 或 deque 容器中存储的数据，从而实现该特性。

**注意，虽然不使用堆结构，通过编写算法调整 vector 或者 deque 容器中存储元素的次序，也能使其具备 “First in,Largest out” 的特性，但执行效率通常没有使用堆结构高。**

**那么，堆到底是什么，它又是怎样组织数据的呢？**

**一priority\_queue底层的堆存储结构**

以下内容要求读者对数据结构中的树存储结构有一定的了解，如果没有，请先阅读《[树存储结构](http://c.biancheng.net/data_structure/tree/" \t "http://c.biancheng.net/view/_blank)》一章。

简单的理解堆，它在是完全二叉树的基础上，要求树中所有的父节点和子节点之间，都要满足既定的排序规则：

（1）如果排序规则为从大到小排序，则表示堆的完全二叉树中，每个父节点的值都要不小于子节点的值，这种堆通常称为大顶堆；

（2）如果排序规则为从小到大排序，则表示堆的完全二叉树中，每个父节点的值都要不大于子节点的值，这种堆通常称为小顶堆；  
图 1 展示了一个由 {10,20,15,30,40,25,35,50,45} 这些元素构成的大顶堆和小顶堆。其中经大顶堆组织后的数据先后次序变为 {50,45,40,20,25,35,30,10,15}，而经小顶堆组织后的数据次序为{10,20,15,25,50,30,40,35,45}。

可以确定的一点是，无论是通过大顶堆或者小顶堆，总可以筛选出最大或最小的那个元素（优先级最大），并将其移至序列的开头，此功能也正是 priority\_queue 容器适配器所需要的。  
 为了验证 priority\_queue 底层确实采用堆存储结构实现的，我们可以尝试用堆结合基础容器 vector 或 deque 实现 priority\_queue。值得庆幸的是，STL 已经为我们封装好了可以使用堆存储结构的方法，它们都位于 <algorithm> 头文件中。表 2 中列出了常用的几个和堆存储结构相关的方法。

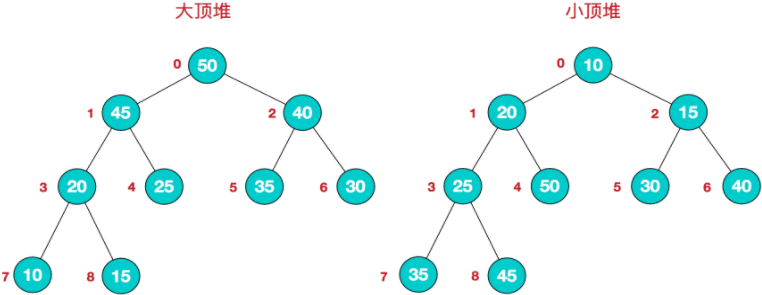
  
图 1 使用堆结构重新组织数据

表 2 STL对堆存储结构的支持

|  |  |
| --- | --- |
| 函数 | 功能 |
| make\_heap(first,last,comp) | 选择位于 [first,last) 区域内的数据，并根据 comp 排序规则建立堆，其中 fist 和 last 可以是指针或者迭代器，默认是建立大顶堆。 |
| push\_heap(first,last,comp) | 当向数组或容器中添加数据之后，此数据可能会破坏堆结构，该函数的功能是重建堆。 |
| pop\_heap(first,last,comp) | 将位于序列头部的元素（优先级最高）移动序列尾部，并使[first,last-1] 区域内的元素满足堆存储结构。 |
| sort\_heap(first,last,comp) | 对 [first,last) 区域内的元素进行堆排序，将其变成一个有序序列。 |
| is\_heap\_until(first,last,comp) | 发现[first,last)区域内的最大堆。 |
| is\_heap(first,last,comp) | 检查 [first,last) 区域内的元素，是否为堆结构。 |

以上方法的实现，基于堆排序算法的思想，有关该算法的具体实现原理，可阅读《[堆排序](http://c.biancheng.net/view/3448.html" \t "http://c.biancheng.net/view/_blank)》一节做详细了解。

下面例子中，使用了表 2 中的部分函数，并结合 vector 容器提供的成员函数，模拟了 priority\_queue 容器适配器部分成员函数的底层实现：

#include <iostream>

#include <vector>

#include<algorithm>

using namespace *std*;

void display(*vector*<int>& val) {

for (auto v : val) {

*cout* << v << " ";

}

*cout* << *endl*;

}

int *main*()

{

*vector*<int>values{ 2,1,3,4 };

//建立堆

*make\_heap*(values.*begin*(), values.*end*());//{4,2,3,1}

display(values);

//添加元素

*cout* << "添加元素：\n";

values.*push\_back*(5);

display(values);

*push\_heap*(values.*begin*(), values.*end*());//{5,4,3,1,2}

display(values);

//移除元素

*cout* << "移除元素：\n";

*pop\_heap*(values.*begin*(), values.*end*());//{4,2,3,1,5}

display(values);

values.*pop\_back*();

display(values);

return 0;

}

**运行结果为：**

**4 2 3 1  
添加元素：  
4 2 3 1 5  
5 4 3 1 2  
移除元素：  
4 2 3 1 5  
4 2 3 1**

上面程序可以用 priority\_queue 容器适配器等效替代：

#include<iostream>

#include<queue>

#include<vector>

using namespace *std*;

int *main*()

{

//创建优先级队列

*std*::*vector*<int>values{ 2,1,3,4 };

*std*::*priority\_queue*<int>copy\_values(values.*begin*(), values.*end*());

//添加元素

copy\_values.*push*(5);

//移除元素

copy\_values.*pop*();

return 0;

}

如果调试此程序，查看各个阶段 priority\_queue 中存储的元素，可以发现，它和上面程序的输出结果是一致。也就是说，此程序在创建 priority\_queue 之后，其存储的元素依次为 {4,2,3,1}，同样当添加元素 5 之后，其存储的元素依次为 {5,4,3,1,2}，移除一个元素之后存储的元素依次为 {4,2,3,1}。

**第六章 C++ STL迭代器适配器完全攻略**

本章将介绍 5 种迭代器适配器，分别是反向迭代器适配器、插入型迭代器适配器、流迭代器适配器、流缓冲区迭代器适配器、移动迭代器适配器。

初学者完全可以将迭代器适配器视为普通迭代器。之所以称为迭代器适配器，是因为这些迭代器是在输入迭代器、输出迭代器、正向迭代器、双向迭代器或者随机访问迭代器这些基础迭代器的基础上实现的。也就是说，使用迭代器适配器的过程中，其本质就是在操作某种基础迭代器。

不同的迭代器适配器，其功能大不相同，这些知识都会在本章中做详细讲解。

**第一C++ STL迭代器适配器是什么？**

通过学习 C++ STL 标准库中的容器我们知道，无论是序列式容器还是关联式容器（包括哈希容器），要想遍历容器中存储的数据，就只能用使用该容器模板类中提供的迭代器。  
 《[C++ STL迭代器](http://c.biancheng.net/view/6675.html" \t "http://c.biancheng.net/view/_blank)》一节提到，C++ STL 标准库中迭代器大致分为 5 种类型，分别是输入迭代器、输出迭代器、前向迭代器、双向迭代器以及随机访问迭代器。值得一提的是，这 5 种迭代器是 STL 标准库提供的最基础的迭代器，很多场景中遍历容器的需求，它们并不适合。  
 举个例子，假设有一个 list 容器，现在需要逆序输出该容器中存储的所有元素。要知道，list 容器模板类提供的是双向迭代器，如果使用该类型迭代器实现逆序操作，实现代码如下：

#include <iostream>

#include <list>

using namespace *std*;

int *main*()

{

*std*::*list*<int> values{ 1,2,3,4,5 };

//找到遍历的开头位置和结尾位置

*std*::*list*<int>::*iterator* begin = --values.*end*();

*std*::*list*<int>::*iterator* end = --values.*begin*();

//开始遍历

while (begin != end)

{

*cout* << \*begin << " ";

--begin;

}

return 0;

}

**程序执行结果为：**

**5 4 3 2 1**

相比上面这种实现思路，C++ STL 标准库中还提供有更简单的方法，就是使用迭代器适配器。

**关于适配器，在讲解容器适配器时就已经做过详细的讲解，这里不再做过多赘述，读者可阅读《[C++ STL容器适配器](http://c.biancheng.net/view/6967.html" \t "http://c.biancheng.net/view/_blank)》一节做详细了解。**

所谓迭代器适配器，其本质也是一个模板类，比较特殊的是，该模板类是借助以上 5 种基础迭代器实现的。换句话说，迭代器适配器模板类的内部实现，是通过对以上 5 种基础迭代器拥有的成员方法进行整合、修改，甚至为了实现某些功能还会添加一些新的成员方法。由此，将基础迭代器“改头换面”，就变成了本节要讲的迭代器适配器。

**本质上讲，迭代器适配器仍属于迭代器，可以理解为是基础迭代器的“翻新版”或者“升级版”。同时，“xxx 迭代器适配器”通常直接称为“xxx 迭代器”。**

**一 C++ STL迭代器适配器种类**

C++ 11 标准中，迭代器适配器供有 4 类，它们各自的名称和功能如表 1 所示。

表 1 C++ STL迭代器适配器种类

|  |  |
| --- | --- |
| 名称 | 功能 |
| 反向迭代器（reverse\_iterator） | 又称“逆向迭代器”，其内部重新定义了递增运算符（++）和递减运算符（--），专门用来实现对容器的逆序遍历。 |
| 安插型迭代器（inserter或者insert\_iterator） | 通常用于在容器的任何位置添加新的元素，需要注意的是，此类迭代器不能被运用到元素个数固定的容器（比如 array）上。 |
| 流迭代器（istream\_iterator / ostream\_iterator） 流缓冲区迭代器（istreambuf\_iterator / ostreambuf\_iterator） | 输入流迭代器用于从文件或者键盘读取数据；相反，输出流迭代器用于将数据输出到文件或者屏幕上。 输入流缓冲区迭代器用于从输入缓冲区中逐个读取数据；输出流缓冲区迭代器用于将数据逐个写入输出流缓冲区。 |
| 移动迭代器（move\_iterator） | 此类型迭代器是 C++ 11 标准中新添加的，可以将某个范围的类对象移动到目标范围，而不需要通过拷贝去移动。 |

以上 4 种迭代器的用法，后续章节会做详细介绍。  
实际上，前面在学习各种容器的迭代器时，我们经常会使用到反向迭代器。下面样例，演示了用反向迭代器适配器遍历 list 容器的实现过程：

#include <iostream>

#include <list>

using namespace *std*;

int *main*()

{

*std*::*list*<int> values{ 1,2,3,4,5 };

//找到遍历的起点和终点，这里无需纠结定义反向迭代器的语法，后续会详细讲解

*std*::*reverse\_iterator*<*std*::*list*<int>::*iterator*> begin = values.*rbegin*();

*std*::*reverse\_iterator*<*std*::*list*<int>::*iterator*> end = values.*rend*();

while (begin != end) {

*cout* << \*begin << " ";

//注意，这里是 ++，因为反向迭代器内部互换了 ++ 和 -- 的含义

++begin;

}

return 0;

}

**程序执行结果为：**

**5 4 3 2 1**

可以看到，程序中通过调用 list 容器模板类提供的 rbegin() 和 rend()，就可以获得逆序遍历容器所需要的反向迭代器，从而轻松实现逆序输出容器中存储的所有数据。

**第二 C++ STL 反向迭代器适配器（reverse\_iterator）详解**

我们知道，C++ 11 的 STL 标准库提供有 4 种迭代器适配器，本节开始将一一介绍它们的功能和用法，这里先讲解反向迭代器适配器。  
 反向迭代器适配器（reverse\_iterator），可简称为反向迭代器或逆向迭代器，常用来对容器进行反向遍历，即从容器中存储的最后一个元素开始，一直遍历到第一个元素。  
 值得一提的是，反向迭代器底层可以选用双向迭代器或者随机访问迭代器作为其基础迭代器。不仅如此，通过对 ++（递增）和 --（递减）运算符进行重载，使得：

（1）当反向迭代器执行 ++ 运算时，底层的基础迭代器实则在执行 -- 操作，意味着反向迭代器在反向遍历容器；

（2）当反向迭代器执行 -- 运算时，底层的基础迭代器实则在执行 ++ 操作，意味着反向迭代器在正向遍历容器。  
 另外，实现反向迭代器的模板类定义在 <iterator> 头文件，并位于 std 命名空间中。因此，在使用反向迭代器时，需包含如下语句：

#include <iterator>

using namespace std;

注意，第二行代码不是必需的，但如果不用，则程序中只要创建该迭代器时，必须手动注明 std 命名空间（强烈建议初学者使用）。

**反向迭代器的模板类定义如下：**

**template <class Iterator>**

**class reverse\_iterator;**

**注意，Iterator 模板参数指的是模板类中所用的基础迭代器的类型，只能选择双向迭代器或者随机访问迭代器。**

这意味着，如果想使用反向迭代器实现逆序遍历容器，则该容器的迭代器类型必须是双向迭代器或者随机访问迭代器。

**一 C++ STL反向迭代器的创建**

reverse\_iterator 模板类中共提供了 3 种创建反向迭代器的方法，这里以 vector<int> 容器的随机访问迭代器作为基础迭代器为例。  
1) 调用该类的默认构造方法，即可创建了一个不指向任何对象的反向迭代器，例如：

std::reverse\_iterator<std::vector<int>::iterator> my\_reiter;

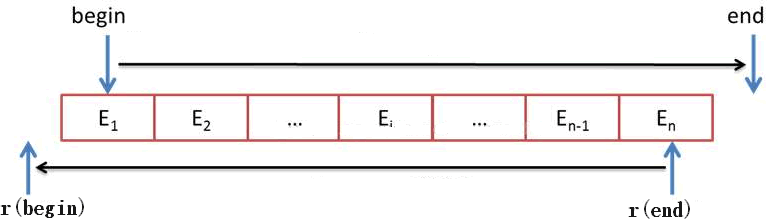
由此，我们就创建好了一个没有指向任何对象的 my\_reiter 反向迭代器。  
2) 当然，在创建反向迭代器的时候，我们可以直接将一个基础迭代器作为参数传递给新建的反向迭代器。例如：

**//创建并初始化一个 myvector 容器**

**std::vector<int> myvector{1,2,3,4,5};**

**//创建并初始化 my\_reiter 迭代器**

**std::reverse\_iterator<std::vector<int>::iterator> my\_reiter(myvector.end());**

我们知道，反向迭代器是通过操纵内部的基础迭代器实现逆向遍历的，但是反向迭代器的指向和底层基础迭代器的指向并不相同。以上面创建的 my\_reiter 为例，其内部的基础迭代器指向的是 myvector 容器中元素 5 之后的位置，但是 my\_reiter 指向的却是元素 5。  
 也就是说，反向迭代器的指向和其底层基础迭代器的指向具有这样的关系，即反向迭代器的指向总是距离基础迭代器偏左 1 个位置；反之，基础迭代器的指向总是距离反向迭代器偏右 1 个位置处。它们的关系如图 1 所示。  
  
图 1 反向迭代器和基础迭代器的关系

其中，begin 和 end 表示基础迭代器，r(begin) 和 r(end) 分别表示有 begin 和 end 获得的反向迭代器。  
3) 除了第 2 种初始化方式之外，reverse\_iterator 模板类还提供了一个复制（拷贝）构造函数，可以实现直接将一个反向迭代器复制给新建的反向迭代器。比如：

**//创建并初始化一个 vector 容器**

**std::vector<int> myvector{1,2,3,4,5};**

**//调用复制构造函数初始化反向迭代器的 2 种方式**

**std::reverse\_iterator<std::vector<int>::iterator> my\_reiter(myvector.rbegin());**

**//std::reverse\_iterator<std::vector<int>::iterator> my\_reiter = myvector.rbegin();**

由此，my\_reiter 反向迭代器指向的就是 myvector 容器中最后一个元素（也就是 5）之后的位置。

**二C++ STL reverse\_iterator模板类中的成员**

前面在学习每一种容器时，都提供有大量的成员函数。但迭代器模板类不同，其内部更多的是对运算符的重载。  
reverse\_iterator模板类中，重载了如表 1 所示的这些运算符。

表 1 reverse\_iterator重载的运算符

|  |  |
| --- | --- |
| 重载运算符 | 功能 |
| operator\* | 以引用的形式返回当前迭代器指向的元素。 |
| operator+ | 返回一个反向迭代器，其指向距离当前指向的元素之后 n 个位置的元素。此操作要求基础迭代器为随机访问迭代器。 |
| operator++ | 重载前置 ++ 和后置 ++ 运算符。 |
| operator+= | 当前反向迭代器前进 n 个位置，此操作要求基础迭代器为随机访问迭代器。 |
| operator- | 返回一个反向迭代器，其指向距离当前指向的元素之前 n 个位置的元素。此操作要求基础迭代器为随机访问迭代器。 |
| operator-- | 重载前置 -- 和后置 -- 运算符。 |
| operator-= | 当前反向迭代器后退 n 个位置，此操作要求基础迭代器为随机访问迭代器。 |
| operator-> | 返回一个指针，其指向当前迭代器指向的元素。 |
| operator[n] | 访问和当前反向迭代器相距 n 个位置处的元素。 |

下面程序演示了表 1 中部分运算符的用法：

#include <iostream>

#include <iterator>

#include <vector>

using namespace *std*;

int *main*() {

//创建并初始化一个 vector 容器

*std*::*vector*<int> myvector{ 1,2,3,4,5,6,7,8 };

//创建并初始化一个反向迭代器

*std*::*reverse\_iterator*<*std*::*vector*<int>::*iterator*> my\_reiter(myvector.*rbegin*());//指向 8

*cout* << \*my\_reiter << *endl*;// 8

*cout* << \*(my\_reiter + 3) << *endl*;// 5

*cout* << \*(++my\_reiter) << *endl*;// 7

*cout* << my\_reiter[4] << *endl*;// 3

return 0;

}

**程序执行结果为：**

**8  
5  
7  
3**

可以看到，首先 my\_reiter 方向迭代器指向 myvector 容器中元素 8，后续我们调用了如下几个运算符：

（1）通过重载的 \* 运算符，输出其指向的元素 8；

（2）通过重载的 + 运算符，输出了距离当前指向位置为 3 的元素 5；

（3）通过重载的前置 ++ 运算符，将反向迭代器前移了 1 位，即指向了元素 7，并将其输出；

（4）通过重载的 [ ] 运算符，输出了距离当前位置为 4 的元素 3。  
除此之外，reverse\_iterator 模板类还提供了 base() 成员方法，该方法可以返回当前反向迭代器底层所使用的基础迭代器。举个例子：

#include <iostream>

#include <iterator>

#include <vector>

using namespace *std*;

int *main*() {

//创建并初始化一个 vector 容器

*std*::*vector*<int> myvector{ 1,2,3,4,5,6,7,8 };

//创建并初始化反向迭代器 begin，其指向元素 1 之前的位置

*std*::*reverse\_iterator*<*std*::*vector*<int>::*iterator*> begin(myvector.*begin*());

//创建并初始化反向迭代器 begin，其指向元素 8

*std*::*reverse\_iterator*<*std*::*vector*<int>::*iterator*> end(myvector.*end*());

//begin底层基础迭代器指向元素 1，end底层基础迭代器指向元素 8 之后的位置

for (auto iter = begin.*base*(); iter != end.*base*(); ++iter) {

*std*::*cout* << \*iter << ' ';

}

return 0;

}

**程序执行结果为：**

**1 2 3 4 5 6 7 8**

**注意程序中第 13 行代码，begin 和 end 都是反向迭代器，通过调用 base() 成员方法，可以获取各自底层对应的基础迭代器。由于基础迭代器类型为随机访问迭代器，因此其 ++ 操作实现的就是普通意义上的正向遍历。**

**第三 C++ STL插入迭代器适配器（insert\_iterator）**

《[C++ STL反向迭代器](http://c.biancheng.net/view/7274.html" \t "http://c.biancheng.net/view/_blank)》一节讲解了反向迭代器适配器的功能和用法，本节继续讲解 C++ STL 标准库提供的另一类型迭代器适配器，即插入迭代器适配器。  
 插入迭代器适配器（insert\_iterator），简称插入迭代器或者插入器，其功能就是向指定容器中插入元素。值得一提的是，根据插入位置的不同，C++ STL 标准库提供了 3 种插入迭代器，如表 1 所示。

表 1 C++ STL插入迭代器适配器种类

|  |  |
| --- | --- |
| 迭代器适配器 | 功能 |
| back\_insert\_iterator | 在指定容器的尾部插入新元素，但前提必须是提供有 push\_back() 成员方法的容器（包括 vector、deque 和 list）。 |
| front\_insert\_iterator | 在指定容器的头部插入新元素，但前提必须是提供有 push\_front() 成员方法的容器（包括 list、deque 和 forward\_list）。 |
| insert\_iterator | 在容器的指定位置之前插入新元素，前提是该容器必须提供有 insert() 成员方法。 |

接下来，将逐个对表 1 中这 3 种插入迭代器的用法做详细的讲解。

**一 C++ STL back\_insert\_iterator迭代器**

back\_insert\_iterator 迭代器可用于在指定容器的末尾处添加新元素。  
需要注意的是，由于此类型迭代器的底层实现需要调用指定容器的 push\_back() 成员方法，这就意味着，此类型迭代器并不适用于 STL 标准库中所有的容器，它只适用于包含 push\_back() 成员方法的容器。

**C++ STL 标准库中，提供有 push\_back() 成员方法的容器包括 vector、deque 和 list。**

另外，back\_insert\_iterator 迭代器定义在 <iterator> 头文件，并位于 std 命名空间中，因此在使用该类型迭代器之前，程序应包含如下语句：

#include <iterator>

using namespace std;

注意，第二行代码不是必需的，但如果不用，则程序中只要创建该类型的迭代器，就必须手动注明 std 命名空间（强烈建议初学者使用）。

和反向迭代器不同，back\_insert\_iterator 插入迭代器的定义方式仅有一种，其语法格式如下：

std::back\_insert\_iterator<Container> back\_it (container);

其中，Container 用于指定插入的目标容器的类型；container 用于指定具体的目标容器。  
举个例子：

//创建一个 vector 容器

std::vector<int> foo;

//创建一个可向 foo 容器尾部添加新元素的迭代器

std::back\_insert\_iterator< std::vector<int> > back\_it(foo);

**当然，如果程序中已经引入了 std 命名空间，则以上程序中所有的 std:: 都可以省略。**

在此基础上，back\_insert\_iterator 迭代器模板类中还对赋值运算符（=）进行了重载，借助此运算符，我们可以直接将新元素插入到目标容器的尾部。例如：

#include <iostream>

#include <iterator>

#include <vector>

using namespace *std*;

int *main*() {

//创建一个 vector 容器

*std*::*vector*<int> foo;

//创建一个可向 foo 容器尾部添加新元素的迭代器

*std*::*back\_insert\_iterator*< *std*::*vector*<int> > back\_it(foo);

//将 5 插入到 foo 的末尾

back\_it = 5;

//将 4 插入到当前 foo 的末尾

back\_it = 4;

//将 3 插入到当前 foo 的末尾

back\_it = 3;

//将 6 插入到当前 foo 的末尾

back\_it = 6;

//输出 foo 容器中的元素

for (*std*::*vector*<int>::*iterator* it = foo.*begin*(); it != foo.*end*(); ++it)

*std*::*cout* << \*it << ' ';

return 0;

}

**程序执行结果为：**

**5 4 3 6**

通过借助赋值运算符，我们依次将 5、4、3、6 插入到 foo 容器中的末尾。这里需要明确的是，每次插入新元素时，该元素都会插入到当前 foo 容器的末尾。换句话说，程序中 11-17 行的每个赋值语句，都可以分解为以下这 2 行代码：

**//pos表示指向容器尾部的迭代器，value 表示要插入的元素**

**pos = foo.insert(pos,value);**

**++pos;**

可以看到，每次将新元素插入到容器的末尾后，原本指向容器末尾的迭代器会后移一位，指向容器新的末尾。  
 除此之外，C++ STL 标准库为了方便用户创建 back\_insert\_iterator 类型的插入迭代器，提供了 back\_inserter() 函数，其语法格式如下：

**template <class Container>  
    back\_insert\_iterator<Container> back\_inserter (Container& x);**

其中，Container 表示目标容器的类型。  
显然在使用该函数时，只需要为其传递一个具体的容器（vector、deque 或者 list）做参数，此函数即可返回一个 back\_insert\_iterator 类型的插入迭代器。因此，上面程序中的第 9 行代码，可替换成如下语句：

**std::back\_insert\_iterator< std::vector<int> > back\_it = back\_inserter(foo);**

通过接收 back\_inserter() 的返回值，我们也可以得到完全一样的 back\_it 插入迭代器。

有关此类型迭代器的底层实现，[C++ STL back\_insert\_iterator官方手册](http://www32.cplusplus.com/reference/iterator/back_insert_iterator/" \t "http://c.biancheng.net/view/_blank)给出了具体的实现代码，有兴趣的读者可自行前往查看。

**二 C++ STL front\_insert\_iterator迭代器**

和 back\_insert\_iterator 正好相反，front\_insert\_iterator 迭代器的功能是向目标容器的头部插入新元素。  
 并且，由于此类型迭代器的底层实现需要借助目标容器的 push\_front() 成员方法，这意味着，只有包含 push\_front() 成员方法的容器才能使用该类型迭代器。

**C++ STL 标准库中，提供有 push\_front() 成员方法的容器，仅有 deque、list 和 forward\_list。**

另外，front\_insert\_iterator 迭代器定义在 <iterator> 头文件，并位于 std 命名空间中，因此在使用该类型迭代器之前，程序应包含如下语句：

#include <iterator>

using namespace std;

值得一提的是，定义 front\_insert\_iterator 迭代器的方式和 back\_insert\_iterator 完全相同，并且 C++ STL 标准库也提供了 front\_inserter() 函数来快速创建该类型迭代器。  
举个例子：

#include <iostream>

#include <iterator>

#include <forward\_list>

using namespace *std*;

int *main*() {

//创建一个 forward\_list 容器

*std*::*forward\_list*<int> foo;

//创建一个前插入迭代器

//std::front\_insert\_iterator< std::forward\_list<int> > front\_it(foo);

*std*::*front\_insert\_iterator*< *std*::*forward\_list*<int> > front\_it = *front\_inserter*(foo);

//向 foo 容器的头部插入元素

front\_it = 5;

front\_it = 4;

front\_it = 3;

front\_it = 6;

for (*std*::*forward\_list*<int>::*iterator* it = foo.*begin*(); it != foo.*end*(); ++it)

*std*::*cout* << \*it << ' ';

return 0;

}

**程序执行结果为：**

**6 3 4 5**

同样，[C++ STL back\_insert\_iterator官方手册](http://www32.cplusplus.com/reference/iterator/back_insert_iterator/" \t "http://c.biancheng.net/view/_blank)也给出了此类型迭代器底层实现的参考代码，有兴趣的读者可自行前往查看。

**三 C++ STL insert\_iterator迭代器**

当需要向容器的任意位置插入元素时，就可以使用 insert\_iterator 类型的迭代器。  
需要说明的是，该类型迭代器的底层实现，需要调用目标容器的 insert() 成员方法。但幸运的是，STL 标准库中所有容器都提供有 insert() 成员方法，因此 insert\_iterator 是唯一可用于关联式容器的插入迭代器。  
 和前 2 种插入迭代器一样，insert\_iterator 迭代器也定义在 <iterator> 头文件，并位于 std 命名空间中，因此在使用该类型迭代器之前，程序应包含如下语句：

#include <iterator>

using namespace std;

不同之处在于，定义 insert\_iterator 类型迭代器的语法格式如下：

std::insert\_iterator<Container> insert\_it (container,it);

其中，Container 表示目标迭代器的类型，参数 container 表示目标迭代器，而 it 是一个基础迭代器，表示新元素的插入位置。  
 和前 2 种插入迭代器相比，insert\_iterator 迭代器除了定义时需要多传入一个参数，它们的用法完全相同。除此之外，C++ STL 标准库中还提供有 inserter() 函数，可以快速创建 insert\_iterator 类型迭代器。  
  
举个例子：

#include <iostream>

#include <iterator>

#include <list>

using namespace *std*;

int *main*() {

//初始化为 {5,5}

*std*::*list*<int> foo(2, 5);

//定义一个基础迭代器，用于指定要插入新元素的位置

*std*::*list*<int>::*iterator* it = ++foo.*begin*();

//创建一个 insert\_iterator 迭代器

//std::insert\_iterator< std::list<int> > insert\_it(foo, it);

*std*::*insert\_iterator*< *std*::*list*<int> > insert\_it = *inserter*(foo, it);

//向 foo 容器中插入元素

insert\_it = 1;

insert\_it = 2;

insert\_it = 3;

insert\_it = 4;

//输出 foo 容器存储的元素

for (*std*::*list*<int>::*iterator* it = foo.*begin*(); it != foo.*end*(); ++it)

*std*::*cout* << \*it << ' ';

return 0;

}

**程序执行结果为：**

**5 1 2 3 4 5**

需要注意的是，如果 insert\_iterator 迭代器的目标容器为关联式容器，由于该类型容器内部会自行对存储的元素进行排序，因此我们指定的插入位置只起到一个提示的作用，即帮助关联式容器从指定位置开始，搜索正确的插入位置。但是，如果提示位置不正确，会使的插入操作的效率更加糟糕。

**[C++ STL insert\_iterator官方手册](http://www32.cplusplus.com/reference/iterator/insert_iterator/" \t "http://c.biancheng.net/view/_blank)中给出了此类型迭代器底层实现的参考代码，有兴趣的读者可自行前往查看。**

**总结**

本节讲解了 3 种插入迭代器，虽然它们都可以借助重载的赋值运算符，实现向目标容器的指定位置插入新元素，但在实际应用中，它们通常和 copy() 函数连用，即作为 copy() 函数的第 3 个参数。

有关 copy() 函数的具体用法，由于不是本节重点，这里不再赘述，后续章节会做详细讲解。

**第四 C++ STL流迭代器(istream\_iterator和ostream\_iterator)**

流迭代器也是一种迭代器适配器，不过和之前讲的迭代器适配器有所差别，它的操作对象不再是某个容器，而是流对象。即通过流迭代器，我们可以读取指定流对象中的数据，也可以将数据写入到流对象中。

**通常情况下，我们经常使用的 cin、cout 就属于流对象，其中 cin 可以获取键盘输入的数据，cout 可以将指定数据输出到屏幕上。除此之外，更常见的还有文件 I/O 流等。关于什么流，更详细的介绍可阅读《[C++流类和流对象](http://c.biancheng.net/view/272.html" \t "http://c.biancheng.net/view/_blank)》一文。**

介于流对象又可细分为输入流对象（istream）和输出流对象（ostream），C++ STL 标准库中，也对应的提供了 2 类流迭代器：

（1）将绑定到输入流对象的迭代器称为输入流迭代器（istream\_iterator），其可以用来读取输入流中的数据；

（2）将绑定到输出流对象的迭代器称为输出流迭代器（ostream\_iterator），其用来将数据写入到输出流中。  
接下来，就分别讲解这 2 个流迭代器的用法。

**一 C++ STL输入流迭代器（istream\_iterator）**

输入流迭代器用于直接从指定的输入流中读取元素，该类型迭代器本质上就是一个输入迭代器，这意味着假设 p 是一个输入流迭代器，则其只能进行 ++p、p++、\*p 操作，同时输入迭代器之间也只能使用 == 和 != 运算符。  
 实际上，输入流迭代器的底层是通过重载 ++ 运算符实现的，该运算符内部会调用operator >>读取数据。也就是说，假设 iit 为输入流迭代器，则只需要执行 ++iit 或者 iit++，即可读取一个指定类型的元素。  
 值得一提的是，istream\_iterator 定义在<iterator>头文件，并位于 std 命名空间中，因此使用此迭代器之前，程序中应包含如下语句：

#include <iterator>

using namespace std;

第二行代码不是必需的，但如果不用，则程序中在创建该类型的迭代器时，必须手动注明 std 命名空间（强烈建议初学者使用）。

创建输入流迭代器的方式有 3 种，分别为：  
1) 调用 istream\_iterator 模板类的默认构造函数，可以创建出一个具有结束标志的输入流迭代器。要知道，当我们从输入流中不断提取数据时，总有将流中数据全部提取完的那一时刻，这一时刻就可以用此方式构建的输入流迭代器表示。  
例如：

std::istream\_iterator<double> eos;

由此，即创建了一个可读取 double 类型元素，并代表结束标志的输入流迭代器。  
2) 除此之外，还可以创建一个可用来读取数据的输入流迭代器，比如：

std::istream\_iterator<double> iit(std::cin);

这里创建了一个可从标准输入流 cin 读取数据的输入流迭代器。值得注意的一点是，通过此方式创建的输入流迭代器，其调用的构造函数中，会自行尝试去指定流中读取一个指定类型的元素。  
3) istream\_iterator 模板类还支持用已创建好的 istream\_iterator 迭代器为新建 istream\_iterator 迭代器初始化，例如，在上面 iit 的基础上，再创建一个相同的 iit2 迭代器：

std::istream\_iterator<double> iit2(iit1);

由此，就创建好了一个和 iit1 完全相同的输入流迭代器。  
  
下面程序演示了输入流迭代器的用法：

#include <iostream>

#include <iterator>

using namespace *std*;

int *main*() {

//用于接收输入流中的数据

double value1, value2;

*cout* << "请输入 2 个小数: ";

//创建表示结束的输入流迭代器

*istream\_iterator*<double> eos;

//创建一个可逐个读取输入流中数据的迭代器，同时这里会让用户输入数据

*istream\_iterator*<double> iit(*cin*);

//判断输入流中是否有数据

if (iit != eos) {

//读取一个元素，并赋值给 value1

value1 = \*iit;

}

//如果输入流中此时没有数据，则用户要输入一个；反之，如果流中有数据，iit 迭代器后移一位，做读取下一个元素做准备

iit++;

if (iit != eos) {

//读取第二个元素，赋值给 value2

value2 = \*iit;

}

//输出读取到的 2 个元素

*cout* << "value1 = " << value1 << *endl*;

*cout* << "value2 = " << value2 << *endl*;

return 0;

}

**程序执行结果为：**

**请输入 2 个小数: 1.2 2.3  
value1 = 1.2  
value2 = 2.3**

**注意，只有读取到 EOF 流结束符时，程序中的 iit 才会和 eos 相等。另外，Windows 平台上使用 Ctrl+Z 组合键输入 ^Z 表示 EOF 流结束符，此结束符需要单独输入，或者输入换行符之后再输入才有效。**

**二 C++ STL输出流迭代器（ostream\_iterator）**

和输入流迭代器恰好相反，输出流迭代器用于将数据写到指定的输出流（如 cout）中。另外，该类型迭代器本质上属于输出迭代器，假设 p 为一个输出迭代器，则它能执行 ++p、p++、\*p=t 以及 \*p++=t 等类似操作。  
 其次，输出迭代器底层是通过重载赋值（=）运算符实现的，即借助该运算符，每个赋值给输出流迭代器的元素都会被写入到指定的输出流中。  
 值得一提的是，实现 ostream\_iterator 迭代器的模板类也定义在<iterator>头文件，并位于 std 命名空间中，因此在使用此类型迭代器时，程序也应该包含以下 2 行代码：

#include <iterator>

using namespace std;  
ostream\_iterator 模板类中也提供了 3 种创建 ostream\_iterator 迭代器的方法。  
  
1) 通过调用该模板类的默认构造函数，可以创建了一个指定输出流的迭代器：

std::ostream\_iterator<int> out\_it(std::cout);

由此，我们就创建了一个可将 int 类型元素写入到输出流（屏幕）中的迭代器。  
2) 在第一种方式的基础上，还可以为写入的元素之间指定一个分隔符，例如：

std::ostream\_iterator<int> out\_it(std::cout，",");

和第一种写入方式不同之处在于，此方式在向输出流写入 int 类型元素的同时，还会附带写入一个逗号（,）。  
3) 另外，在创建输出流迭代器时，可以用已有的同类型的迭代器，为其初始化。例如，利用上面已创建的 out\_it，再创建一个完全相同的 out\_it1：

std::ostream\_iterator<int> out\_it1(out\_it);  
下面程序演示了 ostream\_iterator 输出流迭代器的功能：

#include <iostream>

#include <iterator>

#include <string>

using namespace *std*;

int *main*() {

//创建一个输出流迭代器

*ostream\_iterator*<*string*> out\_it(*cout*);

//向 cout 输出流写入 string 字符串

\*out\_it = "http://c.biancheng.net/stl/";

*cout* << *endl*;

//创建一个输出流迭代器，设置分隔符 ,

*ostream\_iterator*<int> out\_it1(*cout*, ",");

//向 cout 输出流依次写入 1、2、3

\*out\_it1 = 1;

\*out\_it1 = 2;

\*out\_it1 = 3;

return 0;

}

**程序输出结果为：**

**http://c.biancheng.net/stl/  
1,2,3,**

在实际场景中，输出流迭代器常和 copy() 函数连用，即作为该函数第 3 个参数。比如：

#include <iostream>

#include <iterator>

#include <vector>

#include <algorithm> // std::copy

using namespace *std*;

int *main*() {

//创建一个 vector 容器

*vector*<int> myvector;

//初始化 myvector 容器

for (int i = 1; i < 10; ++i) {

myvector.*push\_back*(i);

}

//创建输出流迭代器

*std*::*ostream\_iterator*<int> out\_it(*std*::*cout*, ", ");

//将 myvector 容器中存储的元素写入到 cout 输出流中

*std*::*copy*(myvector.*begin*(), myvector.*end*(), out\_it);

return 0;

}

**程序执行结果为：**

**1, 2, 3, 4, 5, 6, 7, 8, 9,**

**有关 copy() 函数，由于不是本节重点，这里不再介绍，后续章节会做详细讲解。**

**第五 C++ STL流缓冲区迭代器（streambuf\_iterator）**

《[C++ STL流迭代器](http://c.biancheng.net/view/vip_7730.html" \t "http://c.biancheng.net/view/_blank)》一节中，讲解了输入流迭代器和输出迭代器的功能和用法，在此基础上，本节继续讲解输入流缓冲区迭代器。

**在学习本节之前，读者有必要先了解什么是缓冲区，可阅读《[进入缓冲区（缓存）的世界](http://c.biancheng.net/view/vip_1797.html" \t "http://c.biancheng.net/view/_blank)》一节做详细了解。**

我们知道在 C++ STL 标准库中，流迭代器又细分为输入流迭代器和输出流迭代器，流缓冲区迭代器也是如此，其又被细分为输入流缓冲区迭代器和输出流缓冲区迭代器：

（1）输入流缓冲区迭代器（istreambuf\_iterator）：从输入流缓冲区中读取字符元素；

（2）输出流缓冲区迭代器（ostreambuf\_iterator）：将连续的字符元素写入到输出缓冲区中。  
流缓冲区迭代器和流迭代器最大的区别在于，前者仅仅会将元素以字符的形式（包括 char、wchar\_t、char16\_t 及 char32\_t 等）读或者写到流缓冲区中，由于不会涉及数据类型的转换，读写数据的速度比后者要快。  
接下来，将一一对它们的功能和用法做讲解。

**一 C++ STL输入流缓冲区迭代器（istreambuf\_iterator）**

istreambuf\_iterator 输入流缓冲区迭代器的功能是从指定的流缓冲区中读取字符元素。  
值得一提的是，该类型迭代器本质是一个输入迭代器，即假设 p 是一个输入流迭代器，则其只能进行 ++p、p++、\*p 操作，同时迭代器之间也只能使用 == 和 != 运算符。  
 另外，实现该类型迭代器的模板类也定义在<iterator>头文件，并位于 std 命名空间中。因此，在创建并使用该类型迭代器之前，程序中应包含如下代码：

#include <iterator>

using namespace std;

第二行代码不是必需的，但如果不用，则程序中在创建该类型的迭代器时，必须手动注明 std 命名空间（强烈建议初学者使用）。

创建输入流缓冲区迭代器的常用方式，有以下 2 种：  
1) 通过调用 istreambuf\_iterator 模板类中的默认构造函数，可以创建一个表示结尾的输入流缓冲区迭代器。要知道，当我们从流缓冲区中不断读取数据时，总有读取完成的那一刻，这一刻就可以用此方式构建的流缓冲区迭代器表示。  
举个例子：

std::istreambuf\_iterator<char> end\_in;

其中，<> 尖括号中用于指定从流缓冲区中读取的字符类型。  
2) 当然，我们还可以指定要读取的流缓冲区，比如：

std::istreambuf\_iterator<char> in{ std::cin };

除此之外，还可以传入流缓冲区的地址，比如：

std::istreambuf\_iterator<char> in {std::cin.rdbuf()};

其中，rdbuf() 函数的功能是获取指定流缓冲区的地址。

无论是传入流缓冲区，还是传入其地址，它们最终构造的输入流缓冲区迭代器是一样的。

下面程序演示了输入流缓冲区迭代器的用法：

#include <iostream> // std::cin, std::cout

#include <iterator> // std::istreambuf\_iterator

#include <string> // std::string

using namespace *std*;

int *main*() {

//创建结束流缓冲区迭代器

*istreambuf\_iterator*<char> eos;

//创建一个从输入缓冲区读取字符元素的迭代器

*istreambuf\_iterator*<char> iit(*cin*);

*string* mystring;

*cout* << "向缓冲区输入元素：\n";

//不断从缓冲区读取数据，直到读取到 EOF 流结束符

while (iit != eos) {

mystring += \*iit++;

}

*cout* << "string：" << mystring;

return 0;

}

程序执行结果为：

**向缓冲区输入元素：  
abc ↙  
^Z ↙  
string：  
abc**

注意，只有读取到 EOF 流结束符时，程序中的 iit 才会和 eos 相等。在 Windows 平台上，使用 Ctrl+Z 组合键输入 ^Z 表示 EOF 流结束符，此结束符需要单独输入，或者输入换行符之后再输入才有效。

**二 C++ STL输出流缓冲区迭代器（ostreambuf\_iterator）**

和 istreambuf\_iterator 输入流缓冲区迭代器恰恰相反，ostreambuf\_iterator 输出流缓冲区迭代器用于将字符元素写入到指定的流缓冲区中。   
 实际上，该类型迭代器本质上是一个输出迭代器，这意味着假设 p 为一个输出迭代器，则它仅能执行 ++p、p++、\*p=t 以及 \*p++=t 操作。  
 另外，和 ostream\_iterator 输出流迭代器一样，istreambuf\_iterator 迭代器底层也是通过重载赋值（=）运算符实现的。换句话说，即通过赋值运算符，每个赋值给输出流缓冲区迭代器的字符元素，都会被写入到指定的流缓冲区中。  
 需要指出的是，istreambuf\_iterator 类模板也定义在<iterator>头文件，并位于 std 命名空间中，因此使用该类型迭代器，程序中需要包含以下代码：

#include <iterator>

using namespace std;  
在此基础上，创建输出流缓冲区迭代器的常用方式有以下 2 种：  
1) 通过传递一个流缓冲区对象，即可创建一个输出流缓冲区迭代器，比如：

std::ostreambuf\_iterator<char> out\_it (std::cout);

同样，尖括号 <> 中用于指定要写入字符的类型，可以是 char、wchar\_t、char16\_t 以及 char32\_t 等。  
  
2) 还可以借助 rdbuf()，传递一个流缓冲区的地址，也可以成功创建输出流缓冲区迭代器：

std::ostreambuf\_iterator<char> out\_it (std::cout.rdbuf());

下面程序演示了输出流缓冲区迭代器的用法：

#include <iostream> // std::cin, std::cout

#include <iterator> // std::ostreambuf\_iterator

#include <string> // std::string

#include <algorithm> // std::copy

int *main*() {

//创建一个和输出流缓冲区相关联的迭代器

*std*::*ostreambuf\_iterator*<char> out\_it(*std*::*cout*); // stdout iterator

//向输出流缓冲区中写入字符元素

\*out\_it = 'S';

\*out\_it = 'T';

\*out\_it = 'L';

//和 copy() 函数连用

*std*::*string* mystring("\nhttp://c.biancheng.net/stl/");

//将 mystring 中的字符串全部写入到输出流缓冲区中

*std*::*copy*(mystring.*begin*(), mystring.*end*(), out\_it);

return 0;

}

**程序执行结果为：**

**STL  
http://c.biancheng.net/stl/**

**有关 copy() 函数的具体用法，后续章节会做详细讲解。**

**第六 C++ STL move\_iterator移动迭代器用法详解**

C++ 11 还为 STL 标准库增添了一种迭代器适配器，即本节要讲的 move\_iterator 移动迭代器适配器。  
 move\_iterator 迭代器适配器，又可简称为移动迭代器，其可以实现以移动而非复制的方式，将某个区域空间中的元素移动至另一个指定的空间。  
举个例子，前面讲了 vector 容器，该类型容器支持如下初始化的方式（程序一）：

#include <iostream>

#include <vector>

#include <list>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化一个 vector 容器

*vector*<*string*> myvec{ "STL","Python","Java" };

//再次创建一个 vector 容器，利用 myvec 为其初始化

*vector*<*string*>othvec(myvec.*begin*(), myvec.*end*());

*cout* << "myvec:" << *endl*;

//输出 myvec 容器中的元素

for (auto ch : myvec) {

*cout* << ch << " ";

}

*cout* << *endl* << "othvec:" << *endl*;

//输出 othvec 容器中的元素

for (auto ch : othvec) {

*cout* << ch << " ";

}

return 0;

}

**程序执行结果为：**

**myvec:  
STL Python Java  
othvec:  
STL Python Java**

注意程序第 11 行，初始化 othvec 容器是通过复制 myvec 容器中的元素实现的。也就是说，othvec 容器从 myvec 容器中复制了一份 "STL"、"Python"、"Java" 并存储起来，此过程不会影响 myvec 容器。  
 那么，如果不想采用复制的方式，而就是想 myvec 容器中存储的元素全部移动到 othvec 容器中，该怎么办呢？没错，就是采用移动迭代器。  
 值得一提的是，实现移动迭代器的模板类定义在 <iterator> 头文件，并位于 std 命名空间中。因此，在使用该类型迭代器时，程序中应包含如下代码：

#include <iterator>

using namespace std;

第二行代码不是必需的，但如果不用，则程序中在创建该类型的迭代器时，必须手动注明 std 命名空间（强烈建议初学者使用）。  
**实现 move\_iterator 移动迭代器的模板类定义如下：**

**template <class Iterator>**

**class move\_iterator;**

**可以看到，在使用此迭代器时，需要传入一个基础迭代器 Iterator。**  
注意，此基础迭代器的类型虽然没有明确要求，但该模板类中某些成员方法的底层实现，需要此基础迭代器为双向迭代器或者随机访问迭代器。也就是说，如果指定的 Iterator 类型仅仅是输入迭代器，则某些成员方法将无法使用。

**实际上，在 move\_iterator 模板类中就包含有指定 Iterator 类型的基础迭代器，整个模板类也是借助此基础迭代器实现的。关于 move\_iterator 的底层实现，[C++ STL move\_iterator手册](http://www32.cplusplus.com/reference/iterator/move_iterator/" \t "http://c.biancheng.net/view/_blank)给出了详细的参考代码，有兴趣的读者可自行研究。**

**一 C++ STL move\_iterator的创建**

move\_iterator 模板类中，提供了 4 种创建 move\_iterator 迭代器的方法。  
1) 通过调用该模板类的默认构造函数，可以创建一个不指向任何对象的移动迭代器。比如：

//将 vector 容器的随机访问迭代器作为新建移动迭代器底层使用的基础迭代器

typedef std::vector<std::string>::iterator Iter;

//调用默认构造函数，创建移动迭代器

std::move\_iterator<Iter>mIter;

如果程序中引入了 std 命名空间，则上面代码中所有的 std:: 都可以省略。

由此，我们就创建好了一个 mIter 移动迭代器，该迭代器底层使用的是 vector 容器的随机访问迭代器，但这里没有为此基础迭代器明确指向，所以 mIter 迭代器也不知向任何对象。  
2) 当然，在创建 move\_iterator 迭代器的同时，也可以为其初始化。比如：

//创建一个 vector 容器

std::vector<std::string> myvec{ "one","two","three" };

//将 vector 容器的随机访问迭代器作为新建移动迭代器底层使用的基础迭代器

typedef std::vector<std::string>::iterator Iter;

//创建并初始化移动迭代器

std::move\_iterator<Iter>mIter(myvec.begin());

这里，我们创建了一个 mIter 移动迭代器，同时还为底层使用的随机访问迭代器做了初始化，即令其指向 myvec 容器的第一个元素。  
3) move\_iterator 模板类还支持用已有的移动迭代器初始化新建的同类型迭代器，比如，在上面创建好 mIter 迭代器的基础上，还可以向如下这样为新建的移动迭代器初始化：

std::move\_iterator<Iter>mIter2(mIter);

//还可以使用 = 运算符，它们是等价的

//std::move\_iterator<Iter>mIter2 = mIter;

这样创建的 mIter2 迭代器和 mIter 迭代器完全一样。也就是说，mIter2 底层会复制 mIter 迭代器底层使用的基础迭代器。  
4) 以上 3 种创建 move\_iterator 迭代器的方式，其本质都是直接调用 move\_iterator 模板类中的构造方法实现的。除此之外，C++ STL 标准库还提供了一个 make\_move\_iterator()函数，通过调用此函数可以快速创建一个 move\_iterator 迭代器。  
**C++ STL 标准库中，make\_move\_iterator() 是以函数模板的形式提供的，其语法格式如下：**

**template <class Iterator>  
  move\_iterator<Iterator> make\_move\_iterator (const Iterator& it);**

**其中，参数 it 为基础迭代器，用于初始化新建迭代器。同时，该函数会返回一个创建好的移动迭代器。**  
举个例子：

typedef std::vector<std::string>::iterator Iter;

std::vector<std::string> myvec{ "one","two","three" };

//将 make\_move\_iterator() 的返回值赋值给同类型的 mIter 迭代器

std::move\_iterator<Iter>mIter = make\_move\_iterator(myvec.begin());

下面程序对程序一做了修改，即运用移动迭代器为 othvec 容器初始化：

#include <iostream>

#include <vector>

#include <list>

#include <string>

using namespace *std*;

int *main*()

{

//创建并初始化一个 vector 容器

*vector*<*string*> myvec{ "STL","Python","Java" };

//再次创建一个 vector 容器，利用 myvec 为其初始化

*vector*<*string*>othvec(*make\_move\_iterator*(myvec.*begin*()), *make\_move\_iterator*(myvec.*end*()));

*cout* << "myvec:" << *endl*;

//输出 myvec 容器中的元素

for (auto ch : myvec) {

*cout* << ch << " ";

}

*cout* << *endl* << "othvec:" << *endl*;

//输出 othvec 容器中的元素

for (auto ch : othvec) {

*cout* << ch << " ";

}

return 0;

}

**程序执行结果为：**

**myvec:  
  
othvec:  
STL Python Java**

通过和程序一做对比不难看出它们的区别，由于程序第 11 行为 othvec 容器初始化时，使用的是移动迭代器，其会将 myvec 容器中的元素直接移动到 othvec 容器中。

**注意，即便通过移动迭代器将容器中某区域的元素移动到了其他容器中，该区域内仍可能残留有之前存储的元素，但这些元素是不能再被使用的，否则极有可能使程序产生各种其他错误。**  
 和其他迭代器适配器一样，move\_iterator 模板类中也提供有 base() 成员方法，通过该方法，我们可以获取到当前移动迭代器底层所使用的基础迭代器。  
举个例子：

#include <iostream>

#include <vector>

#include <list>

#include <string>

using namespace *std*;

int *main*()

{

typedef *std*::*vector*<*std*::*string*>::*iterator* Iter;

//创建并初始化一个 vector 容器

*vector*<*std*::*string*> myvec{ "STL","Java","Python" };

//创建 2 个移动迭代器

*std*::*move\_iterator*<Iter>begin = *make\_move\_iterator*(myvec.*begin*());

*std*::*move\_iterator*<Iter>end = *make\_move\_iterator*(myvec.*end*());

//以复制的方式初始化 othvec 容器

*vector* <*std*::*string*> othvec(begin.*base*(), end.*base*());

*cout* << "myvec:" << *endl*;

//输出 myvec 容器中的元素

for (auto ch : myvec) {

*cout* << ch << " ";

}

*cout* << *endl* << "othvec:" << *endl*;

//输出 othvec 容器中的元素

for (auto ch : othvec) {

*cout* << ch << " ";

}

return 0;

}

**程序执行结果为：**

**myvec:  
STL Java Python  
othvec:  
STL Java Python**

显然，通过调用 base() 成员方法，初始化 othvec 容器的方式转变为以复制而非移动的方式，因此 myvec 容器不会受到影响。

**第七 C++ STL advance()函数用法详解**

通过前面的学习，已经讲解了 C++ STL 标准库中所有的基础迭代器以及迭代器适配器的用法。但除此之外，为了方便用户操作这些迭代器，C++ STL 标准库中还提供有一些辅助函数，如表 1 所示。

表 1 C++ STL迭代器辅助函数

|  |  |
| --- | --- |
| 迭代器辅助函数 | 功能 |
| advance(it, n) | it 表示某个迭代器，n 为整数。该函数的功能是将 it 迭代器前进或后退 n 个位置。 |
| distance(first, last) | first 和 last 都是迭代器，该函数的功能是计算 first 和 last 之间的距离。 |
| begin(cont) | cont 表示某个容器，该函数可以返回一个指向 cont 容器中第一个元素的迭代器。 |
| end(cont) | cont 表示某个容器，该函数可以返回一个指向 cont 容器中最后一个元素之后位置的迭代器。 |
| prev(it) | it 为指定的迭代器，该函数默认可以返回一个指向上一个位置处的迭代器。注意，it 至少为双向迭代器。 |
| next(it) | it 为指定的迭代器，该函数默认可以返回一个指向下一个位置处的迭代器。注意，it 最少为前向迭代器。 |

本节先讲解 advance() 函数，其他函数后续章节会做详细介绍。

**一 C++ STL advance()函数**

**advance() 函数用于将迭代器前进（或者后退）指定长度的距离，其语法格式如下：**

**template <class InputIterator, class Distance>  
    void advance (InputIterator& it, Distance n);**

**其中 it 指的是目标迭代器，n 通常为一个整数。**  
需要注意的是，如果 it 为输入迭代器或者前向迭代器，则 n 必须为一个正数，即表示将 it 右移（前进） n 个位置；反之，如果 it 为双向迭代器或者随机访问迭代器，则 n 为正数时表示将 it 右移（前进） n 个位置，n 为负数时表示将 it 左移（后退） n 个位置。  
 另外，根据 it 类型是否为随机访问迭代器，advance() 函数底层采用了不同的实现机制：

（1）当 it 为随机访问迭代器时，由于该类型迭代器支持 p+n 或者 p-n（其中 p 就是一个随机访问迭代器）运算，advance() 函数底层采用的就是 it+n 操作实现的；

（2）当 it 为其他类型迭代器时，它们仅支持进行 ++ 或者 -- 运算，这种情况下，advance() 函数底层是通过重复执行 n 个 ++ 或者 -- 操作实现的。  
 值得一提的是，advance() 函数定义在<iterator>头文件，并位于 std 命名空间中。因此，程序在使用该函数之前，应包含如下 2 行代码：

#include <iterator>

using namespace std;

第二行代码不是必须的，但如果不引用，则后续在使用 advance() 函数时，需要额外标注 std 命名空间（强烈建议初学者使用）。

为了让读者更好地知晓 advance() 函数的功能，首先以 forward\_list 容器（仅支持使用前向迭代器）为例，下面程序演示了 advance() 函数的功能：

#include <iostream> // std::cout

#include <iterator> // std::advance

#include <forward\_list>

using namespace *std*;

int *main*() {

//创建一个 forward\_list 容器

*forward\_list*<int> mylist{ 1,2,3,4 };

//it为前向迭代器，其指向 mylist 容器中第一个元素

*forward\_list*<int>::*iterator* it = mylist.*begin*();

//借助 advance() 函数将 it 迭代器前进 2 个位置

*advance*(it, 2);

*cout* << "\*it = " << \*it;

return 0;

}

**程序执行结果为：**

**\*it = 3**

**此程序中，由于 it 为前向迭代器，其只能进行 ++ 操作，即只能前进（右移），所以 advance() 函数的第 2 个参数只能为正数。**  
  
下面程序以 vector 容器为例，演示了 advance() 函数的功能：

#include <iostream> // std::cout

#include <iterator> // std::advance

#include <vector>

using namespace *std*;

int *main*() {

//创建一个 vector 容器

*vector*<int> myvector{ 1,2,3,4 };

//it为随机访问迭代器，其指向 myvector 容器中第一个元素

*vector*<int>::*iterator* it = myvector.*begin*();

//借助 advance() 函数将 it 迭代器前进 2 个位置

*advance*(it, 2);

*cout* << "1、\*it = " << \*it << *endl*;

//继续使用it，其指向 myvector 容器中最后一个元素之后的位置

it = myvector.*end*();

//借助 advance() 函数将 it 迭代器后退 3 个位置

*advance*(it, -3);

*cout* << "2、\*it = " << \*it;

return 0;

}

**程序执行结果为：**

**1、\*it = 3  
2、\*it = 2**

**注意，advance() 函数本身不会检测 it 迭代器移动 n 个位置的可行性，如果 it 迭代器的移动位置超出了合理范围，it 迭代器的指向将无法保证，此时使用 \*it 将会导致程序崩溃。**

**第八 C++ STL distance()函数用法详解**

我们知道，作用于同一容器的 2 个同类型迭代器可以有效指定一个区间范围。在此基础上，如果想获取该指定范围内包含元素的个数，就可以借助本节要讲的 distance() 函数。  
**distance() 函数用于计算两个迭代器表示的范围内包含元素的个数，其语法格式如下：**

**template<class InputIterator>  
  typename iterator\_traits<InputIterator>::difference\_type distance (InputIterator first, InputIterator last);**

其中，first 和 last 都为迭代器，其类型可以是输入迭代器、前向迭代器、双向迭代器以及随机访问迭代器；该函数会返回[first, last)范围内包含的元素的个数。  
注意，first 和 last 的迭代器类型，直接决定了 distance() 函数底层的实现机制：

（1）当 first 和 last 为随机访问迭代器时，distance() 底层直接采用 last - first 求得 [first, last) 范围内包含元素的个数，其时间复杂度为O(1)常数阶；

（2）当 first 和 last 为非随机访问迭代器时，distance() 底层通过不断执行 ++first（或者 first++）直到 first==last，由此来获取 [first, last) 范围内包含元素的个数，其时间复杂度为O(n)线性阶。  
另外，distance() 函数定义在<iterator>头文件，并位于 std 命名空间中。因此在使用此函数前，程序中应包含如下代码：

#include <iterator>

using namespace std;  
下面程序以 list 容器（其迭代器类型为双向迭代器）为例，演示了 distance() 函数的用法：

#include <iostream> // std::cout

#include <iterator> // std::distance

#include <list> // std::list

using namespace *std*;

int *main*() {

//创建一个空 list 容器

*list*<int> mylist;

//向空 list 容器中添加元素 0~9

for (int i = 0; i < 10; i++) {

mylist.*push\_back*(i);

}

//指定 2 个双向迭代器，用于执行某个区间

*list*<int>::*iterator* first = mylist.*begin*();//指向元素 0

*list*<int>::*iterator* last = mylist.*end*();//指向元素 9 之后的位置

//获取 [first,last) 范围内包含元素的个数

*cout* << "distance() = " << *distance*(first, last);

return 0;

}

**程序执行结果为：**

**distance() = 10**

**第九 如何将const\_iterator转换为iterator类型迭代器？**

前面章节中，已经详细介绍了 advance() 和 distance() 函数各自的功能和用法。在此基础上，本节继续讲解如何利用这 2 个函数实现将 const\_iterator 迭代器转换为 iterator 迭代器，或者将 const\_reverse\_iterator 迭代器转换为 reverse\_iterator 迭代器。

**注意，上面提到的 iterator、const\_iterator、reverse\_iterator 和 const\_reverse\_iterator 是 C++ STL 标准库提供了 4 种基础迭代器，关于它们各自的特性和功能可以阅读 《[C++ STL迭代器](http://c.biancheng.net/view/6675.html" \t "http://c.biancheng.net/view/_blank)》一节，这里不再重复赘述。**

要知道，C++ STL标准库为了方便用户更轻松地操作容器，每个容器的模板类都提供有丰富且实用的方法。在这些方法中，有些是以 const\_iterator 类型迭代器作为参数，也就意味着在使用此类方法时，需要为其传入一个 const\_iterator 类型的迭代器。  
**例如，vector 容器模板类中提供有 insert() 方法，该方法的语法格式如下：**

**iterator insert (const\_iterator position, const value\_type& val);**

**注意，此方法有多种语法格式，这里仅列举了其中的一种。有关该方法的具体用法，读者可阅读《[C++ STL vector插入元素](http://c.biancheng.net/view/6834.html" \t "http://c.biancheng.net/view/_blank)》一节，这里不再做详细赘述。**

可以看到，如果想调用此格式的 insert() 方法，就需要为其传入一个 const\_iterator 类型的迭代器。例如：

#include <iostream>

#include <vector>

using namespace *std*;

int *main*()

{

*vector*<int>value{ 1,2,3,4,5 };

//定义一个 const\_iterator 类型的迭代器

*vector*<int>::*const\_iterator* citer = value.*cbegin*();

value.*insert*(citer, 10);

for (auto iter = value.*begin*(); iter != value.*end*(); ++iter) {

*cout* << \*iter << " ";

}

return 0;

}

**程序执行结果为：**

**10 1 2 3 4 5**

**显然通过调用 insert() 方法，并将指向 value 容器中元素 1 位置处的 const\_iterator 类型迭代器作为该方法的实参，就成功将 10 插入到了 value 容器的指定位置。**  
 那么，是不是给 insert() 方法传递其它类型迭代器就不行呢？当然不是，对于给 const\_iterator 类型的迭代器传值，还可以使用 iterator 类型迭代器，但不能使用 const\_reverse\_iterator 和 reverse\_iterator 类型迭代器，这是为什么呢？  
 实际上，当我们将某一类型的迭代器传递给 insert() 方法中 const\_iterator 类型的 position 形参时，即便类型不匹配，编译器也不会立即报错，而是先尝试将其类型转换成 const\_iterator 类型，如果转换成功，则程序仍可以正常执行；反之如果转换失败，编译器才会报错。

**C++ 中，通常将编译器自行尝试进行类型转换的整个过程称为隐式转换（或者自动类型转换）。**

对于 C++ STL 标准库中的这 4 种基础迭代器来说，C++ 编译器的隐式转换仅支持以下 2 种情况：

（1）将 iterator 类型的迭代器隐式转换为 const\_iterator 类型的迭代器；

（2）将 reverse\_iterator 类型的迭代器隐式转换为 const\_reverse\_iterator 类型的迭代器。

**注意，以上 2 种隐式转换是单向的，即编译器只支持从 iterator 转换为 const\_iterator，从 reverse\_iterator 转换为 const\_reverse\_iterator，但不支持逆向转换。**

有些读者可能会好奇，既然隐式转换无法做到，还有其他方式可以实现从 const\_iterator 到 iterator、从 const\_reverse\_iterator 到 reverse\_iterator 的转换吗？  
 很多读者可能会想到使用强制类型转换（const\_cast）的方式。但可以明确的是，强制类型转换并不适用于迭代器，因为 const\_cast 的功能仅是去掉某个类型的 const 修饰符，但 const\_iterator 和iterator 是完全不同的 2 个类，同样 const\_reverse\_iterator 和 reverse\_iterator 也是完全不同的 2 个类，它们仅仅是类名有 const 的差别，但并不是 const T 和 T 的关系。  
  
 这里给读者推荐一种实现方式，就是使用 advance() 和 distance() 这 2 个函数，其语法格式如下：

**//将 const\_iterator 转换为 iterator  
advance(iter, distance<cont<T>::const\_iterator>(iter,citer));  
//将 const\_reverse\_iterator 转换为 reverse\_iterator  
advance(iter, distance<cont<T>::const\_reverse\_iterator>(iter,citer));**

**其中，citer 为指向某个容器（比如 cont）任意位置的 const\_iterator（或者 const\_reverse\_iterator）类型迭代器，而 iter 通常初始为指向 cont 容器中第一个元素的 iterator（或者 reverse\_iterator）类型迭代器。通过套用此格式，最终 iter 会变成一个指向和 citer 一样的 iterator（或者 reverse\_iterator）类型迭代器。**

注意，在使用 distance() 函数时，必须额外指明 2 个参数为 const 迭代器类型，否则会因为传入的 iter 和 citer 类型不一致导致 distance() 函数编译出错。

该实现方式的本质是，先创建一个迭代器 citer，并将其初始化为指向容器中第一个元素的位置。在此基础上，通过计算和目标迭代器 iter 的距离（调用 distance()），将其移动至和 iter 同一个位置（调用 advance()），由此就可以间接得到一个指向同一位置的 iter 迭代器。  
  
举个例子：

#include <iostream>

#include <vector>

using namespace *std*;

int *main*()

{

*vector*<int>value{ 1,2,3,4,5 };

//定义一个 const\_iterator 类型的迭代器，其指向最后一个元素

*vector*<int>::*const\_iterator* citer = --value.*cend*();

//初始化一个非 const 迭代器，另其指向

*vector*<int>::*iterator* iter = value.*begin*();

//将 iter 变成和 citer 同样指向的迭代器

*advance*(iter, *distance*<*vector*<int>::*const\_iterator*>(iter, citer));

*cout* << "\*citer = " << \*citer << *endl*;

*cout* << "\*iter = " << \*iter << *endl*;

return 0;

}

**程序执行结果为：**

**\*citer = 5  
\*iter = 5**

可以看到，通过使用 advance() 和 distance() 函数的组合格式，最终可以得到一个和 citer 指向相同但类型为 iterator 的迭代器。

注意，此方法的实现效率仍取决于目标容器的迭代器类型，如果是随机访问迭代器，则该方法的执行效率为 O(1)；反之，则执行效率为 O(n)。

**第十 C++ STL begin()和end()函数用法**

在前面章节中，我们已经对 C++ STL标准库提供的所有容器做了系统的讲解。读者可能已经注意到，无论是序列式容器还是关联式容器（包括哈希容器），不仅模板类内部提供有 begin() 和 end() 成员方法，C++ STL 标准库中还提供有同名且具有相同功能的 begin() 和 end() 函数。  
 首先需要说明的是，begin() 和 end() 是以函数模板的形式定义的，但它们的模板并没有位于某一个头文件中，而是很多头文件中都有它们的定义。

**C++ STL 标准库中，包含 begin() 和 end() 函数模板的头文件包括：<iterator>, <array>, <deque>, <forward\_list>, <list>, <map>, <regex>（正则表达式的头文件）, <set>, <string>, <unordered\_map>, <unordered\_set> 以及 <vector>。**

不仅如此，begin() 和 end() 都位于 std 命名空间中。因此，在使用这 2 个函数之前，程序中应引入容纳它们函数模板的头文件以及 std 命名空间。  
 在实际的使用场景中，begin() 和 end() 函数往往会一起使用的。根据作用对象的不同，begin() 和 end() 函数可细分为以下 2 个功能。

**一begin()和end()参数为容器**

当将某个具体容器（比如 cont）作为参数分别传给 begin() 和 end() 函数时，其中 begin() 底层会执行 cont.begin() 语句，而 end() 底层会执行 cont.end() 语句，它们最终会将得到的迭代器作为函数的返回值反馈回来。  
 当作用对象为容器时，end() 和 begin() 函数的语法格式是完全一样的，这里以 begin() 函数为例，有以下 2 种格式：

**//① 非 const 修改的容器作为参数，begin() 函数返回的为非 const 类型的迭代器  
template <class Container>  
    auto begin (Container& cont)  
//② 传入 const 修饰的容器，begin() 函数返回的为 const 类型的迭代器  
template <class Container>  
    auto begin (const Container& cont)**

其中，cont 表示指定的容器；同时，函数会返回一个有特定指向的迭代器，且此迭代器的类型也取决于 cont 容器。

**以上 2 种格式的区别仅在与传入的容器是否有 const 修饰，即如果有，则通过该函数获得的迭代器也有 const 修饰（不能用于修改容器中存储的数据）；反之就没有。**

举个例子：

#include <iostream> // std::cout

#include <vector> // std::vector, std::begin, std::end

using namespace *std*;

int *main*() {

//创建并初始化 vector 容器

*std*::*vector*<int> myvector{ 1,2,3,4,5 };

//调用 begin() 和 end() 函数遍历 myvector 容器

for (auto it = *begin*(myvector); it != *end*(myvector); ++it)

*cout* << \*it << ' ';

return 0;

}

**程序执行结果为：**

**1 2 3 4 5**

**程序第 8 行中，begin(myvector) 等同于执行 myvector.begin()，而 end(myvector) 也等同于执行 myvector.end()。**

**二 begin()和end()参数为数组**

除了可以将指定容器作为参数传给 begin() 和 end() 之外，还可以指定数组作为参数传给它们。  
 将指定数组传给 begin() 函数，其会返回一个指向该数组首个元素的指针；将指定数组传给 end() 函数，其会返回一个指向数组中最后一个元素之后位置的指针。  
 同样，数组作为参数时，end() 函数的语法格式和 begin() 函数也完全一样，这里仅给出了 begin() 函数的语法格式：

**template <class T, size\_t N>  
  T\* begin (T(&arr)[N]);**

其中 T 为数组中存储元素的类型，N 为数组的长度；(&arr)[N] 表示以引用的方式传递数组作为参数。  
  
举个例子：

#include <iostream> // std::cout

#include <vector> // std::vector, std::begin, std::end

using namespace *std*;

int *main*() {

//定义一个普通数组

int arr[] = { 1,2,3,4,5 };

//创建一个空 vector 容器

*vector*<int> myvector;

//将数组中的元素添加到 myvector 容器中存储

for (int\* it = *begin*(arr); it != *end*(arr); ++it)

myvector.*push\_back*(\*it);

//输出 myvector 容器中存储的元素

for (auto it = myvector.*begin*(); it != myvector.*end*(); ++it)

*cout* << \*it << ' ';

return 0;

}

**程序执行结果为：**

**1 2 3 4 5**

注意程序中第 10 行，这里用整数指针 it 接收 begin(arr) 的返回值，同时该循环会一直循环到 it 指向 arr 数组中最后一个元素之后的位置。

**第十一 C++ STL prev()和next()函数用法详解**

《[C++ STL advance()函数](http://c.biancheng.net/view/7370.html" \t "http://c.biancheng.net/view/_blank)》一节中，详细讲解了 advance() 函数的功能，其可以将指定迭代器前移或后移 n 个位置的距离。  
但值得一提的是，advance() 函数移动的是源迭代器，举个例子：

#include <iostream> // std::cout

#include <iterator> // std::advance

#include <vector>

using namespace *std*;

int *main*() {

//创建一个 vector 容器

*vector*<int> myvector{ 1,2,3,4 };

//it为随机访问迭代器，其指向 myvector 容器中第一个元素

*vector*<int>::*iterator* it = myvector.*begin*();

//输出 it 迭代器指向的数据

*cout* << "移动前的 \*it = " << \*it << *endl*;

//借助 advance() 函数将 it 迭代器前进 2 个位置

*advance*(it, 2);

*cout* << "移动后的 \*it = " << \*it << *endl*;

return 0;

}

**程序执行结果为：**

**移动前的 \*it = 1  
移动后的 \*it = 3**

通过程序的运行结果不难看出，advance() 函数没有任何返回值，其移动的是 it 迭代器本身。  
  
这就产生一个问题，若我们不想移动 it 迭代器本身，而仅仅是想在 it 迭代器的基础上，得到一个移动指定位置的新迭代器，显然 advance() 函数是不合适的，这时就可以使用 C++ STL 标准库提供的另外 2 个函数，即 prev() 和 next() 函数。

**一 C++ STL prev()函数**

prev 原意为“上一个”，但 prev() 的功能远比它的本意大得多，该函数可用来获取一个距离指定迭代器 n 个元素的迭代器。  
**prev() 函数的语法格式如下：**

**template <class BidirectionalIterator>  
    BidirectionalIterator prev (BidirectionalIterator it, typename iterator\_traits<BidirectionalIterator>::difference\_type n = 1);**

其中，it 为源迭代器，其类型只能为双向迭代器或者随机访问迭代器；n 为指定新迭代器距离 it 的距离，默认值为 1。该函数会返回一个距离 it 迭代器 n 个元素的新迭代器。

**注意，当 n 为正数时，其返回的迭代器将位于 it 左侧；反之，当 n 为负数时，其返回的迭代器位于 it 右侧。**

举个例子：

#include <iostream> // std::cout

#include <iterator> // std::next

#include <list> // std::list

using namespace *std*;

int *main*() {

//创建并初始化一个 list 容器

*std*::*list*<int> mylist{ 1,2,3,4,5 };

*std*::*list*<int>::*iterator* it = mylist.*end*();

//获取一个距离 it 迭代器 2 个元素的迭代器，由于 2 为正数，newit 位于 it 左侧

auto newit = *prev*(it, 2);

*cout* << "prev(it, 2) = " << \*newit << *endl*;

//n为负数，newit 位于 it 右侧

it = mylist.*begin*();

newit = *prev*(it, -2);

*cout* << "prev(it, -2) = " << \*newit;

return 0;

}

**程序执行结果为：**

**prev(it, 2) = 4  
prev(it, -2) = 3**

可以看到，当 it 指向 mylist 容器最后一个元素之后的位置时，通过 prev(it, 2) 可以获得一个新迭代器 newit，其指向的是距离 it 左侧 2 个元素的位置（其存储的是元素 4）；当 it 指向 mylist 容器中首个元素时，通过 prev(it, -2) 可以获得一个指向距离 it 右侧 2 个位置处的新迭代器。

**注意，prev() 函数自身不会检验新迭代器的指向是否合理，需要我们自己来保证其合理性。**

**二 C++ STL next()函数**

和 prev 相反，next 原意为“下一个”，但其功能和 prev() 函数类似，即用来获取一个距离指定迭代器 n 个元素的迭代器。  
  
**next() 函数的语法格式如下：**

**template <class ForwardIterator>  
    ForwardIterator next (ForwardIterator it, typename iterator\_traits<ForwardIterator>::difference\_type n = 1);**

其中 it 为源迭代器，其类似可以为前向迭代器、双向迭代器以及随机访问迭代器；n 为指定新迭代器距离 it 的距离，默认值为 1。该函数会返回一个距离 it 迭代器 n 个元素的新迭代器。

需要注意的是，当 it 为前向迭代器时，n 只能为正数，该函数最终得到的新迭代器位于 it 右侧；当 it 为双向迭代器或者随机访问迭代器时，若 n 为正数，则得到的新迭代器位于 it 右侧，反之位于 it 左侧。

举个例子：

#include <iostream> // std::cout

#include <iterator> // std::next

#include <list> // std::list

using namespace *std*;

int *main*() {

//创建并初始化一个 list 容器

*std*::*list*<int> mylist{ 1,2,3,4,5 };

*std*::*list*<int>::*iterator* it = mylist.*begin*();

//获取一个距离 it 迭代器 2 个元素的迭代器，由于 2 为正数，newit 位于 it 右侧

auto newit = *next*(it, 2);

*cout* << "next(it, 2) = " << \*newit << *endl*;

//n为负数，newit 位于 it 左侧

it = mylist.*end*();

newit = *next*(it, -2);

*cout* << "next(it, -2) = " << \*newit;

return 0;

}

**程序执行结果为：**

**next(it, 2) = 3  
next(it, -2) = 4**

可以看到，和 prev() 函数恰好相反，当 n 值为 2 时，next(it, 2) 函数获得的新迭代器位于 it 迭代器的右侧，距离 2 个元素；反之，当 n 值为 -2 时，新迭代器位于 it 迭代器的左侧，距离 2 个元素。

注意，和 prev() 函数一样，next() 函数自身也不会检查新迭代器指向的有效性，需要我们自己来保证。

**第七章 C++ STL常用算法（排序、合并、搜索和分区）**

本章将介绍一些同排序和合并松散关联的算法。其中有两组算法专门提供排序和合并功能，一组为给定值范围内的元素提供分区机制，另一组提供了在范围内查找一个或多个元素的方法。

与此同时，本章还会介绍STL提供的更多算法。这些算法通常可以分为两类：会改变它们所应用序列的算法以及不改变它们所应用序列的算法。本章会按照它们的用法分类，而不会按照是否改变序列来分类。如果知道算法做了些什么，显然就会知道它是否改变它所应用的数据。

**第一 C++ sort()排序函数用法详解**

C++ STL 标准库提供有很多实用的排序函数，如表 1 所示。通过调用它们，我们可以很轻松地实现对普通数组或者容器中指定范围内的元素进行排序。

表 1 C++ STL 排序函数

|  |  |
| --- | --- |
| 函数名 | 用法 |
| sort (first, last) | 对容器或普通数组中 [first, last) 范围内的元素进行排序，默认进行升序排序。 |
| stable\_sort (first, last) | 和 sort() 函数功能相似，不同之处在于，对于 [first, last) 范围内值相同的元素，该函数不会改变它们的相对位置。 |
| partial\_sort (first, middle, last) | 从 [first,last) 范围内，筛选出 muddle-first 个最小的元素并排序存放在 [first，middle) 区间中。 |
| partial\_sort\_copy (first, last, result\_first, result\_last) | 从 [first, last) 范围内筛选出 result\_last-result\_first 个元素排序并存储到 [result\_first, result\_last) 指定的范围中。 |
| is\_sorted (first, last) | 检测 [first, last) 范围内是否已经排好序，默认检测是否按升序排序。 |
| is\_sorted\_until (first, last) | 和 is\_sorted() 函数功能类似，唯一的区别在于，如果 [first, last) 范围的元素没有排好序，则该函数会返回一个指向首个不遵循排序规则的元素的迭代器。 |
| void nth\_element (first, nth, last) | 找到 [first, last) 范围内按照排序规则（默认按照升序排序）应该位于第 nth 个位置处的元素，并将其放置到此位置。同时使该位置左侧的所有元素都比其存放的元素小，该位置右侧的所有元素都比其存放的元素大。 |

对于表 1 中罗列的这些函数，本教程会一一进行讲解，这里先介绍 sort() 函数。

**一 C++ sort()排序函数**

C++ STL 标准库中的 sort()  函数，本质就是一个模板函数。正如表 1 中描述的，该函数专门用来对容器或普通数组中指定范围内的元素进行排序，排序规则默认以元素值的大小做升序排序，除此之外我们也可以选择标准库提供的其它排序规则（比如std::greater<T>降序排序规则），甚至还可以自定义排序规则。

**sort() 函数是基于快速排序实现的，有关快速排序的具体实现过程，感兴趣的读者可阅读《[快速排序（QSort，快排）算法](http://c.biancheng.net/view/vip_3445.html" \t "http://c.biancheng.net/view/_blank)》一文。**

需要注意的是，sort() 函数受到底层实现方式的限制，它仅适用于普通数组和部分类型的容器。换句话说，只有普通数组和具备以下条件的容器，才能使用 sort() 函数：

（1）容器支持的迭代器类型必须为随机访问迭代器。这意味着，sort() 只对 array、vector、deque 这 3 个容器提供支持。

（2）如果对容器中指定区域的元素做默认升序排序，则元素类型必须支持<小于运算符；同样，如果选用标准库提供的其它排序规则，元素类型也必须支持该规则底层实现所用的比较运算符；

（3）sort() 函数在实现排序时，需要交换容器中元素的存储位置。这种情况下，如果容器中存储的是自定义的类对象，则该类的内部必须提供移动构造函数和移动赋值运算符。

另外还需要注意的一点是，对于指定区域内值相等的元素，sort() 函数无法保证它们的相对位置不发生改变。例如，有如下一组数据：

**2 1 2 3 2**

可以看到，该组数据中包含多个值为 2 的元素，此时如果使用 sort() 函数进行排序，则值为 2 的这 3 个元素的相对位置可能会发生改变，比如排序结果为：

**1 2 2 2 3**

可以看到，原本红色的元素 2 位于绿色 2 和橙色 2 的左侧，但经过 sort() 函数排序之后，它们的相对位置发生了改变，即红色 2 移动到了绿色 2 和橙色 2 的右侧。

实际场景中，如果需要保证值相等元素的相对位置不发生改变，可以选用 stable\_sort() 排序函数。有关该函数的具体用法，后续章节会做详细讲解。   
 值得一提的是，sort() 函数位于<algorithm>头文件中，因此在使用该函数前，程序中应包含如下语句：

#include <algorithm>

**sort() 函数有 2 种用法，其语法格式分别为：**

**//对 [first, last) 区域内的元素做默认的升序排序**

**void sort (RandomAccessIterator first, RandomAccessIterator last);**

**//按照指定的 comp 排序规则，对 [first, last) 区域内的元素进行排序**

**void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);**

其中，first 和 last 都为随机访问迭代器，它们的组合 [first, last) 用来指定要排序的目标区域；另外在第 2 种格式中，comp 可以是 C++ STL 标准库提供的排序规则（比如 std::greater<T>），也可以是自定义的排序规则。

**关于如何自定义一个排序规则，除了《[C++ STL关联式容器自定义排序规则](http://c.biancheng.net/view/7213.html" \t "http://c.biancheng.net/view/_blank)》一节介绍的 2 种方式外，还可以直接定义一个具有 2 个参数并返回 bool 类型值的函数作为排序规则。**

举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::sort

#include <vector> // std::vector

//以普通函数的方式实现自定义排序规则

bool mycomp(int i, int j) {

return (i < j);

}

//以函数对象的方式实现自定义排序规则

class mycomp2 {

public:

bool operator() (int i, int j) {

return (i < j);

}

};

int *main*() {

*std*::*vector*<int> myvector{ 32, 71, 12, 45, 26, 80, 53, 33 };

//调用第一种语法格式，对 32、71、12、45 进行排序

*std*::*sort*(myvector.*begin*(), myvector.*begin*() + 4); //(12 32 45 71) 26 80 53 33

//调用第二种语法格式，利用STL标准库提供的其它比较规则（比如 greater<T>）进行排序

*std*::*sort*(myvector.*begin*(), myvector.*begin*() + 4, *std*::*greater*<int>()); //(71 45 32 12) 26 80 53 33

//调用第二种语法格式，通过自定义比较规则进行排序

*std*::*sort*(myvector.*begin*(), myvector.*end*(), mycomp2());//12 26 32 33 45 53 71 80

//输出 myvector 容器中的元素

for (*std*::*vector*<int>::*iterator* it = myvector.*begin*(); it != myvector.*end*(); ++it) {

*std*::*cout* << \*it << ' ';

}

return 0;

}

**程序执行结果为：**

**12 26 32 33 45 53 71 80**

可以看到，程序中分别以函数和函数对象的方式自定义了具有相同功能的 mycomp 和 mycomp2 升序排序规则。需要注意的是，和为关联式容器设定排序规则不同，给 sort() 函数指定排序规则时，需要为其传入一个函数名（例如 mycomp ）或者函数对象（例如 std::greater<int>() 或者 mycomp2()）。  
  
那么，sort() 函数的效率怎么样吗？该函数实现排序的平均时间复杂度为N\*log2N（其中 N 为指定区域 [first, last) 中 last 和 first 的距离）。

**第二 C++ stable\_sort()用法详解**

通过阅读《[C++ sort()排序函数](http://c.biancheng.net/view/7457.html" \t "http://c.biancheng.net/view/_blank)》一节，读者已经了解了 sort() 函数的功能和用法。值得一提的是，当指定范围内包含多个相等的元素时，sort() 排序函数无法保证不改变它们的相对位置。那么，如果既要完成排序又要保证相等元素的相对位置，该怎么办呢？可以使用 stable\_sort() 函数。

**有些场景是需要保证相等元素的相对位置的。例如对于一个保存某种事务（比如银行账户）的容器，在处理这些事务之前，为了能够有序更新这些账户，需要按照账号对它们进行排序。而这时就很有可能出现相等的账号（即同一账号在某段时间做多次的存取钱操作），它们的相对顺序意味着添加到容器的时间顺序，此顺序不能修改，否则很可能出现账户透支的情况。**

**值得一提的是，stable\_sort() 函数完全可以看作是 sort() 函数在功能方面的升级版。换句话说，stable\_sort() 和 sort() 具有相同的使用场景，就连语法格式也是相同的（后续会讲），只不过前者在功能上除了可以实现排序，还可以保证不改变相等元素的相对位置。**

注意，关于 stable\_sort() 函数的使用场景，《[C++ sort() 排序函数](http://c.biancheng.net/view/7457.html" \t "http://c.biancheng.net/view/_blank)》一节已经做了详细的介绍，这里不再赘述。另外，stable\_sort() 函数是基于归并排序实现的，关于此排序算法的具体实现过程，感兴趣的读者可阅读《[归并排序算法](http://c.biancheng.net/view/vip_3449.html" \t "http://c.biancheng.net/view/_blank)》一文。

和 sort() 函数一样，实现 stable\_sort() 的函数模板也位于<algorithm>头文件中，因此在使用该函数前，程序也应包含如下语句：

#include <algorithm>

并且，table\_sort() 函数的用法也有 2 种，其语法格式和 sort() 函数完全相同（仅函数名不同）：

**//对 [first, last) 区域内的元素做默认的升序排序**

**void stable\_sort ( RandomAccessIterator first, RandomAccessIterator last );**

**//按照指定的 comp 排序规则，对 [first, last) 区域内的元素进行排序**

**void stable\_sort ( RandomAccessIterator first, RandomAccessIterator last, Compare comp );**

**其中，first 和 last 都为随机访问迭代器，它们的组合 [first, last) 用来指定要排序的目标区域；另外在第 2 种格式中，comp 可以是 C++ STL 标准库提供的排序规则（比如 std::greater<T>），也可以是自定义的排序规则。**  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::stable\_sort

#include <vector> // std::vector

//以普通函数的方式实现自定义排序规则

bool mycomp(int i, int j) {

return (i < j);

}

//以函数对象的方式实现自定义排序规则

class mycomp2 {

public:

bool operator() (int i, int j) {

return (i < j);

}

};

int *main*() {

*std*::*vector*<int> myvector{ 32, 71, 12, 45, 26, 80, 53, 33 };

//调用第一种语法格式，对 32、71、12、45 进行排序

*std*::*stable\_sort*(myvector.*begin*(), myvector.*begin*() + 4); //(12 32 45 71) 26 80 53 33

//调用第二种语法格式，利用STL标准库提供的其它比较规则（比如 greater<T>）进行排序

*std*::*stable\_sort*(myvector.*begin*(), myvector.*begin*() + 4, *std*::*greater*<int>()); //(71 45 32 12) 26 80 53 33

//调用第二种语法格式，通过自定义比较规则进行排序,这里也可以换成 mycomp2()

*std*::*stable\_sort*(myvector.*begin*(), myvector.*end*(), mycomp);//12 26 32 33 45 53 71 80

//输出 myvector 容器中的元素

for (*std*::*vector*<int>::*iterator* it = myvector.*begin*(); it != myvector.*end*(); ++it) {

*std*::*cout* << \*it << ' ';

}

return 0;

}

**程序执行结果为：**

**12 26 32 33 45 53 71 80**

那么，stable\_sort() 函数的效率怎么样呢？当可用空间足够的情况下，该函数的时间复杂度可达到O(N\*log2(N))；反之，时间复杂度为O(N\*log2(N)2)，其中 N 为指定区域 [first, last) 中 last 和 first 的距离。

**第三 C++ partial\_sort()函数详解**

假设这样一种情境，有一个存有 100 万个元素的容器，但我们只想从中提取出值最小的 10 个元素，该如何实现呢？  
 通过前面的学习，读者可能会想到使用 sort() 或者 stable\_sort() 排序函数，即通过对容器中存储的 100 万个元素进行排序，就可以成功筛选出最小的 10 个元素。但仅仅为了提取 10 个元素，却要先对 100 万个元素进行排序，可想而知这种实现方式的效率是非常低的。  
 对于解决类似的问题，C++ STL 标准库提供了更高效的解决方案，即使用 partial\_sort() 或者 partial\_sort\_copy() 函数，本节就对这 2 个排序函数的功能和用法做详细的讲解。  
 首先需要说明的是，partial\_sort() 和 partial\_sort\_copy() 函数都位于 <algorithm> 头文件中，因此在使用这 2 个函数之前，程序中应引入此头文件：

#include <algorithm>

**一 C++ partial\_sort()排序函数**

要知道，一个函数的功能往往可以从它的函数名中体现出来，以 partial\_sort() 函数为例，partial sort 可直译为“部分排序”。partial\_sort() 函数的功能确是如此，即该函数可以从指定区域中提取出部分数据，并对它们进行排序。  
 但“部分排序”仅仅是对 partial\_sort() 函数功能的一个概括，如果想彻底搞清楚它的功能，需要结合该函数的语法格式。partial\_sort() 函数有 2 种用法，其语法格式分别为：

**//按照默认的升序排序规则，对 [first, last) 范围的数据进行筛选并排序**

**void partial\_sort (RandomAccessIterator first,**

**RandomAccessIterator middle,**

**RandomAccessIterator last);**

**//按照 comp 排序规则，对 [first, last) 范围的数据进行筛选并排序**

**void partial\_sort (RandomAccessIterator first,**

**RandomAccessIterator middle,**

**RandomAccessIterator last,**

**Compare comp);**

其中，first、middle 和 last 都是随机访问迭代器，comp 参数用于自定义排序规则。  
partial\_sort() 函数会以交换元素存储位置的方式实现部分排序的。具体来说，partial\_sort() 会将 [first, last) 范围内最小（或最大）的 middle-first 个元素移动到 [first, middle) 区域中，并对这部分元素做升序（或降序）排序。  
 需要注意的是，partial\_sort() 函数受到底层实现方式的限制，它仅适用于普通数组和部分类型的容器。换句话说，只有普通数组和具备以下条件的容器，才能使用 partial\_sort() 函数：

（1）容器支持的迭代器类型必须为随机访问迭代器。这意味着，partial\_sort() 函数只适用于 array、vector、deque 这 3 个容器。

（2）当选用默认的升序排序规则时，容器中存储的元素类型必须支持 <小于运算符；同样，如果选用标准库提供的其它排序规则，元素类型也必须支持该规则底层实现所用的比较运算符；

（3）partial\_sort() 函数在实现过程中，需要交换某些元素的存储位置。因此，如果容器中存储的是自定义的类对象，则该类的内部必须提供移动构造函数和移动赋值运算符。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::partial\_sort

#include <vector> // std::vector

using namespace *std*;

//以普通函数的方式自定义排序规则

bool mycomp1(int i, int j) {

return (i > j);

}

//以函数对象的方式自定义排序规则

class mycomp2 {

public:

bool operator() (int i, int j) {

return (i > j);

}

};

int *main*() {

*std*::*vector*<int> myvector{ 3,2,5,4,1,6,9,7 };

//以默认的升序排序作为排序规则，将 myvector 中最小的 4 个元素移动到开头位置并排好序

*std*::*partial\_sort*(myvector.*begin*(), myvector.*begin*() + 4, myvector.*end*());

*cout* << "第一次排序:\n";

for (*std*::*vector*<int>::*iterator* it = myvector.*begin*(); it != myvector.*end*(); ++it)

*std*::*cout* << \*it << ' ';

*cout* << "\n第二次排序:\n";

// 以指定的 mycomp2 作为排序规则，将 myvector 中最大的 4 个元素移动到开头位置并排好序

*std*::*partial\_sort*(myvector.*begin*(), myvector.*begin*() + 4, myvector.*end*(), mycomp2());

for (*std*::*vector*<int>::*iterator* it = myvector.*begin*(); it != myvector.*end*(); ++it)

*std*::*cout* << \*it << ' ';

return 0;

}

**程序执行结果为：**

**第一次排序:  
1 2 3 4 5 6 9 7  
第二次排序:  
9 7 6 5 1 2 3 4**

值得一提的是，partial\_sort() 函数实现排序的平均时间复杂度为N\*log(M)，其中 N 指的是 [first, last) 范围的长度，M 指的是 [first, middle) 范围的长度。

**二 C++ partial\_sort\_copy()排序函数**

partial\_sort\_copy() 函数的功能和 partial\_sort() 类似，唯一的区别在于，前者不会对原有数据做任何变动，而是先将选定的部分元素拷贝到另外指定的数组或容器中，然后再对这部分元素进行排序。  
 partial\_sort\_copy() 函数也有 2 种语法格式，分别为：

**//默认以升序规则进行部分排序**

**RandomAccessIterator partial\_sort\_copy (**

**InputIterator first,**

**InputIterator last,**

**RandomAccessIterator result\_first,**

**RandomAccessIterator result\_last);**

**//以 comp 规则进行部分排序**

**RandomAccessIterator partial\_sort\_copy (**

**InputIterator first,**

**InputIterator last,**

**RandomAccessIterator result\_first,**

**RandomAccessIterator result\_last,**

**Compare comp);**

其中，first 和 last 为输入迭代器；result\_first 和 result\_last 为随机访问迭代器；comp 用于自定义排序规则。  
 partial\_sort\_copy() 函数会将 [first, last) 范围内最小（或最大）的 result\_last-result\_first 个元素复制到 [result\_first, result\_last) 区域中，并对该区域的元素做升序（或降序）排序。  
 值得一提的是，[first, last] 中的这 2 个迭代器类型仅限定为输入迭代器，这意味着相比 partial\_sort() 函数，partial\_sort\_copy() 函数放宽了对存储原有数据的容器类型的限制。换句话说，partial\_sort\_copy() 函数还支持对 list 容器或者 forward\_list 容器中存储的元素进行“部分排序”，而 partial\_sort() 函数不行。  
 但是，介于 result\_first 和 result\_last 仍为随机访问迭代器，因此 [result\_first, result\_last) 指定的区域仍仅限于普通数组和部分类型的容器，这和 partial\_sort() 函数对容器的要求是一样的。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::partial\_sort\_copy

#include <list> // std::list

using namespace *std*;

bool mycomp1(int i, int j) {

return (i > j);

}

class mycomp2 {

public:

bool operator() (int i, int j) {

return (i > j);

}

};

int *main*() {

int myints[5] = { 0 };

*std*::*list*<int> mylist{ 3,2,5,4,1,6,9,7 };

//按照默认的排序规则进行部分排序

*std*::*partial\_sort\_copy*(mylist.*begin*(), mylist.*end*(), myints, myints + 5);

*cout* << "第一次排序：\n";

for (int i = 0; i < 5; i++) {

*cout* << myints[i] << " ";

}

//以自定义的 mycomp2 作为排序规则，进行部分排序

*std*::*partial\_sort\_copy*(mylist.*begin*(), mylist.*end*(), myints, myints + 5, mycomp2());

*cout* << "\n第二次排序：\n";

for (int i = 0; i < 5; i++) {

*cout* << myints[i] << " ";

}

return 0;

}

**程序执行结果为：**

**第一次排序：  
1 2 3 4 5  
第二次排序：  
9 7 6 5 4**

可以看到，程序中调用了 2 次 partial\_sort\_copy() 函数，其作用分别是：

（1）第 20 行：采用默认的升序排序规则，在 mylist 容器中筛选出最小的 5 个元素，然后将它们复制到 myints[5] 数组中，并对这部分元素进行升序排序；

（2）第 27 行：采用自定义的 mycomp2 降序排序规则，从 mylist 容器筛选出最大的 5 个元素，同样将它们复制到 myints[5] 数组中，并对这部分元素进行降序排序；  
 值得一提的是，partial\_sort\_copy() 函数实现排序的平均时间复杂度为N\*log(min(N,M))，其中 N 指的是 [first, last) 范围的长度，M 指的是 [result\_first, result\_last) 范围的长度。

**第四 C++ nth\_element()用法详解**

前面章节中，已经给大家介绍了 sort()、stable\_sort()、partial\_sort() 这些函数的功能和用法，本节再介绍一个排序函数，即 nth\_element() 函数。  
 不过，在系统讲解 nth\_element() 函数之前，我们先形成一个共识，即在有序序列中，我们可以称第 n 个元素为整个序列中“第 n 大”的元素。比如，下面是一个升序序列：

2 4 6 8 10

在这个序列中，我们可以称元素 6 为整个序列中“第 3 小”的元素，并位于第 3 的位置处；同样，元素 8 为整个序列中“第 4 小”的元素，并位于第 4 的位置处。  
 简单的理解 nth\_element() 函数的功能，当采用默认的升序排序规则（std::less<T>）时，该函数可以从某个序列中找到第 n 小的元素 K，并将 K 移动到序列中第 n 的位置处。不仅如此，整个序列经过 nth\_element() 函数处理后，所有位于 K 之前的元素都比 K 小，所有位于 K 之后的元素都比 K 大。  
 当然，我们也可以将 nth\_element() 函数的排序规则自定义为降序排序，此时该函数会找到第 n 大的元素 K 并将其移动到第 n 的位置处，同时所有位于 K 之前的元素都比 K 大，所有位于 K 之后的元素都比 K 小。  
以下面这个序列为例：

3 4 1 2 5

假设按照升序排序，并通过 nth\_element() 函数查找此序列中第 3 小的元素，则最终得到的序列可能为：

2 1 3 4 5

显然，nth\_element() 函数找到了第 3 小的元素 3 并将其位于第 3 的位置，同时元素 3 之前的所有元素都比该元素小，元素 3 之后的所有元素都比该元素大。  
 要知道，nth\_element() 本质也是一个函数模板，定义在<algorithm>头文件中。因此，如果程序中想使用该函数，就需要提前引入这个头文件：

#include <algorithm>  
**nth\_element() 函数有以下 2 种语法格式：**

**//排序规则采用默认的升序排序**

**void nth\_element (RandomAccessIterator first,**

**RandomAccessIterator nth,**

**RandomAccessIterator last);**

**//排序规则为自定义的 comp 排序规则**

**void nth\_element (RandomAccessIterator first,**

**RandomAccessIterator nth,**

**RandomAccessIterator last,**

**Compare comp);**

**其中，各个参数的含义如下：**

**（1）first 和 last：都是随机访问迭代器，[first, last) 用于指定该函数的作用范围（即要处理哪些数据）；**

**（2）nth：也是随机访问迭代器，其功能是令函数查找“第 nth 大”的元素，并将其移动到 nth 指向的位置；**

**（3）comp：用于自定义排序规则。**  
注意，鉴于 nth\_element() 函数中各个参数的类型，其只能对普通数组或者部分容器进行排序。换句话说，只有普通数组和符合以下全部条件的容器，才能使用使用 nth\_element() 函数：

1、容器支持的迭代器类型必须为随机访问迭代器。这意味着，nth\_element() 函数只适用于 array、vector、deque 这 3 个容器。

2、当选用默认的升序排序规则时，容器中存储的元素类型必须支持 <小于运算符；同样，如果选用标准库提供的其它排序规则，元素类型也必须支持该规则底层实现所用的比较运算符；

3、nth\_element() 函数在实现过程中，需要交换某些元素的存储位置。因此，如果容器中存储的是自定义的类对象，则该类的内部必须提供移动构造函数和移动赋值运算符。

举个例子：

#include <iostream>

#include <algorithm> // std::nth\_element

#include <vector> // std::vector

using namespace *std*;

//以普通函数的方式自定义排序规则

bool mycomp1(int i, int j) {

return (i > j);

}

//以函数对象的方式自定义排序规则

class mycomp2 {

public:

bool operator() (int i, int j) {

return (i > j);

}

};

int *main*() {

*std*::*vector*<int> myvector{ 3,1,2,5,4 };

//默认的升序排序作为排序规则

*std*::*nth\_element*(myvector.*begin*(), myvector.*begin*() + 2, myvector.*end*());

*cout* << "第一次nth\_element排序：\n";

for (*std*::*vector*<int>::*iterator* it = myvector.*begin*(); it != myvector.*end*(); ++it) {

*std*::*cout* << \*it << ' ';

}

//自定义的 mycomp2() 或者 mycomp1 降序排序作为排序规则

*std*::*nth\_element*(myvector.*begin*(), myvector.*begin*() + 3, myvector.*end*(), mycomp1);

*cout* << "\n第二次nth\_element排序：\n";

for (*std*::*vector*<int>::*iterator* it = myvector.*begin*(); it != myvector.*end*(); ++it) {

*std*::*cout* << \*it << ' ';

}

return 0;

}

**程序执行结果可能为（不唯一）：**

**第一次nth\_element排序：  
1 2 3 4 5  
第二次nth\_element排序：  
5 4 3 2 1**

上面程序中，共调用了 2 次 nth\_elelment() 函数：

（1）第 20 行：nth\_element() 函数采用的是默认的升序排序，nth 参数设置为 myvector.begin()+2，即指向的是 myvector 容器中第 3 个元素所在的位置。因此，nth\_element() 函数会查找“第 3 小”的元素 3，并将其移动到 nth 指向的位置，同时使 nth 之前的所有元素都比 3 小，使 nth 之后的所有元素都比 3 大。

（2）第 26 行：nth\_element() 函数采用的是默认的降序排序，nth 参数设置为 myvector.begin()+3，即指向的是 myvector 容器中第 4 个元素所在的位置。因此，nth\_element() 函数会查找“第 4 大”的元素 2，并将其移动到 nth 指向的位置，同时使 nth 之前的所有元素都比 2 大，使 nth 之后的所有元素都比 2 小。

**第五 C++ is\_sorted()函数完全攻略**

我们知道，排序操作是比较耗费时间的，尤其当数据量很大的时候。因此在设计程序时，我们应该有意识的去避免执行一些不必要的排序操作。  
那么，何谓不必要的排序操作呢？举个例子，有这样一组数据：

1 2 3 4 5

这本就是一组有序的数据，如果我们恰巧需要这样的升序序列，就没有必要再执行排序操作。  
因此，当程序中涉及排序操作时，我们应该为其包裹一层判断语句，像如下这样：

//...

if(不是有序序列){

//执行排序算法

}

//...

注意这里的“不是有序序列”，即只要该序列不符合我们指定的排序规则，就不是有序序列。

那么，怎样才能判断一个序列是否为有序序列呢？很简单，使用 is\_sorted() 函数即可，此函数专门用于判断某个序列是否为有序序列。

**一 C++ is\_sorted()函数**

和之前学习的其它排序函数（比如 sorted() 函数）一样，is\_sorted() 函数本质上就是一个函数模板，定义在<algorithm>头文件中。因为，在使用该函数之前，程序中必须先引入此头文件：

#include <algorithm>  
**is\_sorted() 函数有 2 种语法格式，分别是：**

**//判断 [first, last) 区域内的数据是否符合 std::less<T> 排序规则，即是否为升序序列**

**bool is\_sorted (ForwardIterator first, ForwardIterator last);**

**//判断 [first, last) 区域内的数据是否符合 comp 排序规则**

**bool is\_sorted (ForwardIterator first, ForwardIterator last, Compare comp);**

其中，first 和 last 都为正向迭代器（这意味着该函数适用于大部分容器），[first, last) 用于指定要检测的序列；comp 用于指定自定义的排序规则。

**注意，如果使用默认的升序排序规则，则 [first, last) 指定区域内的元素必须支持使用 < 小于运算符做比较；同样，如果指定排序规则为 comp，也要保证 [first, last) 区域内的元素支持该规则内部使用的比较运算符。**

另外，该函数会返回一个 bool 类型值，即如果 [first, last) 范围内的序列符合我们指定的排序规则，则返回 true；反之，函数返回 false。值得一提得是，如果 [first, last) 指定范围内只有 1 个元素，则该函数始终返回 true。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::is\_sorted

#include <vector> // std::array

#include <list> // std::list

using namespace *std*;

//以普通函数的方式自定义排序规则

bool mycomp1(int i, int j) {

return (i > j);

}

//以函数对象的方式自定义排序规则

class mycomp2 {

public:

bool operator() (int i, int j) {

return (i > j);

}

};

int *main*() {

*vector*<int> myvector{ 3,1,2,4 };

*list*<int> mylist{ 1,2,3,4 };

//调用第 2 种语法格式的 is\_sorted() 函数，该判断语句会得到执行

if (!*is\_sorted*(myvector.*begin*(), myvector.*end*(), mycomp2())) {

*cout* << "开始对 myvector 容器排序" << *endl*;

//对 myvector 容器做降序排序

*sort*(myvector.*begin*(), myvector.*end*(), mycomp2());

//输出 myvector 容器中的元素

for (auto it = myvector.*begin*(); it != myvector.*end*(); ++it) {

*cout* << \*it << " ";

}

}

//调用第一种语法格式的 is\_sorted() 函数，该判断语句得不到执行

if (!*is\_sorted*(mylist.*begin*(), mylist.*end*())) {

*cout* << "开始对 mylist 排序" << *endl*;

//......

}

return 0;

}

**程序执行结果为：**

**开始对 myvector 容器排序  
4 3 2 1**

结合输出结果可以看到，虽然 myvector 容器中的数据为降序序列，但我们需要的是升序序列。因此第 22 行代码中 is\_sorted() 函数的返回值为 false，而 !false 即 true，所以此 if 判断语句会得到执行。  
同样在 33 行代码中，mylist 容器中存储的数据为升序序列，和 is\_sorted() 函数的要求相符，因此该函数的返回值为 true，而 !true 即 false，所以此 if 判断语句将无法得到执行。

**C++标准库官方网站给出了 is\_sorted() 函数底层实现的等效代码，感兴趣的读者可自行前往查看。**

**二 C++ is\_sorted\_until()函数**

和 is\_sorted() 函数相比，is\_sorted\_until() 函数不仅能检测出某个序列是否有序，还会返回一个正向迭代器，该迭代器指向的是当前序列中第一个破坏有序状态的元素。  
 is\_sorted\_until() 函数的定义也位于<algorithm>头文件中。因为，在使用该函数之前，程序中必须先引入此头文件：

#include <algorithm>

**is\_sorted\_until() 函数有以下 2 种语法格式：**

**//排序规则为默认的升序排序**

**ForwardIterator is\_sorted\_until (ForwardIterator first, ForwardIterator last);**

**//排序规则是自定义的 comp 规则**

**ForwardIterator is\_sorted\_until (ForwardIterator first,**

**ForwardIterator last,**

**Compare comp);**

其中，first 和 last 都为正向迭代器（这意味着该函数适用于大部分容器），[first, last) 用于指定要检测的序列；comp 用于指定自定义的排序规则。

**注意，如果使用默认的升序排序规则，则 [first, last) 指定区域内的元素必须支持使用 < 小于运算符做比较；同样，如果指定排序规则为 comp，也要保证 [first, last) 区域内的元素支持该规则内部使用的比较运算符。**

可以看到，该函数会返回一个正向迭代器。对于第一种语法格式来说，该函数返回的是指向序列中第一个破坏升序规则的元素；对于第二种语法格式来说，该函数返回的是指向序列中第一个破坏 comp 排序规则的元素。注意，如果 [first, last) 指定的序列完全满足默认排序规则或者 comp 排序规则的要求，则该函数将返回一个和 last 迭代器指向相同的正向迭代器。  
  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::is\_sorted\_until

#include <vector> // std::array

#include <list> // std::list

using namespace *std*;

//以普通函数的方式自定义排序规则

bool mycomp1(int i, int j) {

return (i > j);

}

//以函数对象的方式自定义排序规则

class mycomp2 {

public:

bool operator() (int i, int j) {

return (i > j);

}

};

int *main*() {

*vector*<int> myvector{ 3,1,2,4 };

*list*<int> mylist{ 1,2,3,4 };

//如果返回值为 myvector.end()，则表明 myvector 容器中的序列符合 mycomp2() 规则

if (*is\_sorted\_until*(myvector.*begin*(), myvector.*end*(), mycomp2()) != myvector.*end*()) {

*cout* << "开始对 myvector 容器排序" << *endl*;

//对 myvector 容器做降序排序

*sort*(myvector.*begin*(), myvector.*end*(), mycomp2());

//输出 myvector 容器中的元素

for (auto it = myvector.*begin*(); it != myvector.*end*(); ++it) {

*cout* << \*it << " ";

}

}

//该判断语句得不到执行

if (*is\_sorted\_until*(mylist.*begin*(), mylist.*end*()) != mylist.*end*()) {

*cout* << "开始对 mylist 排序" << *endl*;

//......

}

return 0;

}

**程序执行结果为：**

**开始对 myvector 容器排序  
4 3 2 1**

**第六 C++ STL标准库这么多排序函数，该如何选择？**

通过前面的学习我们知道，C++ STL 标准库共提供了 4 种排序函数，这里先带大家回顾一下，如表 1 所示。

表 1 C++ STL排序函数

|  |  |
| --- | --- |
| 排序函数 | 功能 |
| sort() | 对指定范围内所有的数据进行排序，排序后各个元素的相对位置很可能发生改变。 |
| stable\_sort() | 对指定范围内所有的数据进行排序，并确保排序后各个元素的相对位置不发生改变。 |
| partial\_sort() | 对指定范围内最大或最小的 n 个元素进行排序。 |
| nth\_element() | 调整指定范围内元素的存储位置，实现位于位置 n 的元素正好是全排序情况下的第 n 个元素，并且按照全排序规则排在位置 n 之前的元素都在该位置之前，按照全排序规则排在位置 n 之后的元素都在该位置之后。 |

关于以上 4 种排序函数各自的用法，读者可阅读之前的文章，这里不再过多赘述。

值得一提的是，以上 4 种排序函数在使用时，都要求传入随机访问迭代器，因此这些函数都只适用于 array、vector、deque 以及普通数组。

**当操作对象为 list 或者 forward\_list 序列式容器时，其容器模板类中都提供有 sort() 排序方法，借助此方法即可实现对容器内部元素进行排序。其次，对关联式容器（包括哈希容器）进行排序是没有实际意义的，因为这类容器会根据既定的比较函数（和哈希函数）维护内部元素的存储位置。**

那么，当需要对普通数组或者 array、vector 或者 deque 容器中的元素进行排序时，怎样选择最合适（效率最高）的排序函数呢？这里为大家总结了以下几点：

（1）如果需要对所有元素进行排序，则选择 sort() 或者 stable\_sort() 函数；

（2）如果需要保持排序后各元素的相对位置不发生改变，就只能选择 stable\_sort() 函数，而另外 3 个排序函数都无法保证这一点；

（3）如果需要对最大（或最小）的 n 个元素进行排序，则优先选择 partial\_sort() 函数；

（4）如果只需要找到最大或最小的 n 个元素，但不要求对这 n 个元素进行排序，则优先选择 nth\_element() 函数。  
 除此之外，很多读者都关心这些排序函数的性能。总的来说，函数功能越复杂，做的工作越多，它的性能就越低（主要体现在时间复杂度上）。对于以上 4 种排序函数，综合考虑它们的时间和空间效率，其性能之间的比较如下所示：

**nth\_element() > partial\_sort() > sort() > stable\_sort()       <--从左到右，性能由高到低**

建议大家，在实际选择排序函数时，应更多从所需要完成的功能这一角度去考虑，而不是一味地追求函数的性能。换句话说，如果你选择的算法更有利于实现所需要的功能，不仅会使整个代码的逻辑更加清晰，还会达到事半功倍的效果。

**第七 自定义STL算法规则，应优先使用函数对象！**

作为一门面向对象的编程语言，使用 C++ 编写程序有一个缺点，即随着代码面向对象程度的提高，其执行效率反而会降低。例如，经实验证明几乎在所有情况下，直接操作一个 double 类型变量的执行效率，要比操作一个含 double 类型成员属性的类对象更高。  
 对于大多数读者来说，以上所说是很容易想通的，因为它符合我们对高级编程语言的认知。但本节要介绍的内容，一定程序上会打破这个认知。  
 前面章节中，我们学习了 STL 标准库中所有的排序算法，比如 sort()、stable\_sort() 以及 nth\_element() 等。不知读者有没有发现，这些排序算法都单独提供了带有 comp 参数的语法格式，借助此参数，我们可以自定义排序规则。  
  
**以 sort() 排序函数为例，其语法格式有以下 2 种：**

**//无 comp 参数**

**void sort (RandomAccessIterator first, RandomAccessIterator last);**

**//有 comp 参数**

**void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);**

显然仅从使用语法上看，它们唯一的区别在于，第 2 种多了一个 comp 参数。

事实上，对于 STL 标准库中的每个算法，只要用户需要自定义规则，该算法都会提供有带 comp 参数的语法格式。

本质上讲，comp 参数用于接收用户自定义的函数，其定义的方式有 2 种，既可以是普通函数，也可以是函数对象。例如：

#include <iostream> // std::cout

#include <algorithm> // std::sort

#include <vector> // std::vector

//以普通函数的方式实现自定义排序规则

inline bool mycomp(int i, int j) {

return (i < j);

}

//以函数对象的方式实现自定义排序规则

class mycomp2 {

public:

bool operator() (int i, int j) {

return (i < j);

}

};

int *main*() {

*std*::*vector*<int> myvector{ 32, 71, 12, 45, 26, 80, 53, 33 };

//调用普通函数定义的排序规则

*std*::*sort*(myvector.*begin*(), myvector.*end*(), mycomp);

//调用函数对象定义的排序规则

//std::sort(myvector.begin(), myvector.end(), mycomp2());

for (*std*::*vector*<int>::*iterator* it = myvector.*begin*(); it != myvector.*end*(); ++it) {

*std*::*cout* << \*it << ' ';

}

return 0;

}

**程序执行结果为：**

**12 26 32 33 45 53 71 80**

注意，为了提高执行效率，其函数都定义为内联函数（在类内部定义的函数本身就是内联函数）。至于为什么内联函数比普通函数的执行效率高，可阅读《[C++ inline内联函数](http://c.biancheng.net/view/2201.html" \t "http://c.biancheng.net/view/_blank)》一文。

要知道，函数对象可以理解为伪装成函数的对象，根据以往的认知，函数对象的执行效率应该不如普通函数。但事实恰恰相反，即便如上面程序那样，将普通函数定义为更高效的内联函数，其执行效率也无法和函数对象相比。

通过在 4 个不同的 STL 平台上，对包含 100 万个 double 类型数据的 vector 容器进行排序，最差情况下使用函数对象的执行效率要比普通内联函数高 50%，最好情况下则高 160%。

那么，是什么原因导致了它们执行效率上的差异呢？以 mycomp2() 函数对象为例，其 mycomp2::operator() 也是一个内联函数，编译器在对 sort() 函数进行实例化时会将该函数直接展开，这也就意味着，展开后的 sort() 函数内部不包含任何函数调用。  
 而如果使用 mycomp 作为参数来调用 sort() 函数，情形则大不相同。要知道，C++ 并不能真正地将一个函数作为参数传递给另一个函数，换句话说，如果我们试图将一个函数作为参数进行传递，编译器会隐式地将它转换成一个指向该函数的指针，并将该指针传递过去。  
  
也就是说，上面程序中的如下代码：

**std::sort(myvector.begin(), myvector.end(), mycomp);**

**并不是真正地将 mycomp 传递给 sort() 函数，它传递的仅是一个指向 mycomp() 函数的指针。当 sort() 函数被实例化时，编译器生成的函数声明如下所示：**

**std::sort(vector<int>::iterator first,**

**vector<int>::iterator last,**

**bool (\*comp)(int, int));**

可以看到，参数 comp 只是一个指向函数的指针，所以 sort() 函数内部每次调用 comp 时，编译器都会通过指针产生一个间接的函数调用。

**也正是基于这个原因，C++ sort() 函数要比 C 语言 qsort() 函数的执行效率更高。读者可能会问，程序中 comp() 函数也是内联函数，为什么 C++ 不像函数对象那样去处理呢？具体原因我们无从得知，事实上也没必要关心，也许是编译器开发者觉得这种优化不值得去做。**

除了效率上的优势之外，相比普通函数，以函数对象的方式自定义规则还有很多隐藏的优势。例如在某些特殊情况下，以普通函数的形式编写的代码看似非常合理，但就是无法通过编译，这也许是由于 STL 标准库的原因，也许是编译器缺陷所至，甚至两者都有可能。而使用函数对象的方式，则可以有效避开这些“坑”，而且还大大提升的代码的执行效率。  
 总之，以函数对象的方式为 STL 算法自定义规则，具有效率在内的诸多优势。当调用带有 comp 参数的 STL 算法时，除非调用 STL 标准库自带的比较函数，否则应优先以函数对象的方式自定义规则。

**第八 C++ merge()和inplace\_merge()函数用法（详解版）**

有些场景中，我们需要将 2 个有序序列合并为 1 个有序序列，这时就可以借助 merge() 或者 inplace\_merge() 函数实现。  
 值得一提的是，merge() 和 inplace\_merge() 函数都定义在<algorithm>头文件中，因此在使用它们之前，程序中必须提前引入该头文件：

#include <algorithm>

**一 C++ merge()函数**

merge() 函数用于将 2 个有序序列合并为 1 个有序序列，前提是这 2 个有序序列的排序规则相同（要么都是升序，要么都是降序）。并且最终借助该函数获得的新有序序列，其排序规则也和这 2 个有序序列相同。  
  **举个例子，假设有 2 个序列，分别为5,10,15,20,25和7,14,21,28,35,42，显然它们不仅有序，而且都是升序序列。因此借助 merge() 函数，我们就可以轻松获得如下这个有序序列：**

**5 7 10 15 17 20 25 27 37 47 57**

可以看到，该序列不仅包含以上 2 个序列中所有的元素，并且其本身也是一个升序序列。  
值得一提的是，C++ STL 标准库的开发人员考虑到用户可能需要自定义排序规则，因此为 **merge() 函数设计了以下 2 种语法格式：**

**//以默认的升序排序作为排序规则**

**OutputIterator merge (InputIterator1 first1, InputIterator1 last1,**

**InputIterator2 first2, InputIterator2 last2,**

**OutputIterator result);**

**//以自定义的 comp 规则作为排序规则**

**OutputIterator merge (InputIterator1 first1, InputIterator1 last1,**

**InputIterator2 first2, InputIterator2 last2,**

**OutputIterator result, Compare comp);**

可以看到，first1、last1、first2 以及 last2 都为输入迭代器，[first1, last1) 和 [first2, last2) 各用来指定一个有序序列；result 为输出迭代器，用于为最终生成的新有序序列指定存储位置；comp 用于自定义排序规则。同时，该函数会返回一个输出迭代器，其指向的是新有序序列中最后一个元素之后的位置。

**注意，当采用第一种语法格式时，[first1, last1) 和 [first2, last2) 指定区域内的元素必须支持 < 小于运算符；同样当采用第二种语法格式时，[first1, last1) 和 [first2, last2) 指定区域内的元素必须支持 comp 排序规则内的比较运算符。**

举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::merge

#include <vector> // std::vector

using namespace *std*;

int *main*() {

//first 和 second 数组中各存有 1 个有序序列

int first[] = { 5,10,15,20,25 };

int second[] = { 7,17,27,37,47,57 };

//用于存储新的有序序列

*vector*<int> myvector(11);

//将 [first,first+5) 和 [second,second+6) 合并为 1 个有序序列，并存储到 myvector 容器中。

*merge*(first, first + 5, second, second + 6, myvector.*begin*());

//输出 myvector 容器中存储的元素

for (*vector*<int>::*iterator* it = myvector.*begin*(); it != myvector.*end*(); ++it) {

*cout* << \*it << ' ';

}

return 0;

}

**程序执行结果为：**

**5 7 10 15 17 20 25 27 37 47 57**

可以看到，first 数组和 second 数组中各存有 1 个升序序列，通过借助 merge() 函数，我们成功地将它们合并成了一个有序序列，并存储到 myvector 容器中。

**注意，merge() 函数底层是通过拷贝的方式实现合并操作的。换句话说，上面程序在采用 merge() 函数实现合并操作的同时，并不会对 first 和 second 数组有任何影响。有关该函数的具体实现过程，可查看 [C++ STL merge() 官网](http://www.cplusplus.com/reference/algorithm/merge/" \t "http://c.biancheng.net/view/_blank)。**

实际上，对于 2 个有序序列是各自存储（像 first 和 second 这样）还是存储到一起，merge() 函数并不关心，只需要给它传入恰当的迭代器（或指针），该函数就可以正常工作。因此，我们还可以将上面程序改写为：

**//该数组中存储有 2 个有序序列**

**int first[] = { 5,10,15,20,25,7,17,27,37,47,57 };**

**//用于存储新的有序序列**

**vector<int> myvector(11);**

**//将 [first,first+5) 和 [first+5,first+11) 合并为 1 个有序序列，并存储到 myvector 容器中。**

**merge(first, first + 5, first + 5, first +11 , myvector.begin());**

可以看到，2 个有序序列全部存储到了 first 数组中，但只要给 merge() 函数传入正确的指针，仍可以将它们合并为 1 个有序序列。

**感兴趣的读者，可自行验证这段程序，其最终会得到和上面程序相同的 myvector 容器。**

二 C++ inplace\_merge()函数

事实上，当 2 个有序序列存储在同一个数组或容器中时，如果想将它们合并为 1 个有序序列，除了使用 merge() 函数，更推荐使用 inplace\_merge() 函数。  
和 merge() 函数相比，inplace\_merge() 函数的语法格式要简单很多：

**//默认采用升序的排序规则**

**void inplace\_merge (BidirectionalIterator first, BidirectionalIterator middle,**

**BidirectionalIterator last);**

**//采用自定义的 comp 排序规则**

**void inplace\_merge (BidirectionalIterator first, BidirectionalIterator middle,**

**BidirectionalIterator last, Compare comp);**

其中，first、middle 和 last 都为双向迭代器，[first, middle) 和 [middle, last) 各表示一个有序序列。  
 和 merge() 函数一样，inplace\_merge() 函数也要求 [first, middle) 和 [middle, last) 指定的这 2 个序列必须遵循相同的排序规则，且当采用第一种语法格式时，这 2 个序列中的元素必须支持 < 小于运算符；同样，当采用第二种语法格式时，这 2 个序列中的元素必须支持 comp 排序规则内部的比较运算符。不同之处在于，merge() 函数会将最终合并的有序序列存储在其它数组或容器中，而 inplace\_merge() 函数则将最终合并的有序序列存储在 [first, last) 区域中。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::merge

using namespace *std*;

int *main*() {

//该数组中存储有 2 个有序序列

int first[] = { 5,10,15,20,25,7,17,27,37,47,57 };

//将 [first,first+5) 和 [first+5,first+11) 合并为 1 个有序序列。

*inplace\_merge*(first, first + 5, first + 11);

for (int i = 0; i < 11; i++) {

*cout* << first[i] << " ";

}

return 0;

}

**程序执行结果为：**

**5 7 10 15 17 20 25 27 37 47 57**

可以看到，first 数组中包含 2 个升序序列，借助 inplace\_merge() 函数，实现了将这 2 个序列合并为 1 个升序序列，且新序列仍存储在 first 数组中。

**第九 C++ find()函数用法详解**

find() 函数本质上是一个模板函数，用于在指定范围内查找和目标元素值相等的第一个元素。  
**如下为 find() 函数的语法格式：**

**InputIterator find (InputIterator first, InputIterator last, const T& val);**

**其中，first 和 last 为输入迭代器，[first, last) 用于指定该函数的查找范围；val 为要查找的目标元素。**

正因为 first 和 last 的类型为输入迭代器，因此该函数适用于所有的序列式容器。

另外，该函数会返回一个输入迭代器，当 find() 函数查找成功时，其指向的是在 [first, last) 区域内查找到的第一个目标元素；如果查找失败，则该迭代器的指向和 last 相同。  
 值得一提的是，find() 函数的底层实现，其实就是用==运算符将 val 和 [first, last) 区域内的元素逐个进行比对。这也就意味着，[first, last) 区域内的元素必须支持==运算符。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::find

#include <vector> // std::vector

using namespace *std*;

int *main*() {

//find() 函数作用于普通数组

char stl[] = "http://c.biancheng.net/stl/";

//调用 find() 查找第一个字符 'c'

char\* p = *find*(stl, stl + *strlen*(stl), 'c');

//判断是否查找成功

if (p != stl + *strlen*(stl)) {

*cout* << p << *endl*;

}

//find() 函数作用于容器

*std*::*vector*<int> myvector{ 10,20,30,40,50 };

*std*::*vector*<int>::*iterator* it;

it = *find*(myvector.*begin*(), myvector.*end*(), 30);

if (it != myvector.*end*())

*cout* << "查找成功：" << \*it;

else

*cout* << "查找失败";

return 0;

}

**程序执行结果为：**

**c.biancheng.net/stl/  
查找成功：30**

可以看到，find() 函数除了可以作用于序列式容器，还可以作用于普通数组。  
对于 find() 函数的底层实现，C++ 标准库中给出了参数代码，感兴趣的读者可自行研究：

**template<class InputIterator, class T>**

**InputIterator find (InputIterator first, InputIterator last, const T& val)**

**{**

**while (first!=last) {**

**if (\*first==val) return first;**

**++first;**

**}**

**return last;**

**}**

**第十 能用STL算法，绝不自己实现！**

前面章节已经介绍了很多算法函数，比如 find()、merge()、sort() 等。不知读者有没有发现，每个算法函数都至少要用一对迭代器来指明作用区间，并且为了实现自己的功能，每个函数内部都势必会对指定区域内的数据进行遍历操作。  
 举几个例子，find() 函数会对指定区域的数据逐个进行遍历，确认其是否为要查找的目标元素；merge() 函数内部也会分别对 2 个有序序列做逐个遍历，从而将它们合并为一个有序序列；sort() 函数在对指定区域内的元素进行排序时，其底层也会遍历每个元素。  
 事实上，虽然这些算法函数的内部实现我们不得而知，但无疑都会用到循环结构。可以这么说，STL 标准库中几乎所有的算法函数，其底层都是借助循环结构实现的。  
 在此基础上，由于 STL 标准库使用场景很广，因此很多需要手动编写循环结构实现的功能，用 STL 算法函数就能完成。举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::for\_each

#include <string> // std::string

#include <vector> // std::vector

#include <functional>

using namespace *std*;

class Address {

public:

Address(*string* url) :url(url) {};

void display() {

*cout* << "url:" << this->url << *endl*;

}

private:

*string* url;

};

int *main*() {

*vector*<Address>adds{ Address("http://c.biancheng.net/stl/"),

Address("http://c.biancheng.net/java/"),

Address("http://c.biancheng.net/python/") };

//手动编写循环结构

*cout* << "first：\n";

for (auto it = adds.*begin*(); it != adds.*end*(); ++it) {

(\*it).display();

}

//调用 STL 标准库中的算法函数

*cout* << "second：\n";

*for\_each*(adds.*begin*(), adds.*end*(), *mem\_fun\_ref*(&Address::display));

return 0;

}

**程序执行结果为：**

**first：  
url:http://c.biancheng.net/stl/  
url:http://c.biancheng.net/java/  
url:http://c.biancheng.net/python/  
second：  
url:http://c.biancheng.net/stl/  
url:http://c.biancheng.net/java/  
url:http://c.biancheng.net/python/**

可以看到，对于输出 adds 容器中存储的元素，除了可以手动编写循环结构实现，还可以使用 STL 标准库提供的 for\_each() 函数。  
 那么，手动编写循环结构和调用 STL 算法函数相比，哪种实现方式更好呢？毫无疑问，直接调用算法会更好，理由有以下几个：

（1）算法函数通常比自己写的循环结构效率更高；

（2）自己写循环比使用算法函数更容易出错；

（3）相比自己编写循环结构，直接调用算法函数的代码更加简洁明了。

（4）使用算法函数编写的程序，可扩展性更强，更容易维护；

后面 3 个理由相信读者很容易理解，接下来重点讲一下“为什么算法函数的效率更高”。

**一 为什么STL算法效率更高**

仍以上面程序为例，如下是我们手动编写的循环代码：

for (auto it = adds.begin(); it != adds.end(); ++it) {

(\*it).display();

}

此段代码中，每一次循环都要执行一次 end() 方法，事实上该方法并不需要多次调用，因为它的值自始至终都没有发生改变。也就是说，end() 方法只需要调用一次就够啦，for\_each() 函数就对这一点进行了优化：

**for\_each(adds.begin(), adds.end(), mem\_fun\_ref(&Address::display));**

可以看到，通过将 end() 方法作为参数传入 for\_each() 函数，该方法只执行了 1 次。当然，这也仅是众多优化中的一处。事实上，STL 标准库的开发者对每个算法函数的底层实现代码都多了优化，使它们的执行效率达到最高。  
 **有读者可能会说，难道我们自己对循环结构进行优化不行吗？可以，但是其执行效率仍无法和算法函数相提并论。**  
 一方面，STL 开发者可以根据他们对容器底层的了解，对整个遍历过程进行优化，而这是我们难以做到的。以 deque 容器为例，该容器底层会将数据存储在多个大小固定的连续空间中。对于这些连续空间的遍历，只有 STL 开发者才知道这些连续空间的大小，才知道如何控制指针逐个遍历这些连续空间。  
 另一方面，某些 STL 函数的底层实现使用了复杂的科学计算方法，并不是普通 C++ 程序员能驾驭的。例如，在实现对某个序列进行排序时，我们很难编写出比 sort() 函数更高效的代码。

**总之，STL 开发者比使用者更了解内部的实现细节，他们会充分利用这些知识来对算法进行优化。**

当然，只有熟悉 STL 标准库提供的函数，才能在实际编程时想到使用它们。作为一个专业的 C++ 程序员，我们必须熟悉 STL 标准库中的每个算法函数，并清楚它们各自的功能。

**C++ STL 标准库中包含 70 多个算法函数，如果考虑到函数的重载，大约有 100 多个不同的函数模板。本章仅介绍一些常用的算法函数，如果想了解全部的 STL 算法，读者可参考[C++ STL标准库官网](http://www.cplusplus.com/reference/algorithm/" \t "http://c.biancheng.net/view/_blank)。**

**第十一 STL算法和容器中的成员方法同名时，该如何选择？**

通过前面的学习，我们已经掌握了一些 STL 算法的功能和用法。值得一提的是，STL 标准库提供有 70 多种算法函数，其中有些函数名称和 STL 容器模板类中提供的成员方法名相同。  
 例如，**STL 标准库提供了 sort() 和 merge() 函数，而 list 容器模板类中也提供有同名的 sort() 和 merge() 成员方法。再比如，STL 标准库提供有 count()、find()、lower\_bound()、upper\_bound() 以及 equal\_range() 这些函数，而每个关联式容器（除哈希容器外）也提供有相同名称的成员方法。** 那么，当某个 STL 容器提供有和算法同名的成员方法时，应该使用哪一个呢？大多数**情况下，我们应该使用 STL 容器提供的成员方法，而不是同名的 STL 算法，原因包括：**

**（1）虽然同名，但它们的底层实现并不完全相同。相比同名的算法，容器的成员方法和自身结合地更加紧密。**

**（2）相比同名的算法，STL 容器提供的成员方法往往执行效率更高；**举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::find

#include <set> // std::set

#include <string> // std::string

using namespace *std*;

//为 set 容器自定义排序规则，即按照字符串长度进行排序

class mycomp {

public:

bool operator() (const *string*& i, const *string*& j) const {

return i.*length*() < j.*length*();

}

};

int *main*() {

//定义 set 容器，其排序规则为 mycomp

*std*::*set*<*string*, mycomp> myset{ "123","1234","123456" };

//调用 set 容器成员方法

*set*<*string*>::*iterator* iter = myset.*find*(*string*("abcd"));

if (iter == myset.*end*()) {

*cout* << "查找失败" << *endl*;

}

else {

*cout* << \*iter << *endl*;

}

//调用 find() 函数

auto iter2 = *find*(myset.*begin*(), myset.*end*(), *string*("abcd"));

if (iter2 == myset.*end*()) {

*cout* << "查找失败" << *endl*;

}

else {

*cout* << \*iter << *endl*;

}

return 0;

}

**程序执行结果为：**

**1234  
查找失败**

可以看到，程序中分别调用了 find() 函数和 set 容器自带的 find() 成员方法，都用于查找 "abcd" 这个字符串，但查找结果却不相同。其中，find() 成员方法成功找到了和 "abcd" 长度相同的 "1234"，但 find() 函数却查找失败。  
 之所以会这样，是因为 find() 成员方法和 find() 函数底层的实现机制不同。前者会依照 mycomp() 规则查找和 "abcd" 匹配的元素，**而 find() 函数底层仅会依据 "==" 运算符查找 myset 容器中和 "abcd" 相等的元素，所以会查找失败。** 不仅如此，无论是序列式容器还是关联式容器，成员方法的执行效率要高于同名的 STL 算法。仍以 find() 函数和 set 容器中的 find() 成员方法为例。要知道，find() 函数是通过“逐个比对”来实现查找的，它以线性时间运行；而由于 set 容器底层存储结构采用的是红黑树，所以 find() 成员方法以对数时间运行，而非线性时间。  
 换句话说，对于含有一百万个元素的 set 容器，如果使用 find() 成员方法查找目标元素，其最差情况下的比对次数也不会超过 40 次（平均只需要比对 20 次就可以查找成功）；而使用同名的 find() 函数查找目标元素，最差情况下要比对一百万次（平均比对 50 万次才能查找成功）。

所谓“最差情况”，指的是当前 set 容器中未存储有目标元素。

并且需要注意的一点是，虽然有些容器提供的成员方法和某个 STL 算法同名，但该容器只能使用自带的成员方法，而不适用同名的 STL 算法。比如，sort() 函数根本不能应用到 list 容器上，因为该类型容器仅支持双向迭代器，而 sort() 函数的参数类型要求为随机访问迭代器；merge() 函数和 list 容器的 merge() 成员方法之间也存在行为上的不同，即 merge() 函数是不允许修改源数据的，而 list::merge() 成员方法就是对源数据做修改。  
  **总之，当读者需要在 STL 算法与容器提供的同名成员方法之间做选择的时候，应优先考虑成员方法。几乎可以肯定地讲，成员方法的性能更优越，也更贴合当前要操作的容器。**

**第十二 C++ find\_if()和find\_if\_not()函数用法详解**

继《[C++ find()函数](http://c.biancheng.net/view/7489.html" \t "http://c.biancheng.net/view/_blank)》一节后，本节再讲解 2 个和 find() 功能类似的函数，分别为 find\_if()函数和 find\_if\_not() 函数。  
 值得一提的是，find\_if() 和 find\_if\_not() 函数都定义在<algorithm>头文件中。因此在使用它们之前，程序中要先引入此头文件：

#include <algorithm>

**一 C++ find\_if()函数**

和 find() 函数相同，find\_if() 函数也用于在指定区域内执行查找操作。不同的是，前者需要明确指定要查找的元素的值，而后者则允许自定义查找规则。  
 所谓自定义查找规则，实际上指的是有一个形参且返回值类型为 bool 的函数。值得一提的是，该函数可以是一个普通函数（又称为一元谓词函数），比如：

**bool mycomp(int i) {**

**return ((i%2)==1);**

**}**

上面的 mycomp() 就是一个一元谓词函数，其可用来判断一个整数是奇数还是偶数。

如果读者想更深层次地了解 C++ 谓词函数，可阅读《[C++谓词函数](http://c.biancheng.net/view/1513.html" \t "http://c.biancheng.net/view/_blank)》一节。

**也可以是一个函数对象，比如：**

**//以函数对象的形式定义一个 find\_if() 函数的查找规则**

**class mycomp2 {**

**public:**

**bool operator()(const int& i) {**

**return ((i % 2) == 1);**

**}**

**};**

**此函数对象的功能和 mycomp() 函数一样。**  
 确切地说，find\_if() 函数会根据指定的查找规则，在指定区域内查找第一个符合该函数要求（使函数返回 true）的元素。  
  
**find\_if() 函数的语法格式如下：**

**InputIterator find\_if (InputIterator first, InputIterator last, UnaryPredicate pred);**

其中，first 和 last 都为输入迭代器，其组合 [first, last) 用于指定要查找的区域；pred 用于自定义查找规则。

**值得一提的是，由于 first 和 last 都为输入迭代器，意味着该函数适用于所有的序列式容器。甚至当采用适当的谓词函数时，该函数还适用于所有的关联式容器（包括哈希容器）。**

同时，该函数会返回一个输入迭代器，当查找成功时，该迭代器指向的是第一个符合查找规则的元素；反之，如果 find\_if() 函数查找失败，则该迭代器的指向和 last 迭代器相同。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::find\_if

#include <vector> // std::vector

using namespace *std*;

//自定义一元谓词函数

bool mycomp(int i) {

return ((i % 2) == 1);

}

//以函数对象的形式定义一个 find\_if() 函数的查找规则

class mycomp2 {

public:

bool operator()(const int& i) {

return ((i % 2) == 1);

}

};

int *main*() {

*vector*<int> myvector{ 4,2,3,1,5 };

//调用 find\_if() 函数，并以 IsOdd() 一元谓词函数作为查找规则

*vector*<int>::*iterator* it = *find\_if*(myvector.*begin*(), myvector.*end*(), mycomp2());

*cout* << "\*it = " << \*it;

return 0;

}

**程序执行结果为：**

**\*it = 3**

结合程序执行结果不难看出，对于 myvector 容器中的元素 4 和 2 来说，它们都无法使 (i%2)==1 这个表达式成立，因此 mycomp2() 返回 false；而对于元素 3 来说，它可以使 mycomp2() 函数返回 true，因此，find\_if() 函数找到的第一个元素就是元素 3。  
  
  **值得一提的是，[C++ STL find\_if()官网](http://www.cplusplus.com/reference/algorithm/find_if/" \t "http://c.biancheng.net/view/_blank)给出了 find\_if() 函数底层实现的参考代码（如下所示），感兴趣的读者可自行分析，这里不做过多描述：**

**template<class InputIterator, class UnaryPredicate>**

**InputIterator find\_if (InputIterator first, InputIterator last, UnaryPredicate pred)**

**{**

**while (first!=last) {**

**if (pred(\*first)) return first;**

**++first;**

**}**

**return last;**

**}**

**二 C++ find\_if\_not()函数**

find\_if\_not() 函数和 find\_if() 函数的功能恰好相反，通过上面的学习我们知道，find\_if() 函数用于查找符合谓词函数规则的第一个元素，而 find\_if\_not() 函数则用于查找第一个不符合谓词函数规则的元素。  
**find\_if\_not() 函数的语法规则如下所示：**

**InputIterator find\_if\_not (InputIterator first, InputIterator last, UnaryPredicate pred);**

**其中，first 和 last 都为输入迭代器，[first, last) 用于指定查找范围；pred 用于自定义查找规则。**

**和 find\_if() 函数一样，find\_if\_not() 函数也适用于所有的容器，包括所有序列式容器和关联式容器。**

同样，该函数也会返回一个输入迭代器，当 find\_if\_not() 函数查找成功时，该迭代器指向的是查找到的那个元素；反之，如果查找失败，该迭代器的指向和 last 迭代器相同。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::find\_if\_not

#include <vector> // std::vector

using namespace *std*;

//自定义一元谓词函数

bool mycomp(int i) {

return ((i % 2) == 1);

}

int *main*() {

*vector*<int> myvector{ 4,2,3,1,5 };

//调用 find\_if() 函数，并以 mycomp() 一元谓词函数作为查找规则

*vector*<int>::*iterator* it = *find\_if\_not*(myvector.*begin*(), myvector.*end*(), mycomp);

*cout* << "\*it = " << \*it;

return 0;

}

程序执行结果为：

\*it = 4

可以看到，由于第一个元素 4 就不符合 (i%2)==1，因此 find\_if\_not() 成功找到符合条件的元素，并返回一个指向该元素的迭代器。  
 **find\_if\_not() 函数的底层实现和 find\_if() 函数非常类似，[C++ STL find\_if\_not()官网](http://www.cplusplus.com/reference/algorithm/find_if_not/" \t "http://c.biancheng.net/view/_blank)给出了该函数底层实现的参考代码，感兴趣的读者可自行分析，这里不做过多描述：**

**template<class InputIterator, class UnaryPredicate>**

**InputIterator find\_if\_not (InputIterator first, InputIterator last, UnaryPredicate pred)**

**{**

**while (first!=last) {**

**if (!pred(\*first)) return first;**

**++first;**

**}**

**return last;**

**}**

**第十三 C++ find\_end()函数详解**

Find\_end() 函数定义在<algorithm>头文件中，常用于在序列 A 中查找序列 B 最后一次出现的位置。例如，有如下 2 个序列：

**序列 A：1,2,3,4,5,1,2,3,4,5  
序列 B：1,2,3**

通过观察不难发现，序列 B 在序列 A 中出现了 2 次，而借助 find\_end() 函数，可以轻松的得到序列 A 中最后一个（也就是第 2 个） {1,2,3}。  
  
find\_end() 函数的语法格式有 2 种：

**//查找序列 [first1, last1) 中最后一个子序列 [first2, last2)**

**ForwardIterator find\_end (ForwardIterator first1, ForwardIterator last1,**

**ForwardIterator first2, ForwardIterator last2);**

**//查找序列 [first2, last2) 中，和 [first2, last2) 序列满足 pred 规则的最后一个子序列**

**ForwardIterator find\_end (ForwardIterator first1, ForwardIterator last1,**

**ForwardIterator first2, ForwardIterator last2,**

**BinaryPredicate pred);**

其中，各个参数的含义如下：

**（1）first1、last1：都为正向迭代器，其组合 [first1, last1) 用于指定查找范围（也就是上面例子中的序列 A）；**

**（2）first2、last2：都为正向迭代器，其组合 [first2, last2) 用于指定要查找的序列（也就是上面例子中的序列 B）；**

**（3）pred：用于自定义查找规则。该规则实际上是一个包含 2 个参数且返回值类型为 bool 的函数（第一个参数接收 [first1, last1) 范围内的元素，第二个参数接收 [first2, last2) 范围内的元素）。函数定义的形式可以是普通函数，也可以是函数对象。**

**实际上，第一种语法格式也可以看做是包含一个默认的 pred 参数，该参数指定的是一种相等规则，即在 [first1, last1) 范围内查找和 [first2, last2) 中各个元素对应相等的子序列；而借助第二种语法格式，我们可以自定义一个当前场景需要的匹配规则。**

同时，find\_end() 函数会返回一个正向迭代器，当函数查找成功时，该迭代器指向查找到的子序列中的第一个元素；反之，如果查找失败，则该迭代器的指向和 last1 迭代器相同。  
  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::find\_end

#include <vector> // std::vector

using namespace *std*;

//以普通函数的形式定义一个匹配规则

bool mycomp1(int i, int j) {

return (i % j == 0);

}

//以函数对象的形式定义一个匹配规则

class mycomp2 {

public:

bool operator()(const int& i, const int& j) {

return (i % j == 0);

}

};

int *main*() {

*vector*<int> myvector{ 1,2,3,4,8,12,18,1,2,3 };

int myarr[] = { 1,2,3 };

//调用第一种语法格式

*vector*<int>::*iterator* it = *find\_end*(myvector.*begin*(), myvector.*end*(), myarr, myarr + 3);

if (it != myvector.*end*()) {

*cout* << "最后一个{1,2,3}的起始位置为：" << it - myvector.*begin*() << ",\*it = " << \*it << *endl*;

}

int myarr2[] = { 2,4,6 };

//调用第二种语法格式

it = *find\_end*(myvector.*begin*(), myvector.*end*(), myarr2, myarr2 + 3, mycomp2());

if (it != myvector.*end*()) {

*cout* << "最后一个{2,3,4}的起始位置为：" << it - myvector.*begin*() << ",\*it = " << \*it;

}

return 0;

}

**程序执行结果为：**

**匹配{1,2,3}的起始位置为：7,\*it = 1  
匹配{2,3,4}的起始位置为：4,\*it = 8**

上面程序中共调用了 2 次 find\_end() 函数：

（1）第 22 行代码：调用了第一种语法格式的 find\_end() 函数，其功能是在 myvector 容器中查找和 {1,2,3} 相等的最后一个子序列，显然最后一个 {1,2,3} 中元素 1 的位置下标为 7（myvector 容器下标从 0 开始）；

（2）第 29 行代码：调用了第二种格式的 find\_end() 函数，其匹配规则为 mycomp2，即在 myvector 容器中找到最后一个子序列，该序列中的元素能分别被 {2、4、6} 中的元素整除。显然，myvector 容器中 {4,8,12} 和 {8,12,18} 都符合，该函数会找到后者并返回一个指向元素 8 的迭代器。

**注意，find\_end() 函数的第一种语法格式，其底层是借助 == 运算符实现的。这意味着，如果 [first1, last1] 和 [first2, last2] 区域内的元素为自定义的类对象或结构体变量时，使用该函数之前需要对 == 运算符进行重载。**

**C++ STL标准库官方给出了 find\_end() 函数底层实现的参考代码，感兴趣的读者可自行分析，这里不再做过多描述：**

**template<class ForwardIterator1, class ForwardIterator2>**

**ForwardIterator1 find\_end(ForwardIterator1 first1, ForwardIterator1 last1,**

**ForwardIterator2 first2, ForwardIterator2 last2)**

**{**

**if (first2 == last2) return last1; // specified in C++11**

**ForwardIterator1 ret = last1;**

**while (first1 != last1)**

**{**

**ForwardIterator1 it1 = first1;**

**ForwardIterator2 it2 = first2;**

**while (\*it1 == \*it2) { // or: while (pred(\*it1,\*it2)) for version (2)**

**++it1; ++it2;**

**if (it2 == last2) { ret = first1; break; }**

**if (it1 == last1) return ret;**

**}**

**++first1;**

**}**

**return ret;**

**}**

另外，C++ STL 标准库还提供了和 find\_end() 函数功能恰恰相反的 search() 函数，有关该函数的用法，可阅读《[C++ search() 函数](http://c.biancheng.net/view/7503.html" \t "http://c.biancheng.net/view/_blank)》一节。

**第十四 C++ find\_first\_of()函数完全攻略**

在某些情境中，我们可能需要在 A 序列中查找和 B 序列中任意元素相匹配的第一个元素，这时就可以使用 find\_first\_of() 函数。  
 仅仅用一句话概述 find\_first\_of() 函数的功能，读者可能并不理解。别急，下面我们将从语法格式的角度继续阐述该函数的功能。  
  
find\_first\_of() 函数定义于<algorithm>头文件中，因此使用该函数之前，程序中要先引入此头文件：

#include <algorithm>

**find\_first\_of() 函数有 2 种语法格式，分别是：**

**//以判断两者相等作为匹配规则**

**InputIterator find\_first\_of (InputIterator first1, InputIterator last1,**

**ForwardIterator first2, ForwardIterator last2);**

**//以 pred 作为匹配规则**

**InputIterator find\_first\_of (InputIterator first1, InputIterator last1,**

**ForwardIterator first2, ForwardIterator last2,**

**BinaryPredicate pred);**

其中，各个参数的含义如下：

（1）first1、last1：都为输入迭代器，它们的组合 [first1, last1) 用于指定该函数要查找的范围；

（2）first2、last2：都为正向迭代器，它们的组合 [first2, last2) 用于指定要进行匹配的元素所在的范围；

（3）pred：可接收一个包含 2 个形参且返回值类型为 bool 的函数，该函数可以是普通函数（又称为二元谓词函数），也可以是函数对象。

**有关谓词函数，读者可阅读《[C++谓词函数](http://c.biancheng.net/view/1513.html" \t "http://c.biancheng.net/view/_blank)》一节详细了解。**

find\_first\_of() 函数用于在 [first1, last1) 范围内查找和 [first2, last2) 中任何元素相匹配的第一个元素。如果匹配成功，该函数会返回一个指向该元素的输入迭代器；反之，则返回一个和 last1 迭代器指向相同的输入迭代器。  
值得一提的是，不同语法格式的匹配规则也是不同的：

（1）第 1 种语法格式：逐个取 [first1, last1) 范围内的元素（假设为 A），和 [first2, last2) 中的每个元素（假设为 B）做 A==B 运算，如果成立则匹配成功；

（2）第 2 种语法格式：逐个取 [first1, last1) 范围内的元素（假设为 A），和 [first2, last2) 中的每个元素（假设为 B）一起带入 pred(A, B) 谓词函数，如果函数返回 true 则匹配成功。

**注意，当采用第一种语法格式时，如果 [first1, last1) 或者 [first2, last2) 范围内的元素类型为自定义的类对象或者结构体变量，此时应对 == 运算符进行重载，使其适用于当前场景。**

举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::find\_first\_of

#include <vector> // std::vector

using namespace *std*;

//自定义二元谓词函数，作为 find\_first\_of() 函数的匹配规则

bool mycomp(int c1, int c2) {

return (c2 % c1 == 0);

}

//以函数对象的形式定义一个 find\_first\_of() 函数的匹配规则

class mycomp2 {

public:

bool operator()(const int& c1, const int& c2) {

return (c2 % c1 == 0);

}

};

int *main*() {

char url[] = "http://c.biancheng.net/stl/";

char ch[] = "stl";

//调用第一种语法格式，找到 url 中和 "stl" 任一字符相同的第一个字符

char\* it = *find\_first\_of*(url, url + 27, ch, ch + 4);

if (it != url + 27) {

*cout* << "\*it = " << \*it << '\n';

}

*vector*<int> myvector{ 5,7,3,9 };

int inter[] = { 4,6,8 };

//调用第二种语法格式，找到 myvector 容器中和 3、5、7 任一元素有 c2%c1=0 关系的第一个元素

*vector*<int>::*iterator* iter = *find\_first\_of*(myvector.*begin*(), myvector.*end*(), inter, inter + 3, mycomp2());

if (iter != myvector.*end*()) {

*cout* << "\*iter = " << \*iter;

}

return 0;

}

**程序执行结果为：**

**\*it = t  
\*iter = 3**

此程序给读者演示了 find\_first\_of() 函数 2 种语法格式的用法。其中第 20 行代码中 find\_first\_of() 函数发挥的功能是，在 url 字符数组中逐个查找和 's'、't'、'l' 这 3 个字符相等的字符，显然 url 数组第 2 个字符 't' 就符合此规则。  
 在第 29 行代码中，find\_first\_of() 会逐个提取 myvector 容器中的每个元素（假设为 A），并尝试和 inter 数组中的每个元素（假设为 B）一起带入 mycomp2(A, B) 函数对象中。显然，当将 myvector 容器中的元素 3 和 inter 数组中的元素 6 带入该函数时，c2 % c1=0 表达式第一次成立。  
 C++ STL 标准库给出了 find\_first\_of() 函数底层实现的参考代码，感兴趣的读者可自行分析：

template<class InputIt, class ForwardIt, class BinaryPredicate>

InputIt find\_first\_of(InputIt first, InputIt last,

ForwardIt s\_first, ForwardIt s\_last,

BinaryPredicate p)

{

for (; first != last; ++first) {

for (ForwardIt it = s\_first; it != s\_last; ++it) {

//第二种语法格式换成 if (p(\*first, \*it))

if (p(\*first, \*it)) {

return first;

}

}

}

return last;

}

**第十五 C++ adjacent\_find()函数用法详解**

adjacent\_find() 函数用于在指定范围内查找 2 个连续相等的元素。该函数的语法格式为：

**//查找 2 个连续相等的元素**

**ForwardIterator adjacent\_find (ForwardIterator first, ForwardIterator last);**

**//查找 2 个连续满足 pred 规则的元素**

**ForwardIterator adjacent\_find (ForwardIterator first, ForwardIterator last,**

**BinaryPredicate pred);**

其中，first 和 last 都为正向迭代器，其组合 [first, last) 用于指定该函数的查找范围；pred 用于接收一个包含 2 个参数且返回值类型为 bool 的函数，以实现自定义查找规则。

**值得一提的是，pred 参数接收的函数既可以定义为普通函数，也可以用函数对象的形式定义。有关谓词函数，读者可阅读《[C++谓词函数](http://c.biancheng.net/view/1513.html" \t "http://c.biancheng.net/view/_blank)》一节详细了解。**

**另外，该函数会返回一个正向迭代器，当函数查找成功时，该迭代器指向的是连续相等元素的第 1 个元素；而如果查找失败，该迭代器的指向和 last 迭代器相同。**  
值得一提的是，adjacent\_find() 函数定义于<algorithm>头文件中，因此使用该函数之前，程序中要先引入此头文件：

#include <algorithm>

举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::adjacent\_find

#include <vector> // std::vector

using namespace *std*;

//以创建普通函数的形式定义一个查找规则

bool mycomp1(int i, int j) {

return (i == j);

}

//以函数对象的形式定义一个查找规则

class mycomp2 {

public:

bool operator()(const int& \_Left, const int& \_Right) {

return (\_Left == \_Right);

}

};

int *main*() {

*std*::*vector*<int> myvector{ 5,20,5,30,30,20,10,10,20 };

//调用第一种语法格式

*std*::*vector*<int>::*iterator* it = *adjacent\_find*(myvector.*begin*(), myvector.*end*());

if (it != myvector.*end*()) {

*cout* << "one : " << \*it << '\n';

}

//调用第二种格式，也可以使用 mycomp1

it = *adjacent\_find*(++it, myvector.*end*(), mycomp2());

if (it != myvector.*end*()) {

*cout* << "two : " << \*it;

}

return 0;

}

**程序执行结果为：**

**one : 30  
two : 10**

可以看到，程序中调用了 2 次 adjacent\_find() 函数：

（1）第 19 行：使用该函数的第一种语法格式，查找整个 myvector 容器中首个连续 2 个相等的元素，显然最先找到的是 30；

（2）第 25 行：使用该函数的第二种语法格式，查找 {30,20,10,10,20} 部分中是否有连续 2 个符合 mycomp2 规则的元素。不过，程序中自定义的 mycomp1 或 mycomp2 查找规则也是查找 2 个连续相等的元素，因此最先找到的是元素 10。

**注意，对于第一种语法格式的 adjacent\_find() 函数，其底层使用的是 == 运算符来判断连续 2 个元素是否相等。这意味着，如果指定区域内的元素类型为自定义的类对象或者结构体变量时，需要先对 == 运算符进行重载，然后才能使用此函数。**

C++ STL标准库官方给出了 adjacent\_find() 函数底层实现的参考代码，感兴趣的读者可自行分析，这里不再做过多描述：

template <class ForwardIterator>

ForwardIterator adjacent\_find (ForwardIterator first, ForwardIterator last)

{

if (first != last)

{

ForwardIterator next=first; ++next;

while (next != last) {

if (\*first == \*next) // 或者 if (pred(\*first,\*next)), 对应第二种语法格式

return first;

++first; ++next;

}

}

return last;

}

**第十六 C++ search()函数用法完全攻略**

通过《[C++ find\_end()函数](http://c.biancheng.net/view/7500.html" \t "http://c.biancheng.net/view/_blank)》一节的讲解我们知道，find\_end() 函数用于在序列 A 中查找序列 B 最后一次出现的位置。那么，如果想知道序列 B 在序列 A 中第一次出现的位置，该如何实现呢？可以借助 search() 函数。  
 search() 函数定义在<algorithm>头文件中，其功能恰好和 find\_end() 函数相反，用于在序列 A 中查找序列 B 第一次出现的位置。  
例如，仍以如下两个序列为例：

**序列 A：1,2,3,4,5,1,2,3,4,5  
序列 B：1,2,3**

可以看到，序列 B 在序列 A 中出现了 2 次。借助 find\_end() 函数，我们可以找到序列 A 中最后一个（也就是第 2 个）{1,2,3}；而借助 search() 函数，我们可以找到序列 A 中第 1 个 {1,2,3}。  
**和 find\_end() 相同，search() 函数也提供有以下 2 种语法格式：**

**//查找 [first1, last1) 范围内第一个 [first2, last2) 子序列**

**ForwardIterator search (ForwardIterator first1, ForwardIterator last1,**

**ForwardIterator first2, ForwardIterator last2);**

**//查找 [first1, last1) 范围内，和 [first2, last2) 序列满足 pred 规则的第一个子序列**

**ForwardIterator search (ForwardIterator first1, ForwardIterator last1,**

**ForwardIterator first2, ForwardIterator last2,**

**BinaryPredicate pred);**

其中，各个参数的含义分别为：

（1）first1、last1：都为正向迭代器，其组合 [first1, last1) 用于指定查找范围（也就是上面例子中的序列 A）；

（2）first2、last2：都为正向迭代器，其组合 [first2, last2) 用于指定要查找的序列（也就是上面例子中的序列 B）；

（3）pred：用于自定义查找规则。该规则实际上是一个包含 2 个参数且返回值类型为 bool 的函数（第一个参数接收 [first1, last1) 范围内的元素，第二个参数接收 [first2, last2) 范围内的元素）。函数定义的形式可以是普通函数，也可以是函数对象。

实际上，第一种语法格式也可以看做是包含一个默认的 pred 参数，该参数指定的是一种相等规则，即在 [first1, last1) 范围内查找和 [first2, last2) 中各个元素对应相等的子序列；而借助第二种语法格式，我们可以自定义一个当前场景需要的匹配规则。

同时，search() 函数会返回一个正向迭代器，当函数查找成功时，该迭代器指向查找到的子序列中的第一个元素；反之，如果查找失败，则该迭代器的指向和 last1 迭代器相同。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::search

#include <vector> // std::vector

using namespace *std*;

//以普通函数的形式定义一个匹配规则

bool mycomp1(int i, int j) {

return (i % j == 0);

}

//以函数对象的形式定义一个匹配规则

class mycomp2 {

public:

bool operator()(const int& i, const int& j) {

return (i % j == 0);

}

};

int *main*() {

*vector*<int> myvector{ 1,2,3,4,8,12,18,1,2,3 };

int myarr[] = { 1,2,3 };

//调用第一种语法格式

*vector*<int>::*iterator* it = *search*(myvector.*begin*(), myvector.*end*(), myarr, myarr + 3);

if (it != myvector.*end*()) {

*cout* << "第一个{1,2,3}的起始位置为：" << it - myvector.*begin*() << ",\*it = " << \*it << *endl*;

}

int myarr2[] = { 2,4,6 };

//调用第二种语法格式

it = *search*(myvector.*begin*(), myvector.*end*(), myarr2, myarr2 + 3, mycomp2());

if (it != myvector.*end*()) {

*cout* << "第一个{2,3,4}的起始位置为：" << it - myvector.*begin*() << ",\*it = " << \*it;

}

return 0;

}

**程序执行结果为：**

**第一个{1,2,3}的起始位置为：0,\*it = 1  
第一个{2,3,4}的起始位置为：3,\*it = 4**

通过程序的执行结果可以看到，第 22 行代码借助 search() 函数找到了 myvector 容器中第一个 {1,2,3}，并返回了一个指向元素 1 的迭代器（其下标位置为 0）。  
 而在第 29 行中，search() 函数使用的是第 2 种格式，其自定义了 mycomp2 匹配规则，即在 myvector 容器中找到第一个连续的 3 个元素，它们能分别被 2、4、6 整除。显然，myvector 容器中符合要求的子序列有 2 个，分别为 {4,8,12} 和 {8,12,18}，但 search() 函数只会查找到第一个，并返回指向元素 4 的迭代器（其下标为 3）。

**注意，search() 函数的第一种语法格式，其底层是借助 == 运算符实现的。这意味着，如果 [first1, last1] 和 [first2, last2] 区域内的元素为自定义的类对象或结构体变量时，使用该函数之前需要对 == 运算符进行重载。**

C++ STL标准库官方给出了 search() 函数底层实现的参考代码，感兴趣的读者可自行分析，这里不再做过多描述：

template<class ForwardIterator1, class ForwardIterator2>

ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,

ForwardIterator2 first2, ForwardIterator2 last2)

{

if (first2 == last2) return first1;

while (first1 != last1)

{

ForwardIterator1 it1 = first1;

ForwardIterator2 it2 = first2;

while (\*it1 == \*it2) { // 或者 while (pred(\*it1,\*it2)) 对应第二种语法格式

if (it2 == last2) return first1;

if (it1 == last1) return last1;

++it1; ++it2;

}

++first1;

}

return last1;

}

**第十七 C++ search\_n()函数用法**

《[C++ search()函数](http://c.biancheng.net/view/7503.html" \t "http://c.biancheng.net/view/_blank)》一节中，已经详细介绍了 search() 函数的功能和用法。在此基础上，本节再介绍一个功能类似的函数，即 search\_n() 函数。  
 和 search() 一样，search\_n() 函数也定义在<algorithm>头文件中，用于在指定区域内查找第一个符合要求的子序列。不同之处在于，前者查找的子序列中可包含多个不同的元素，而后者查找的只能是包含多个相同元素的子序列。  
**关于 search() 函数和 search\_n() 函数的区别，给大家举个例子，下面有 3 个序列：**

**序列 A：1,2,3,4,4,4,1,2,3,4,4,4  
序列 B：1,2,3  
序列 C：4,4,4**

如果想查找序列 B 在序列 A 中第一次出现的位置，就只能使用 search() 函数；而如果想查找序列 C 在序列 A 中第一次出现的位置，既可以使用 search() 函数，也可以使用 search\_n() 函数。  
  
**search\_n() 函数的语法格式如下：**

**//在 [first, last] 中查找 count 个 val 第一次连续出现的位置**

**ForwardIterator search\_n (ForwardIterator first, ForwardIterator last,**

**Size count, const T& val);**

**//在 [first, last] 中查找第一个序列，该序列和 count 个 val 满足 pred 匹配规则**

**ForwardIterator search\_n ( ForwardIterator first, ForwardIterator last,**

**Size count, const T& val, BinaryPredicate pred );**

其中，各个参数的含义分别为：

（1）first、last：都为正向迭代器，其组合 [first, last) 用于指定查找范围（也就是上面例子中的序列 A）；

（2）count、val：指定要查找的元素个数和元素值，以上面的序列 B 为例，该序列实际上就是 3 个元素 4，其中 count 为 3，val 为 4；

（3）pred：用于自定义查找规则。该规则实际上是一个包含 2 个参数且返回值类型为 bool 的函数（第一个参数接收[first, last) 范围内的元素，第二个参数接收 val）。函数定义的形式可以是普通函数，也可以是函数对象。

**实际上，第一种语法格式也可以看做是包含一个默认的 pred 参数，该参数指定的是一种相等规则，即在 [first, last) 范围内查找和 count 个 val 相等的子序列；而借助第二种语法格式，我们可以自定义一个当前场景需要的匹配规则。**

同时，search\_n() 函数会返回一个正向迭代器，当函数查找成功时，该迭代器指向查找到的子序列中的第一个元素；反之，如果查找失败，则该迭代器的指向和 last 迭代器相同。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::search\_n

#include <vector> // std::vector

using namespace *std*;

//以普通函数的形式定义一个匹配规则

bool mycomp1(int i, int j) {

return (i % j == 0);

}

//以函数对象的形式定义一个匹配规则

class mycomp2 {

public:

bool operator()(const int& i, const int& j) {

return (i % j == 0);

}

};

int *main*() {

int a[] = { 1,2,3,4,4,4,1,2,3,4,4,4 };

//调用第一种语法格式,查找 myvector 容器中第一个 {4,4,4}

int\* it = *search\_n*(a, a + 12, 3, 4);

if (it != a + 12) {

*cout* << "one：" << it - a << ",\*it = " << \*it << *endl*;

}

*vector*<int> myvector{ 1,2,4,8,3,4,6,8 };

//调用第二种语法格式，以自定义的 mycomp2 作为匹配规则，查找 myvector 容器中和 {16,16,16} 满足 mycomp2 规则的序列

*vector*<int>::*iterator* iter = *search\_n*(myvector.*begin*(), myvector.*end*(), 3, 2, mycomp2());

if (iter != myvector.*end*()) {

*cout* << "two：" << iter - myvector.*begin*() << ",\*iter = " << \*iter;

}

return 0;

}

**程序执行结果为：**

**one：3,\*it = 4  
two：1,\*iter = 2**

程序中先后调用了 2 种语法格式的 search\_n() 函数，其中第 28 行代码中，search\_n() 函数不再采用默认的相等匹配规则，而是采用了自定义了 mycomp2 匹配规则。这意味着，该函数会去 myvector 容器中查找一个子序列，该序列中的 3 个元素都满足和 2 有 (i%j == 0) 的关系。显然，myvector 容器中符合条件的子序列有 2 个，分别为 {2,4,8} 和 {4,6,8}，但 search\_n() 函数只会查找到 {2,4,8}。

**注意，search\_n() 函数的第一种语法格式，其底层是借助 == 运算符实现的。这意味着，如果 [first, last] 区域内的元素为自定义的类对象或结构体变量时，使用此格式的 search\_n() 函数之前，需要对 == 运算符进行重载。**

C++ STL标准库官方给出了 search\_n() 函数底层实现的参考代码，感兴趣的读者可自行分析，这里不再做过多描述：

template<class ForwardIterator, class Size, class T>

ForwardIterator search\_n (ForwardIterator first, ForwardIterator last,

Size count, const T& val)

{

ForwardIterator it, limit;

Size i;

limit=first; std::advance(limit,std::distance(first,last)-count);

while (first!=limit)

{

it = first; i=0;

while (\*it==val) // 或者 while (pred(\*it,val))，对应第二种格式

{ ++it; if (++i==count) return first; }

++first;

}

return last;

}

**第十八 C++ partition()和stable\_partition()函数详解**

partition 可直译为“分组”，partition() 函数可根据用户自定义的筛选规则，重新排列指定区域内存储的数据，使其分为 2 组，第一组为符合筛选条件的数据，另一组为不符合筛选条件的数据。  
 举个例子，假设有一个数组 a[9]，其存储数据如下：

**1 2 3 4 5 6 7 8 9**

在此基础上，如果设定筛选规则为 i%2=0（其中 i 即代指数组 a 中的各个元素），则借助 partition() 函数，a[9] 数组中存储数据的顺序可能变为：

1 9 3 7 5 6 4 8 2

其中 {1,9,3,7,5} 为第一组，{6,4,8,2} 为第二组。显然前者中的各个元素都符合筛选条件，而后者则都不符合。由此还可看出，partition() 函数只会根据筛选条件将数据进行分组，并不关心分组后各个元素具体的存储位置。

**如果想在分组之后仍不改变各元素之间的相对位置，可以选用 stable\_partition() 函数。有关此函数的功能和用法，本节后续会做详细讲解。**

值得一提得是，partition() 函数定义于<algorithm>头文件中，因此在使用该函数之前，程序中应先引入此头文件：

#include <algorithm>

**如下为 partition() 函数的语法格式：**

**ForwardIterator partition (ForwardIterator first,**

**ForwardIterator last,**

**UnaryPredicate pred);**

其中，first 和 last 都为正向迭代器，其组合 [first, last) 用于指定该函数的作用范围；pred 用于指定筛选规则。

**所谓筛选规则，其本质就是一个可接收 1 个参数且返回值类型为 bool 的函数，可以是普通函数，也可以是一个函数对象。**

同时，partition() 函数还会返回一个正向迭代器，其指向的是两部分数据的分界位置，更确切地说，指向的是第二组数据中的第 1 个元素。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::partition

#include <vector> // std::vector

using namespace *std*;

//以普通函数的方式定义partition()函数的筛选规则

bool mycomp(int i) { return (i % 2) == 0; }

//以函数对象的形式定义筛选规则

class mycomp2 {

public:

bool operator()(const int& i) {

return (i % 2 == 0);

}

};

int *main*() {

*std*::*vector*<int> myvector{ 1,2,3,4,5,6,7,8,9 };

*std*::*vector*<int>::*iterator* bound;

//以 mycomp2 规则，对 myvector 容器中的数据进行分组

bound = *std*::*partition*(myvector.*begin*(), myvector.*end*(), mycomp2());

for (*std*::*vector*<int>::*iterator* it = myvector.*begin*(); it != myvector.*end*(); ++it) {

*cout* << \*it << " ";

}

*cout* << "\nbound = " << \*bound;

return 0;

}

**程序执行结果为：**

**8 2 6 4 5 3 7 1 9  
bound = 5**

可以看到，程序中借助 partition() 对 myvector 容器中的数据进行了再加工，基于 mycomp2() 筛选规则，能够被 2 整除的元素位于第 1 组，不能被 2 整除的元素位于第 2 组。  
 同时，parition() 函数会返回一个迭代器，通过观察程序的执行结果可以看到，该迭代器指向的是元素 5，同时也是第 2 组数据中的第 1 个元素。

值得一提的是，[C++ STL partition()函数官方](http://www.cplusplus.com/reference/algorithm/partition/" \t "http://c.biancheng.net/view/_blank)给出了该函数底层实现的参考代码，感兴趣的读者可自行前往分析，这里不再做过多描述。

**一 C++ stable\_partition()函数**

前面提到，partition() 函数只负责对指定区域内的数据进行分组，并不保证各组中元素的相对位置不发生改变。而如果想在分组的同时保证不改变各组中元素的相对位置，可以使用 stable\_partition() 函数。  
 也就是说，stable\_partition() 函数可以保证对指定区域内数据完成分组的同时，不改变各组内元素的相对位置。  
仍以数组 a[9] 举例，其存储的数据如下：

1 2 3 4 5 6 7 8 9

假定筛选规则为 i%2=0（其中 i 即代指数组 a 中的各个元素），则借助 stable\_partition() 函数，a[9] 数组中存储数据的顺序为：

2 4 6 8 1 3 5 7 9

其中 {2,4,6,8} 为一组，{1,3,5,7,9} 为另一组。通过和先前的 a[9] 对比不难看出，各个组中元素的相对位置没有发生改变。

**所谓元素的相对位置不发生改变，以 {2,4,6,8} 中的元素 4 为例，在原 a[9] 数组中，该元素位于 2 的右侧，6 和 8 的左侧；在经过 stable\_partition() 函数处理后的 a[9] 数组中，元素 4 仍位于 2 的右侧，6 和 8 的左侧。因此，该元素的相对位置确实没有发生改变。**

**stable\_partition() 函数定义在<algorithm>头文件中，其语法格式如下：**

**BidirectionalIterator stable\_partition (BidirectionalIterator first,**

**BidirectionalIterator last,**

**UnaryPredicate pred);**

**其中，first 和 last 都为双向迭代器，其组合 [first, last) 用于指定该函数的作用范围；pred 用于指定筛选规则。**  
 同时，stable\_partition() 函数还会返回一个双向迭代器，其指向的是两部分数据的分界位置，更确切地说，指向的是第二组数据中的第 1 个元素。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::stable\_partition

#include <vector> // std::vector

using namespace *std*;

//以普通函数的方式定义partition()函数的筛选规则

bool mycomp(int i) { return (i % 2) == 1; }

//以函数对象的形式定义筛选规则

class mycomp2 {

public:

bool operator()(const int& i) {

return (i % 2 == 1);

}

};

int *main*() {

*std*::*vector*<int> myvector{ 1,2,3,4,5,6,7,8,9 };

*std*::*vector*<int>::*iterator* bound;

//以 mycomp2 规则，对 myvector 容器中的数据进行分组

bound = *std*::*stable\_partition*(myvector.*begin*(), myvector.*end*(), mycomp);

for (*std*::*vector*<int>::*iterator* it = myvector.*begin*(); it != myvector.*end*(); ++it) {

*cout* << \*it << " ";

}

*cout* << "\nbound = " << \*bound;

return 0;

}

**程序执行结果为：**

**1 3 5 7 9 2 4 6 8  
bound = 2**

**第十九 C++ partition\_copy()函数详解**

《C++ partition()和stable\_partition()函数》一节中，已经详细介绍了 partition() 和 stable\_partition() 函数的功能和用法。不知道读者是否发现，这 2 个函数在实现功能时，都直接修改了原序列中元素的存储位置。

而在某些场景中，我们需要类似 partition() 或者 stable\_partition() 函数“分组”的功能，但并不想对原序列做任何修改。这种情况下，就可以考虑使用 partition\_copy() 函数。

和 stable\_partition() 一样，partition\_copy() 函数也能按照某个筛选规则对指定区域内的数据进行“分组”，并且分组后不会改变各个元素的相对位置。更重要的是，partition\_copy() 函数不会对原序列做修改，而是以复制的方式将序列中各个元组“分组”到其它的指定位置存储。

举个例子，有如下一个数组 a[10]：

1 2 3 4 5 6 7 8 9

假设筛选条件为 i % 2 == 0（也就是筛选出偶数），如果借助 stable\_partition() 函数，则数组 a[10] 中元素的存储顺序会变成：

2 4 6 8 1 3 5 7 9

**而如果选用同样的筛选规则，使用 partition\_copy() 函数还需要为其配备 2 个存储空间（例如 b[10] 和 c[10]），其中 b[10] 用于存储符合筛选条件的偶数，而 c[10] 用于存储不符合筛选条件的奇数，也就是说，partition\_copy() 函数执行的最终结果为：**

**a[10]: 1 2 3 4 5 6 7 8 9**

**b[10] : 2 4 6 8**

**c[10] : 1 3 5 7 9**

注意，这里仅展示了 b[10] 和 c[10] 数组中存储的有效数据。

**值得一提的是，partition\_copy() 函数定义在<algorithm>头文件中，其语法格式如下：**

**pair<OutputIterator1, OutputIterator2> partition\_copy(**

**InputIterator first, InputIterator last,**

**OutputIterator1 result\_true, OutputIterator2 result\_false,**

**UnaryPredicate pred);**

其中，各个参数的含义为：

（1）first、last：都为输入迭代器，其组合[first, last) 用于指定该函数处理的数据区域；

（2）result\_true：为输出迭代器，其用于指定某个存储区域，以存储满足筛选条件的数据；

（3）result\_false：为输出迭代器，其用于指定某个存储区域，以存储满足筛选条件的数据；

（4）pred：用于指定筛选规则，其本质就是接收一个具有 1 个参数且返回值类型为 bool 的函数。注意，该函数既可以是普通函数，还可以是一个函数对象。

除此之外，该函数还会返回一个 pair 类型值，其包含 2 个迭代器，第一个迭代器指向的是 result\_true 区域内最后一个元素之后的位置；第二个迭代器指向的是 result\_false 区域内最后一个元素之后的位置

举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::partition\_copy

#include <vector> // std::vector

using namespace *std*;

//以普通函数的方式定义筛选规则

bool mycomp(int i) { return (i % 2) == 0; }

//以函数对象的形式定义筛选规则

class mycomp2 {

public:

bool operator()(const int& i) {

return (i % 2 == 0);

}

};

int *main*() {

*vector*<int> myvector{ 1,2,3,4,5,6,7,8,9 };

int b[10] = { 0 }, c[10] = { 0 };

//以 mycomp 规则，对 myvector 容器中的数据进行分组，这里的 mycomp 还可以改为 mycomp2()，即以 mycomp2 为筛选规则

*pair*<int\*, int\*> result = *partition\_copy*(myvector.*begin*(), myvector.*end*(), b, c, mycomp);

*cout* << "b[10]：";

for (int\* p = b; p < result.*first*; p++) {

*cout* << \*p << " ";

}

*cout* << "\nc[10]：";

for (int\* p = c; p < result.*second*; p++) {

*cout* << \*p << " ";

}

return 0;

}

**程序执行结果为：**

**b[10]：2 4 6 8**

**c[10]：1 3 5 7 9**

程序中仅演示了如何用数组来存储 partition\_copy() 函数分组后的数据，当然也可以用容器来存储。

C++ 标准库中还给出了 partition\_copy() 函数底层实现的参考代码，感兴趣的读者可自行研究，这里不再进行过多赘述。

template <class InputIterator, class OutputIterator1,

class OutputIterator2, class UnaryPredicate pred>

pair<OutputIterator1, OutputIterator2>

partition\_copy(InputIterator first, InputIterator last,

OutputIterator1 result\_true, OutputIterator2 result\_false,

UnaryPredicate pred)

{

while (first != last) {

if (pred(\*first)) {

\*result\_true = \*first;

++result\_true;

}

else {

\*result\_false = \*first;

++result\_false;

}

++first;

}

return std::make\_pair(result\_true, result\_false);

}

**第二十 C++ partition\_point()函数**

在前面章节中，我们系统学习了 partition()、stable\_partition() 和 partition\_copy() 这 3 个函数，它们的功能本质上都是根据某个筛选规则对指定范围内的数据进行分组（即符合条**件的为一组，不符合条件的为另一组），并且反馈给我们两组数据之间的分界位置。**

**事实上，有些数据本身就已经是按照某个筛选规则分好组的，例如：**

**1,2,3,4,5,6,7 <-- 根据规则 i<4，{1,2,3} 为一组，{4,5,6,7} 为另一组**

**2,4,6,8,1,3,5,7,9 <-- 根据规则 i%2=0，{2,4,6,8} 为一组，{1,3,5,7,9} 为另一组**

类似上面这样已经“分好组”的数据，在使用时会有一个问题，即不知道两组数据之间的分界在什么位置。有读者可能想到，再调用一次 partition()、stale\_partition() 或者 partition\_copy() 不就可以了吗？这种方法确实可行，但对已经分好组的数据再进行一次分组，是没有任何必要的。  
 实际上，对于如何在已分好组的数据中找到分界位置，C++ 11标准库提供了专门解决此问题的函数，即 partition\_point() 函数。  
**partition\_point() 函数定义在<algorithm>头文件中，其语法格式为：**

**ForwardIterator partition\_point (ForwardIterator first, ForwardIterator last,**

**UnaryPredicate pred);**

**其中，first 和 last 为正向迭代器，[first, last) 用于指定该函数的作用范围；pred 用于指定数据的筛选规则。**

所谓筛选规则，其实就是包含 1 个参数且返回值类型为 bool 的函数，此函数可以是一个普通函数，也可以是一个函数对象。

同时，该函数会返回一个正向迭代器，该迭代器指向的是 [first, last] 范围内第一个不符合 pred 筛选规则的元素。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::partition\_point

#include <vector> // std::vector

using namespace *std*;

//以普通函数的方式定义筛选规则

bool mycomp(int i) { return (i % 2) == 0; }

//以函数对象的形式定义筛选规则

class mycomp2 {

public:

bool operator()(const int& i) {

return (i % 2 == 0);

}

};

int *main*() {

*vector*<int> myvector{ 2,4,6,8,1,3,5,7,9 };

//根据 mycomp 规则，为 myvector 容器中的数据找出分界

*vector*<int>::*iterator* iter = *partition\_point*(myvector.*begin*(), myvector.*end*(), mycomp);

//输出第一组的数据

for (auto it = myvector.*begin*(); it != iter; ++it) {

*cout* << \*it << " ";

}

*cout* << "\n";

//输出第二组的数据

for (auto it = iter; it != myvector.*end*(); ++it) {

*cout* << \*it << " ";

}

*cout* << "\n\*iter = " << \*iter;

return 0;

}

**程序执行结果为：**

**2 4 6 8  
1 3 5 7 9  
\*iter = 1**

通过分析程序并结合输出结果可以看到，partition\_point() 返回了一个指向元素 1 的迭代器，而该元素为 myvector 容器中第一个不符合 mycomp 规则的元素，同时其也可以第二组数据中第一个元素。  
 值得一提的是，C++ 11标准库中给出了 partition\_point() 函数底层实现的参考代码（如下所示），感兴趣的读者可自行分析，这里不再进行赘述：

template <class ForwardIterator, class UnaryPredicate>

ForwardIterator partition\_point (ForwardIterator first, ForwardIterator last,

UnaryPredicate pred)

{

auto n = distance(first,last);

while (n>0)

{

ForwardIterator it = first;

auto step = n/2;

std::advance (it,step);

if (pred(\*it)) { first=++it; n-=step+1; }

else n=step;

}

return first;

}

**第二十一 C++ lower\_bound()函数用法详解**

前面章节中，已经给大家系统地介绍了几个查找函数，如 find()、find\_if()、search() 等。值得一提的是，这些函数的底层实现都采用的是顺序查找（逐个遍历）的方式，在某些场景中的执行效率并不高。例如，当指定区域内的数据处于有序状态时，如果想查找某个目标元素，更推荐使用二分查找的方法（相比顺序查找，二分查找的执行效率更高）。  
 幸运的是，除了前面讲过的几个函数外，C++ STL标准库中还提供有 lower\_bound()、upper\_bound()、equal\_range() 以及 binary\_search() 这 4 个查找函数，它们的底层实现采用的都是二分查找的方式。  
 从本节开始，将给大家系统地讲解这 4 个二分查找函数的功能和用法，这里先从lower\_bound() 函数开始讲起。

有关二分查找算法的实现原理，感兴趣的读者可阅读《[二分查找（折半查找）](http://c.biancheng.net/view/3428.html" \t "http://c.biancheng.net/view/_blank)》一节做详细了解。

**一 C++ lower\_bound()函数**

lower\_bound() 函数用于在指定区域内查找不小于目标值的第一个元素。也就是说，使用该函数在指定范围内查找某个目标值时，最终查找到的不一定是和目标值相等的元素，还可能是比目标值大的元素。  
**lower\_bound() 函数定义在<algorithm>头文件中，其语法格式有 2 种，分别为：**

**//在 [first, last) 区域内查找不小于 val 的元素**

**ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last,**

**const T& val);**

**//在 [first, last) 区域内查找第一个不符合 comp 规则的元素**

**ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last,**

**const T& val, Compare comp);**

其中，first 和 last 都为正向迭代器，[first, last) 用于指定函数的作用范围；val 用于指定目标元素；comp 用于自定义比较规则，此参数可以接收一个包含 2 个形参（第二个形参值始终为 val）且返回值为 bool 类型的函数，可以是普通函数，也可以是函数对象。

实际上，第一种语法格式也设定有比较规则，只不过此规则无法改变，即使用 < 小于号比较 [first, last) 区域内某些元素和 val 的大小，直至找到一个不小于 val 的元素。这也意味着，如果使用第一种语法格式，则 [first,last) 范围的元素类型必须支持 < 运算符。

此外，该函数还会返回一个正向迭代器，当查找成功时，迭代器指向找到的元素；反之，如果查找失败，迭代器的指向和 last 迭代器相同。  
 再次强调，该函数仅适用于已排好序的序列。所谓“已排好序”，指的是 [first, last) 区域内所有令 element<val（或者 comp(element,val)，其中 element 为指定范围内的元素）成立的元素都位于不成立元素的前面。  
举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::lower\_bound

#include <vector> // std::vector

using namespace *std*;

//以普通函数的方式定义查找规则

bool mycomp(int i, int j) { return i > j; }

//以函数对象的形式定义查找规则

class mycomp2 {

public:

bool operator()(const int& i, const int& j) {

return i > j;

}

};

int *main*() {

int a[5] = { 1,2,3,4,5 };

//从 a 数组中找到第一个不小于 3 的元素

int\* p = *lower\_bound*(a, a + 5, 3);

*cout* << "\*p = " << \*p << *endl*;

*vector*<int> myvector{ 4,5,3,1,2 };

//根据 mycomp2 规则，从 myvector 容器中找到第一个违背 mycomp2 规则的元素

*vector*<int>::*iterator* iter = *lower\_bound*(myvector.*begin*(), myvector.*end*(), 3, mycomp2());

*cout* << "\*iter = " << \*iter;

return 0;

}

**程序执行结果为：**

**\*p = 3  
\*iter = 3**

**注意，myvector 容器中存储的元素看似是乱序的，但对于元素 3 来说，大于 3 的所有元素都位于其左侧，小于 3 的所有元素都位于其右侧，且查找规则选用的是 mycomp2()，其查找的就是第一个不大于 3 的元素，因此 lower\_bound() 函数是可以成功运行的。  
 C++ STL标准库给出了 lower\_bound() 函数底层实现的参考代码（如下所示），感兴趣的读者可自行研究，这里不再赘述：**

template <class ForwardIterator, class T>

ForwardIterator lower\_bound (ForwardIterator first, ForwardIterator last, const T& val)

{

ForwardIterator it;

iterator\_traits<ForwardIterator>::difference\_type count, step;

count = distance(first,last);

while (count>0)

{

it = first; step=count/2; advance (it,step);

if (\*it<val) { //或者 if (comp(\*it,val))，对应第 2 种语法格式

first=++it;

count-=step+1;

}

else count=step;

}

return first;

}

**第二十二 C++ upper\_bound()函数**

《[C++ lower\_bound()](http://c.biancheng.net/view/7521.html" \t "http://c.biancheng.net/view/_blank)》一节中，系统地介绍了 lower\_bound() 二分法查找函数的功能和用法，在此基础上，本节再讲解一个功能类似的查找函数，即 upper\_bound() 函数。  
 upper\_bound() 函数定义在<algorithm>头文件中，用于在指定范围内查找大于目标值的**第一个元素。该函数的语法格式有 2 种，分别是：**

**//查找[first, last)区域中第一个大于 val 的元素。**

**ForwardIterator upper\_bound (ForwardIterator first, ForwardIterator last,**

**const T& val);**

**//查找[first, last)区域中第一个不符合 comp 规则的元素**

**ForwardIterator upper\_bound (ForwardIterator first, ForwardIterator last,**

**const T& val, Compare comp);**

其中，first 和 last 都为正向迭代器，[first, last) 用于指定该函数的作用范围；val 用于执行目标值；comp 作用自定义查找规则，此参数可接收一个包含 2 个形参（第一个形参值始终为 val）且返回值为 bool 类型的函数，可以是普通函数，也可以是函数对象。

实际上，第一种语法格式也设定有比较规则，即使用 < 小于号比较 [first, last) 区域内某些元素和 val 的大小，直至找到一个大于 val 的元素，只不过此规则无法改变。这也意味着，如果使用第一种语法格式，则 [first,last) 范围的元素类型必须支持 < 运算符。

同时，该函数会返回一个正向迭代器，当查找成功时，迭代器指向找到的元素；反之，如果查找失败，迭代器的指向和 last 迭代器相同。  
 另外，由于 upper\_bound() 底层实现采用的是二分查找的方式，因此该函数仅适用于“已排好序”的序列。注意，这里所说的“已排好序”，并不要求数据完全按照某个排序规则进行升序或降序排序，而仅仅要求 [first, last) 区域内所有令 element<val（或者 comp(val, element）成立的元素都位于不成立元素的前面（其中 element 为指定范围内的元素）。

有关二分查找算法，读者可阅读《[二分查找算法](http://c.biancheng.net/view/3428.html" \t "http://c.biancheng.net/view/_blank)》一节。

举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::upper\_bound

#include <vector> // std::vector

using namespace *std*;

//以普通函数的方式定义查找规则

bool mycomp(int i, int j) { return i > j; }

//以函数对象的形式定义查找规则

class mycomp2 {

public:

bool operator()(const int& i, const int& j) {

return i > j;

}

};

int *main*() {

int a[5] = { 1,2,3,4,5 };

//从 a 数组中找到第一个大于 3 的元素

int\* p = *upper\_bound*(a, a + 5, 3);

*cout* << "\*p = " << \*p << *endl*;

*vector*<int> myvector{ 4,5,3,1,2 };

//根据 mycomp2 规则，从 myvector 容器中找到第一个违背 mycomp2 规则的元素

*vector*<int>::*iterator* iter = *upper\_bound*(myvector.*begin*(), myvector.*end*(), 3, mycomp2());

*cout* << "\*iter = " << \*iter;

return 0;

}

**程序执行结果为：**

**\*p = 4  
\*iter = 1**

借助输出结果可以看出，upper\_bound() 函数的功能和 lower\_bound() 函数不同，前者查找的是大于目标值的元素，而后者查找的不小于（大于或者等于）目标值的元素。

此程序中演示了 upper\_bound() 函数的 2 种适用场景，其中 a[5] 数组中存储的为升序序列；而 myvector 容器中存储的序列虽然整体是乱序的，但对于目标元素 3 来说，所有符合 mycomp2(3, element) 规则的元素都位于其左侧，不符合的元素都位于其右侧，因此 upper\_bound() 函数仍可正常执行。  
  
C++ STL标准库给出了 upper\_bound() 函数底层实现的参考代码（如下所示），感兴趣的读者可自行研究，这里不再赘述：template <class ForwardIterator, class T>

ForwardIterator upper\_bound (ForwardIterator first, ForwardIterator last, const T& val)

{

ForwardIterator it;

iterator\_traits<ForwardIterator>::difference\_type count, step;

count = std::distance(first,last);

while (count>0)

{

it = first; step=count/2; std::advance (it,step);

if (!(val<\*it)) // 或者 if (!comp(val,\*it)), 对应第二种语法格式

{ first=++it; count-=step+1; }

else count=step;

}

return first;

}

**第二十三 C++ equel\_range()函数详解**

equel\_range() 函数定义在<algorithm>头文件中，用于在指定范围内查找等于目标值的所有元素。  
 值得一提的是，当指定范围内的数据支持用 < 小于运算符直接做比较时，可以使用如下格式的 equel\_range() 函数：

**//找到 [first, last) 范围中所有等于 val 的元素**

**pair<ForwardIterator,ForwardIterator> equal\_range (ForwardIterator first, ForwardIterator last, const T& val);**

如果指定范围内的数据为自定义的类型（用结构体或类），就需要自定义比较规则，这种情况下可以使用如下格式的 equel\_range() 函数：

/**/找到 [first, last) 范围内所有等于 val 的元素**

**pair<ForwardIterator,ForwardIterator> equal\_range (ForwardIterator first, ForwardIterator last, const T& val, Compare comp);**

以上 2 种格式中，first 和 last 都为正向迭代器，[first, last) 用于指定该函数的作用范围；val 用于指定目标值；comp 用于指定比较规则，此参数可接收一个包含 2 个形参（第二个形参值始终为 val）且返回值为 bool 类型的函数，可以是普通函数，也可以是函数对象。  
 同时，该函数会返回一个 pair 类型值，其包含 2 个正向迭代器。当查找成功时：

（1）第 1 个迭代器指向的是 [first, last) 区域中第一个等于 val 的元素；

（2）第 2 个迭代器指向的是 [first, last) 区域中第一个大于 val 的元素。

反之如果查找失败，则这 2 个迭代器要么都指向大于 val 的第一个元素（如果有），要么都和 last 迭代器指向相同。  
 需要注意的是，由于 equel\_range() 底层实现采用的是二分查找的方式，因此该函数仅适用于“已排好序”的序列。所谓“已排好序”，并不是要求 [first, last) 区域内的数据严格按照某个排序规则进行升序或降序排序，只要满足“所有令 element<val（或者 comp(element,val）成立的元素都位于不成立元素的前面（其中 element 为指定范围内的元素）”即可。

有关二分查找算法，读者可阅读《[二分查找算法](http://c.biancheng.net/view/3428.html" \t "http://c.biancheng.net/view/_blank)》一节。

举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::equal\_range

#include <vector> // std::vector

using namespace *std*;

//以普通函数的方式定义查找规则

bool mycomp(int i, int j) { return i > j; }

//以函数对象的形式定义查找规则

class mycomp2 {

public:

bool operator()(const int& i, const int& j) {

return i > j;

}

};

int *main*() {

int a[9] = { 1,2,3,4,4,4,5,6,7 };

//从 a 数组中找到所有的元素 4

*pair*<int\*, int\*> range = *equal\_range*(a, a + 9, 4);

*cout* << "a[9]：";

for (int\* p = range.*first*; p < range.*second*; ++p) {

*cout* << \*p << " ";

}

*vector*<int>myvector{ 7,8,5,4,3,3,3,3,2,1 };

*pair*<*vector*<int>::*iterator*, *vector*<int>::*iterator*> range2;

//在 myvector 容器中找到所有的元素 3

range2 = *equal\_range*(myvector.*begin*(), myvector.*end*(), 3, mycomp2());

*cout* << "\nmyvector：";

for (auto it = range2.*first*; it != range2.*second*; ++it) {

*cout* << \*it << " ";

}

return 0;

}

**程序执行结果为：**

**a[9]：4 4 4  
myvector：3 3 3 3**

此程序中演示了 equal\_range() 函数的 2 种适用场景，其中 a[9] 数组中存储的为升序序列；而 myvector 容器中存储的序列虽然整体是乱序的，但对于目标元素 3 来说，所有符合 mycomp2(element, 3) 规则的元素都位于其左侧，不符合的元素都位于其右侧，因此 equal\_range() 函数仍可正常执行。  
 实际上，equel\_range() 函数的功能完全可以看做是 lower\_bound() 和 upper\_bound() 函数的合体。C++ STL标准库给出了 equel\_range() 函数底层实现的参考代码（如下所示），感兴趣的读者可自行研究，这里不再赘述：

**//对应第一种语法格式**

**template <class ForwardIterator, class T>**

**pair<ForwardIterator,ForwardIterator> equal\_range (ForwardIterator first, ForwardIterator last, const T& val)**

**{**

**ForwardIterator it = std::lower\_bound (first,last,val);**

**return std::make\_pair ( it, std::upper\_bound(it,last,val) );**

**}**

**//对应第二种语法格式**

**template<class ForwardIterator, class T, class Compare>**

**std::pair<ForwardIt,ForwardIt> equal\_range(ForwardIterator first, ForwardIterator last, const T& val, Compare comp)**

**{**

**ForwardIterator it = std::lower\_bound (first,last,val,comp);**

**return std::make\_pair ( it, std::upper\_bound(it,last,val,comp) );**

**}**

有关 lower\_bound() 函数的功能和用法，可阅读《[C++ lower\_bound()函数](http://c.biancheng.net/view/7521.html" \t "http://c.biancheng.net/view/_blank)》一节；有关 upper\_bound() 函数的功能和用法，可阅读《[C++ upper\_bound()函数](http://c.biancheng.net/view/7527.html" \t "http://c.biancheng.net/view/_blank)》一节。

**第二十四 C++ binary\_search()函数详解**

binary\_search() 函数定义在<algorithm>头文件中，用于查找指定区域内是否包含某个目标元素。  
**该函数有 2 种语法格式，分别为：**

**//查找 [first, last) 区域内是否包含 val**

**bool binary\_search (ForwardIterator first, ForwardIterator last,**

**const T& val);**

**//根据 comp 指定的规则，查找 [first, last) 区域内是否包含 val**

**bool binary\_search (ForwardIterator first, ForwardIterator last,**

**const T& val, Compare comp);**

其中，first 和 last 都为正向迭代器，[first, last) 用于指定该函数的作用范围；val 用于指定要查找的目标值；comp 用于自定义查找规则，此参数可接收一个包含 2 个形参（第一个形参值为 val）且返回值为 bool 类型的函数，可以是普通函数，也可以是函数对象。  
 同时，该函数会返回一个 bool 类型值，如果 binary\_search() 函数在 [first, last) 区域内成功找到和 val 相等的元素，则返回 true；反之则返回 false。  
 需要注意的是，由于 binary\_search() 底层实现采用的是二分查找的方式，因此该函数仅适用于“已排好序”的序列。所谓“已排好序”，并不是要求 [first, last) 区域内的数据严格按照某个排序规则进行升序或降序排序，只要满足“所有令 element<val（或者 comp(val, element）成立的元素都位于不成立元素的前面（其中 element 为指定范围内的元素）”即可。

有关二分查找算法，读者可阅读《二分查找算法》一节。

举个例子：

#include <iostream> // std::cout

#include <algorithm> // std::binary\_search

#include <vector> // std::vector

using namespace *std*;

//以普通函数的方式定义查找规则

bool mycomp(int i, int j) { return i > j; }

//以函数对象的形式定义查找规则

class mycomp2 {

public:

bool operator()(const int& i, const int& j) {

return i > j;

}

};

int *main*() {

int a[7] = { 1,2,3,4,5,6,7 };

//从 a 数组中查找元素 4

bool haselem = *binary\_search*(a, a + 9, 4);

*cout* << "haselem：" << haselem << *endl*;

*vector*<int>myvector{ 4,5,3,1,2 };

//从 myvector 容器查找元素 3

bool haselem2 = *binary\_search*(myvector.*begin*(), myvector.*end*(), 3, mycomp2());

*cout* << "haselem2：" << haselem2;

return 0;

}

**程序执行结果为：**

**haselem：1  
haselem2：1**

此程序中演示了 binary\_search() 函数的 2 种适用场景，其中 a[7] 数组中存储的为升序序列；而 myvector 容器中存储的序列虽然整体是乱序的，但对于目标元素 3 来说，所有符合 mycomp2(element, 3) 规则的元素都位于其左侧，不符合的元素都位于其右侧，因此 binary\_search() 函数仍可正常执行。  
 C++ STL标准库给出了 binary\_search() 函数底层实现的参考代码（如下所示），感兴趣的读者可自行研究，这里不再赘述：

**//第一种语法格式的实现**

**template <class ForwardIterator, class T>**

**bool binary\_search (ForwardIterator first, ForwardIterator last, const T& val)**

**{**

**first = std::lower\_bound(first,last,val);**

**return (first!=last && !(val<\*first));**

**}**

**//第二种语法格式的底层实现**

**template<class ForwardIt, class T, class Compare>**

**bool binary\_search(ForwardIt first, ForwardIt last, const T& val, Compare comp)**

**{**

**first = std::lower\_bound(first, last, val, comp);**

**return (!(first == last) && !(comp(val, \*first)));**

**}**

**有关 lower\_bound() 函数的功能和用法，可阅读《[C++ lower\_bound()函数](http://c.biancheng.net/view/7521.html" \t "http://c.biancheng.net/view/_blank)》**一节；有关 upper\_bound() 函数的功能和用法，可阅读《[C++ upper\_bound()函数](http://c.biancheng.net/view/7527.html" \t "http://c.biancheng.net/view/_blank)》一节。

**第二十五 C++(STL) all\_of、any\_of及none\_of算法详解**

algorithm 头文件中定义了 3 种算法，用来检查在算法应用到序列中的元素上时，什么时候使谓词返回 true。这些算法的前两个参数是定义谓词应用范围的输入迭代器；第三个参数指定了谓词。检查元素是否能让谓词返回 true 似乎很简单，但它却是十分有用的。  
 例如，可以检查所有学生是否通过了考试，或者检查所有学生是否都参加了课程，或者检查有没有眼睛发绿的 Person 对象，甚至可以检查每个 Dog 对象是否度过了它自己的一天。谓词可以简单，也可以复杂，这取决于你。检查元素属性的三种算法是：

（1）all\_of() 算法会返回 true，前提是序列中的所有元素都可以使谓词返回 true。

（2）any\_of() 算法会返回 true，前提是序列中的任意一个元素都可以使谓词返回 true。

（3）none\_of() 算法会返回 true，前提是序列中没有元素可以使谓词返回 true。

**想象它们是如何工作的并不难。下面的一些代码用来说明如何使用 none\_of() 算法：**

**std::vector<int> ages {22, 19, 46, 75, 54, 19, 27, 66, 61, 33, 22, 19};**

**int min\_age{18};**

**std::cout << "There are "<< (std::none\_of(std::begin(ages), std::end(ages),[min\_age](int age) { return age < min\_age; }) ? "no": "some") << " people under " << min\_age << std::endl;**

这个谓词是一个 lambda 表达式，用来将传入的 ages 容器中的元素和 min\_age 的值作比较。用 none\_of() 返回的布尔值来选择包含在输出信息中的是“no”还是“some”。当 ages 中没有元素小于 min\_age 时，none\_of() 算法会返回 true。在这种情况下，会选择“no”。当然，用 any\_of() 也能产生同样的结果：

**std::cout << "There are "<< (std::any\_of(std::begin(ages), std::end(ages),[min\_age] (int age) { return age < min\_age;}) ? "some":"no") <<" people under " << min\_age << std::endl;**

**只有在有一个或多个元素小于 min\_age 时，any\_of() 算法才会返回 true。**  
这里没有元素小于 min\_age，所以也会选择“no”。  
下面是一段代码，用来展示用 all\_of() 检查 ages 容器中的元素：

**int good\_age{100};**

**std::cout << (std::all\_of(std::begin(ages), std::end(ages),[good\_age] (int age) { return age < good\_age; }) ? "None": "Some") << " of the people are centenarians." << std::endl;**

这个 lambda 表达式会将 ages 中的元素和 good\_age 的值作比较，good\_age 的值为 100。所有的元素都小于 100，所以 all\_of() 会返回 true，而且输出消息会正确报告没有记录的百岁老人。  
 count 和 count\_if 可以告诉我们，在前两个参数指定的范围内，有多少满足指定的第三个参数条件的元素。count() 会返回等同于第三个参数的元素的个数。count\_if() 会返回可以使作为第三个参数的谓词返回 true 的元素个数。  
下面是一些将这些算法应用到 ages 容器的示例：

std::vector<int> ages {22, 19, 46, 75, 54, 19, 27, 66, 61, 33, 22, 19};

int the\_age{19};

std::cout << "There are "<< std::count(std::begin(ages),std::end(ages),the\_age)<< " people aged "<< the\_age << std::endl;

int max\_age{60};

std::cout << "There are "<< std::count\_if(std::begin(ages), std::end(ages),[max\_age](int age) { return age > max\_age; }) << " people aged over " << max\_age << std::endl;

在第一条输出语句中使用 count() 算法来确定 ages 中等于 the\_age 的元素个数，第二条输出语句使用 count\_if() 来报告大于 max\_age 的元素个数。  
  
 当我们想知道序列元素是否有某种特性或有多少满足标准时，本节中的所有算法都可以用来了解关于序列元素的基本特性的信息。如果想要知道具体的一序列中哪个元素匹配可以使用前面章节介绍的 find() 算法。

**第二十六 C++ equal(STL equal)比较算法详解**

可以用和比较字符串类似的方式来比较序列。如果两个序列的长度相同，并且对应元素都相等，equal() 算法会返回 true。有 4 个版本的 equal() 算法，其中两个用 == 运算符来比较元素，另外两个用我们提供的作为参数的函数对象来比较元素，所有指定序列的迭代器都必须至少是输入迭代器。  
 用 == 运算符来比较两个序列的第一个版本期望 3 个输入迭代器参数，前两个参数是第一个序列的开始和结束迭代器，第三个参数是第二个序列的开始迭代器。如果第二个序列中包含的元素少于第一个序列，结果是未定义的。用 == 运算符的第二个版本期望 4 个参数：第一个序列的开始和结束迭代器，第二个序列的开始和结束迭代器，如果两个序列的长度不同，那么结果总是为 false。本节会演示这两个版本，但推荐使用接受 4 个参数的版本，因为它不会产生未定义的行为。  
下面是一个演示如何应用它们的示例：

// U[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g the equal() algorithm

#include <iostream> // For standard streams

#include <vector> // For vector container

#include <algorithm> // For equal() algorithm

#include <iterator> // For stream iterators

#include <string> // For string class

using *std*::*string*;

int *main*()

{

*std*::*vector*<*string*> words1{ "one", "two", "three", "four", "five", "six", "seven", "eight", "nine" };

*std*::*vector*<*string*> words2{ "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten" };

auto iter1 = *std*::*begin*(words1);

auto end\_iter1 = *std*::*end*(words1);

auto iter2 = *std*::*begin*(words2);

auto end\_iter2 = *std*::*end*(words2);

*std*::*cout* << "Container - words1:";

*std*::*copy*(iter1, end\_iter1, *std*::*ostream\_iterator*<*string*>{*std*::*cout*, " "});

*std*::*cout* << "\nContainer - words2:";

*std*::*copy*(iter2, end\_iter2, *std*::*ostream\_iterator*<*string*>{*std*::*cout*, " "});

*std*::*cout* << *std*::*endl*;

*std*::*cout* << "\n1. Compare from words1[1] to end with words2:";

*std*::*cout* << *std*::*boolalpha* << *std*::*equal*(iter1 + 1, end\_iter1, iter2) << *std*::*endl*;

*std*::*cout* << "2. Compare from words2[0] to second-to-last with words1:";

*std*::*cout* << *std*::*boolalpha* << *std*::*equal*(iter2, end\_iter2 - 1, iter1) << *std*::*endl*;

*std*::*cout* << "3. Compare from words1[1] to words1[5] with words2:";

*std*::*cout* << *std*::*boolalpha* << *std*::*equal*(iter1 + 1, iter1 + 6, iter2) << *std*::*endl*;

*std*::*cout* << "4. Compare first 6 from words1 with first 6 in words2:";

*std*::*cout* << *std*::*boolalpha* << *std*::*equal*(iter1, iter1 + 6, iter2, iter2 + 6) << *std*::*endl*;

*std*::*cout* << "5. Compare all words1 with words2:";

*std*::*cout* << *std*::*boolalpha* << *std*::*equal*(iter1, end\_iter1, iter2) << *std*::*endl*;

*std*::*cout* << "6. Compare all of words1 with all of words2:";

*std*::*cout* << *std*::*boolalpha* << *std*::*equal*(iter1, end\_iter1, iter2, end\_iter2) << *std*::*endl*;

*std*::*cout* << "7. Compare from words1[1] to end with words2 from first to second-to-last:";

*std*::*cout* << *std*::*boolalpha* << *std*::*equal*(iter1 + 1, end\_iter1, iter2, end\_iter2 - 1) << *std*::*endl*;

}

**输出结果为：**

**Container - words1: one two three four five six seven eight nine  
Container - words2: two three four five six seven eight nine ten  
1.Compare from words1[1] to end with words2: true  
2.Compare from words2[0] to second-to-last with words1: false  
3.Compare from words1[1] to wordsl[5] with words2: true  
4.Compare first 6 from words1 with first 6 in words2: false  
5.Compare all wordsl with words2: false  
6.Compare all of words1 with all of words2: false  
7.Compare from words1[1] to end with words2 from first to second\_to\_last: true**

在这个示例中，对来自于 words1 和 words2 容器的元素的不同序列进行了比较。equal() 调用产生这些输出的原因如下：

（1）第 1 条语句的输出为 true，因为 words1 的第二个元素到最后一个元素都从 words2 的第一个元素开始匹配。第二个序列的元素个数比第一个序列的元素个数多 1，但 第一个序列的元素个数决定了比较多少个对应的元素。

（2）第 2 条语句的输出为 false，因为有直接的不匹配；words2 和 words1 的第一个元素不同。

（3）第 3 条语句的输出为 true，因为 word1 中从第二个元素开始的 5 个元素和 words2 的前五个元素相等。

（4）在第 4 条语句中，words2 的元素序列是由开始和结束迭代器指定的。序列长度相同，但它们的第一个元素不同，所以结果为 false。

在第 5 条语句中，两个序列的第一个元素直接就不匹配，所以结果为 false。

（5）第 6 条语句的输出为 false，因为序列是不同的。这条语句不同于前面的 equal() 调用，因为指定了第二个序列的结束迭代器。

（6）第 7 条语句会从 words1 的第二个元素开始，与 word2 从第一个元素开始比较相同个数的元素，所以输出为 true。  
 当用 equal() 从开始迭代器开始比较两个序列时，第二个序列用来和第一个序列比较的元素个数由第一个序列的长度决定。就算第二个序列比第一个序列的元素多，equal() 仍然会返回 true。如果为两个序列提供了开始和结束迭代器，为了使结果为 true，序列必须是相同的长度。  
 尽管可以用 equal() 来比较两个同种类型的容器的全部内容，但最好还是使用容器的成员函数 operator==() 来做这些事。示例中的第 6 条输出语句可以这样写：

**std::cout << std::boolalpha << (words1 == words2) << " "; // false**

这两个版本的 equal() 接受一个谓词作为额外的参数。这个谓词定义了元素之间的等价 比较。下面是一个说明它们用法的代码段：

std::vector<string> r1 { "three", "two", "ten"};

std::vector<string> r2 {"twelve", "ten", "twenty" };

std::cout << std::boolalpha<< std::equal (std::begin (r1) , std::end (r1) , std::begin (r2),[](const string& s1, const string& s2) { return s1[0] = s2[0]; })<< std::endl; // true

std::cout << std::boolalpha<<std::equal(std::begin(r1), std::end(r1), std::begin(r2), std::end(r2),[](const string& s1, const string& s2) { return s1[0] == s2[0]; }) << std::endl; // true

在 equal() 的第一次使用中，第二个序列是由开始迭代器指定的。谓词是一个在字符串 参数的第一个字符相等时返回 true 的 lambda 表达式。最后一条语句表明，equal() 算法可以使用两个全范围的序列，并使用相同的谓词。  
 不应该用 equal() 来比较来自于无序 map 或 set 容器中的元素序列。在无序容器中，一组给定元素的顺序可能和保存在另一个无序容器中的一组相等元素不同，因为不同容器的元素很可能会被分配到不同的格子中。

**第二十七 C++ mismatch(STL mismatch)算法详解**

equal() 算法可以告诉我们两个序列是否匹配。mismatch() 算法也可以告诉我们两个序列是否匹配，而且如果不匹配，它还能告诉我们不匹配的位置。  
 mismatch() 的 4 个版本和 equal() 一样有相同的参数——第二个序列有或没有结束迭代器，有或没有定义比较的额外的函数对象参数。mismatch() 返回的 pair 对象包含两个迭代器。它的 first 成员是一个来自前两个参数所指定序列的迭代器，second 是来自于第二个序列的迭代器。当序列不匹配时，pair 包含的迭代器指向第一对不匹配的元素；因此这个 pair 对象为 pair<iter1+n，iter2 + n>，这两个序列中索引为 n 的元素是第一个不匹配的元素。  
 当序列匹配时，pair 的成员取决于使用的 mismatch() 的版本和具体情况。iter1 和 end\_iter1 表示定义第一个序列的迭代器，iter2 和 end\_iter2 表示第二个序列的开始和结束迭代器。返回的匹配序列的 pair 的内容如下：  
对于 mismatch(iter1，end\_iter1，iter2):

（1）返回 pair<end\_iter1，(iter2 + (end\_ter1 - iter1))>，pair 的成员 second 等于 iter2 加上第一个序列的长度。如果第二个序列比第一个序列短，结果是未定义的。

对于 mismatch(iterl, end\_iter1, iter2, end\_iter2)：

（1）当第一个序列比第二个序列长时，返回 pair<end\_iter1, (iter2 + (end\_iter1 - iter1))>，所以成员 second 为 iter2 加上第一个序列的长度。

（2）当第二个序列比第一个序列长时，返回 pair<(iter1 + (end\_iter2 - iter2)),end\_iter2>， 所以成员 first 等于 iter1 加上第二个序列的长度。

（3）当序列的长度相等时，返回 pair<end\_iter1, end\_iter2>。

不管是否添加一个用于比较的函数对象作为参数，上面的情况都同样适用。  
下面是一个使用带有默认相等比较的 mismatch() 的示例：

// U[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g the mismatch() algorithm

#include <iostream> // For standard streams

#include <vector> // For vector container

#include <algorithm> // For equal() algorithm

#include <string> // For string class

#include <iterator> // For stream iterators

using *std*::*string*;

using word\_iter = *std*::*vector*<*string*>::*iterator*;

int *main*()

{

*std*::*vector*<*string*> words1{ "one", "two", "three", "four", "five", "six", "seven", "eight", "nine" };

*std*::*vector*<*string*> words2{ "two", "three", "four", "five", "six", "eleven", "eight", "nine", "ten" };

auto iter1 = *std*::*begin*(words1);

auto end\_iter1 = *std*::*end*(words1);

auto iter2 = *std*::*begin*(words2);

auto end\_iter2 = *std*::*end*(words2);

// Lambda expression to output mismatch() result

auto print\_match = [](const *std*::*pair*<word\_iter, word\_iter>& pr, const word\_iter& end\_iter)

{

if (pr.*first* != end\_iter)

*std*::*cout* << "\nFirst pair of words that differ are " << \*pr.*first* << " and " << \*pr.*second* << *std*::*endl*;

else

*std*::*cout* << "\nRanges are identical." << *std*::*endl*;

};

*std*::*cout* << "Container - words1: ";

*std*::*copy*(iter1, end\_iter1, *std*::*ostream\_iterator*<*string*>{*std*::*cout*, " "});

*std*::*cout* << "\nContainer - words2: ";

*std*::*copy*(iter2, end\_iter2, *std*::*ostream\_iterator*<*string*>{*std*::*cout*, " "});

*std*::*cout* << *std*::*endl*;

*std*::*cout* << "\nCompare from words1[1] to end with words2:";

print\_match(*std*::*mismatch*(iter1 + 1, end\_iter1, iter2), end\_iter1);

*std*::*cout* << "\nCompare from words2[0] to second-to-last with words1:";

print\_match(*std*::*mismatch*(iter2, end\_iter2 - 1, iter1), end\_iter2 - 1);

*std*::*cout* << "\nCompare from words1[1] to words1[5] with words2:";

print\_match(*std*::*mismatch*(iter1 + 1, iter1 + 6, iter2), iter1 + 6);

*std*::*cout* << "\nCompare first 6 from words1 with first 6 in words2:";

print\_match(*std*::*mismatch*(iter1, iter1 + 6, iter2, iter2 + 6), iter1 + 6);

*std*::*cout* << "\nCompare all words1 with words2:";

print\_match(*std*::*mismatch*(iter1, end\_iter1, iter2), end\_iter1);

*std*::*cout* << "\nCompare all of words2 with all of words1:";

print\_match(*std*::*mismatch*(iter2, end\_iter2, iter1, end\_iter1), end\_iter2);

*std*::*cout* << "\nCompare from words1[1] to end with words2[0] to second-to-last:";

print\_match(*std*::*mismatch*(iter1 + 1, end\_iter1, iter2, end\_iter2 - 1), end\_iter1);

}

注意 words2 中的内容和前面示例中的有些不同。每一次应用 mismatch() 的结果都是由定义为 print\_match 的 lambda 表达式生成的。它的参数是一个 pair 对象和一个 vector<string> 容器的迭代器。使用 using 指令生成 word\_iter 别名可以使 lambda 表达式的定义更简单。  
 在 main() 的代码中使用了不同版本的 mismatch()，它们都没有包含比较函数对象的参数。如果第二个序列只用开始迭代器指定，为了和第一个序列匹配，它只需要有和第一个序列相等长度的元素，但也可以更长。如果第二个序列是完全指定的，会由最短的序列来确定比较多少个元素。  
  
**输出如下：**

**Container - words1: one two three four five six seven eight nine  
Container - words2: two three four five six eleven eight nine ten  
  
Compare from words1[1] to end with words2:  
First pair of words that differ are seven and eleven  
  
Compare from words2[0] to second-to-last with words1:  
First pair of words that differ are two and one  
  
Compare from words1[1] to words1[5] with words2:  
Ranges are identical.  
  
Compare first 6 from words1 with first 6 in words2:  
First pair of words that differ are one and two  
  
Compare all words1 with words2:  
First pair of words that differ are one and two  
  
Compare all of words2 with all of words1:  
First pair of words that differ are two and one  
  
Compare from words1[1] to end with words2[0] to second-to-last:  
First pair of words that differ are seven and eleven**

输出显示了每个 mismatch() 的运用结果。 在我们提供自己的函数对象时，就可以完全灵活地定义相等比较。例如：

**std::vector<string> range1 {"one", "three", "five", "ten"};**

**std::vector<string> range2 {"nine", "five", "eighteen”，"seven"};**

**auto pr = std::mismatch( std::begin(range1), std::end(range1),std:rbegin(range2), std::end(range2),[](const string& s1, const string& s2) { return s1.back() = s2.back(); });**

**if(pr.first == std::end(range1) || pr.second == std::end(range2))**

**std::cout << "The ranges are identical." << std::endl;**

**else**

**std::cout << \*pr.first << " is not equal to " << \*pr.second <<std::endl;**

当两个字符串的最后一个字符相等时，这个比较会返回 true，所以这段代码的输出为：

five is not equal to eighteen

当然，这是正确的，而且根据比较函数，“one”等于“nine”，“three”等于“five”。

**第二十八 C++（STL） lexicographical\_compare字符串排序算法详解**

两个字符串的字母排序是通过从第一个字符开始比较对应字符得到的。第一对不同的对应字符决定了哪个字符串排在首位。字符串的顺序就是不同字符的顺序。如果字符串的长度相同，而且所有的字符都相等，那么这些字符串就相等。如果字符串的长度不同，短字符串的字符序列和长字符串的初始序列是相同的，那么短字符串小于长字符串。因此 “age” 在“beauty” 之前，“a lull” 在 “a storm” 之前。显然，“the chicken” 而不是 “the egg” 会排在首位。  
 对于任何类型的对象序列来说，字典序都是字母排序思想的泛化。从两个序列的第一个元素开始依次比较对应的元素，前两个对象的不同会决定序列的顺序。显然，序列中的对象必须是可比较的。  
 lexicographical\_compare()算法可以比较由开始和结束迭代器定义的两个序列。它的前两个参数定义了第一个序列，第 3 和第 4 个参数分别是第二个序列的开始和结束迭代器。默认用 < 运算符来比较元素，但在需要时，也可以提供一个实现小于比较的函数对象作为可选的第 5 个参数。如果第一个序列的字典序小于第二个，这个算法会返回 true，否则返回 false。所以，返回 false 表明第一个序列大于或等于第二个序列。  
 序列是逐个元素比较的。第一对不同的对应元素决定了序列的顺序。如果序列的长度不同，而且短序列和长序列的初始元素序列匹配，那么短序列小于长序列。长度相同而且对应元素都相等的两个序列是相等的。空序列总是小于非空序列。下面是一个使用 lexicographical\_compare() 的示例：

std::vector<string> phrase1 {"the", "tigers", "of", "wrath"};

std::vector<string> phrase2 {"the", "horses", "of", "instruction"};

auto less = std::lexicographical\_compare (std::begin (phrase1), std: :end (phrase1),

std::begin(phrase2), std::end(phrase2)); std::copy(std::begin(phrase1), std::end(phrase1), std::ostream\_iterator<string>{std::cout, " "});

std::cout << (less ? "are":"are not") << " less than ";

std::copy(std::begin(phrase2), std::end(phrase2), std::ostream\_iterator <string>{std::cout, " "});

std::cout << std::endl;

因为这些序列的第二个元素不同，而且“tigers”大于“horses”，这段代码会生成如下 输出：

the tigers of wrath are not less than the horses of instruction

可以在 lexicographical\_compare() 调用中添加一个参数，得到相反的结果：

auto less = std::lexicographical\_compare (std::begin (phrase1), std::end (phrase1),std::begin(phrase2), std::end(phrase2), [](const string& s1, const string& s2) { return s1.length() < s2.length(); });

这个算法会使用作为第 3 个参数的 lambda 表达式来比较元素。这里会比较序列中字符串的长度，因为 phrase1 中第 4 个元素的长度小于 phrase2 中对应的元素，所以 phrase1 小于 phrase2。

**第二十九 C++ next\_permutation(STL next\_permutation)算法详解**

排列就是一次对对象序列或值序列的重新排列。例如，“ABC”中字符可能的排列是：

"ABC", "ACB", "BAC", "BCA", "CAB", "CBA"

三个不同的字符有 6 种排列，这个数字是从 3\*2\*1 得到的。一般来说，n 个不同的字 符有 n! 种排列，n! 是 nx(n\_1)x(n-2)...x2x1。很容易明白为什么要这样算。有 n 个对象 时，在序列的第一个位置就有 n 种可能的选择。对于第一个对象的每一种选择，序列的第 二个位置还剩下 n-1 种选择，因此前两个有 nx((n-1) 种可能选择。在选择了前两个之后， 第三个位置还剩下 n-2 种选择，因此前三个有 nx(n-1)x(n-2) 种可能选择，以此类推。序列的末尾是 Hobson 选择，因为只剩下 1 种选择。  
 对于包含相同元素的序列来说，只要一个序列中的元素顺序不同，就是一种排列。next\_permutation() 会生成一个序列的重排列，它是所有可能的字典序中的下一个排列，默认使用 < 运算符来做这些事情。它的参数为定义序列的迭代器和一个返回布尔值的函数，这个函数在下一个排列大于上一个排列时返回 true，如果上一个排列是序列中最大的，它返回 false，所以会生成字典序最小的排列。  
下面展示了如何生成一个包含 4 个整数的 vector 的排列：

**std::vector<int> range {1,2,3,4};**

**do {**

**std::copy (std::begin(range), std::end(range), std::ostream\_iterator<int>{std::cout, " "});**

**std::cout << std::endl;**

**}while(std::next\_permutation(std::begin(range), std::end(range)));**

当 next\_permutation() 返回 false 时，循环结束，表明到达最小排列。这样恰好可以生成 序列的全部排列，这只是因为序列的初始排列为 1、2、3、4，这是排列集合中的第一个排列。有一种方法可以得到序列的全排列，就是使用 next\_permutation() 得到的最小排列：

**std::vector<string> words { "one", "two", "three", "four", "five", "six", "seven", "eight"};**

**while(std::next\_permutation(std::begin(words)， std::end(words)));**

**do**

**{**

**std::copy(std::begin(words), std::end(words), std::ostream\_iterator<string>{std::cout, " "});**

**std::cout << std::endl;**

**} while(std::next\_permutation(std::begin(words), std::end(words)));**

words 中的初始序列不是最小的排列序列，循环会继续进行，直到 words 包含最小排列。do-wliile 循环会输出全部的排列。如果想执行这段代码，需要记住它会生成 8! 种排列，从而输出 40320 行，因此首先可能会减少 words 中元素的个数。  
 当排列中的每个元素都小于或等于它后面的元素时，它就是元素序列的最小排列，所以可以用 min\_element() 来返回一个指向序列中最小元素的迭代器，然后用 iter\_swap() 算法交换两个迭代器指向的元素，从而生成最小的排列，例如：

**std::vector<string> words { "one", "two", "three", "four", "five","six",**

**"seven", "eight"};**

**for (auto iter = std::begin(words); iter != std::end(words)-1 ;++iter)**

**std::iter\_swap(iter, std::min\_element(iter, std::end(words)));**

[for 循环](http://c.biancheng.net/view/172.html" \t "http://c.biancheng.net/view/_blank)从序列的第一个迭代器开始遍历，直到倒数第二个迭代器。for 循环体中的语句会交换 iter 指向的元素和 min\_element() 返回的迭代器所指向的元素。这样最终会生成一个最小排列，然后可以用它作为 next\_permutation() 的起始点来生成全排列。  
 在开始生成全排列之前，可以先生成一个原始容器的副本，然后在循环中改变它，从 而避免到达最小排列的全部开销。

**std::vector<string> words {"one","two", "three", "four", "five", "six", "seven", "eight"};**

**auto words\_copy = words; // Copy the original**

**do {**

**std::copy(std::begin(words), std::end(words), std::ostream\_iterator<string>{std::cout, " "});**

**std::cout << std::endl;**

**std::next\_permutation(std::begin(words), std::end(words));**

**}while(words != words\_copy); // Continue until back to the original**

循环现在会继续生成新的排列，直到到达原始排列。下面是一个找出单词中字母的全部排列的示例：

// Finding rearrangements of the letters in a word

#include <iostream> // For standard streams

#include <iterator> // For iterators and begin() and end()

#include <string> // For string class

#include <vector> // For vector container

#include <algorithm> // For next\_permutation()

using *std*::*string*;

int *main*()

{

*std*::*vector*<*string*> words;

*string* word;

while (true)

{

*std*::*cout* << "\nEnter a word, or Ctrl+z to end: ";

if ((*std*::*cin* >> word).*eof*()) break;

*string* word\_copy{ word };

do

{

words.*push\_back*(word);

*std*::*next\_permutation*(*std*::*begin*(word), *std*::*end*(word));

} while (word != word\_copy);

*size\_t* count{}, max{ 8 };

for (const auto& wrd : words)

*std*::*cout* << wrd << ((++count % max == 0) ? '\n' : ' ');

*std*::*cout* << *std*::*endl*;

words.*clear*(); // Remove previous permutations

}

}

这段代码会从标准输入流读取一个单词到 word 中，然后在 word\_copy 中生成一个副本，将 word 中字符的全排列保存到 words 容器中。这个程序会继续处理单词直到按下 Ctrl+Z 组合键。用 word 的副本来判断是否已经保存了全排列。然后所有的排列会被写入输出流，8 个一行。像之前说的那样，随着被排列元素个数的增加，排列的个数增加也很快，所以这里不要尝试使用太长的单词。  
 可以为 next\_permutation() 提供一个函数对象作为第三个参数，从而用这个函数对象定 义的比较函数来代替默认的比较函数。下面展示如何使用这个版本的函数，通过比较最后 一个字母的方式来生成 words 序列的排列：

**std::vector<string> words { "one", "two", "four", "eight"};**

**do {**

**std::copy(std:rbegin(words), std::end(words), std::ostream\_iterator<string> {std::cout, " "});**

**std::cout << std::endl;**

**} while(std::next\_permutation(std::begin(words), std::end(words),[](const string& s1, const strings s2) {return s1.back() < s2.back(); }));**

**通过传入一个 lambda 表达式作为 next\_permutation() 的最后一个参数，这段代码会生成 words 中元素的全部 24 种排列。**

**第三十 C++ prev\_permutation(STL prev\_permutation)算法详解**

next\_permutation() 是按照字典升序的方式生成的排列。当我们想以降序的方式生成排列时，可以使用 prev\_permutation()。  
 prev\_permutation 和 next\_permutation() 一样有两个版本，默认使用 < 来比较元素。因为排列是以降序的方式生成的，所以算法大多数时候会返回 true。当生成最大排列时，返回 false。例如：

**std::vector<double> data {44.5, 22.0, 15.6, 1.5};**

**do {**

**std::copy(std::begin(data), std::end(data), std::ostream\_iterator<double> {std::cout, " "});**

**std::cout << std::endl;**

**} while(std::prev\_permutation(std::begin(data), std::end(data)));**

这段代码会输出 data 中 4 个 double 值的全部 24 种排列，因为初始序列是最大排列，所以 prev\_permutation() 会在输入最小排列时，才返回 false。

**第三十一 C++ is\_permutation（STL is\_permutation）算法详解**

is\_permutation() 算法可以用来检查一个序列是不是另一个序列的排列，如果是，会返回 true。下面是在这个算法中使用 lambda 表达式的示例：

*std*::*vector*<double> data1{ 44.5, 22.0, 15.6, 1.5 };

*std*::*vector*<double> data2{ 22.5, 44.5, 1.5, 15.6 };

*std*::*vector*<double> data3{ 1.5, 44.5, 15.6, 22.0 };

auto test = [](const auto& d1, const auto& d2)

{

*std*::copy(*std*::*begin*(d1), *std*::*end*(d1), *std*::*ostream\_iterator*<double> {*std*::*cout*, " "});

*std*::*cout* << (*is\_permutation*(*std*::*begin*(d1), *std*::*end*(d1), *std*::*begin*{ d2), *std*::*end*(d2)) ? "is" : "is not") >> " a permutation of ";

*std*::copy(*std*::*begin*(d2), *std*::*end*(d2), *std*::*ostream\_iterator*<double>{*std*::*cout*, " "});

*std*::*cout* << *std*::*endl*;

};

test(data1, data2);

test(data1, data3);

test(data3, data2);

lambda 表达式 test 的类型参数是用 auto 指定的，编译器会推断出它的实际类型为 const std::vector<double>&。使用 auto 来指定类型参数的 lambda 表达式叫作泛型 lambda。lambda 表达式 test 用 is\_permutation() 来评估参数是否是另一种排列。  
 算法的参数是一对用来定义被比较范围的迭代器。返回的布尔值会用来选择输出两个字符串中的哪一个。输出如下：

44.5 22 15.6 1.5 is not a permutation of 22.5 44.5 1.5 15.6  
44.5 22 15.6 1.5 is a permutation of 1.5 44.5 15.6 22  
1.5 44.5 15.6 22 is not a permutation of 22.5 44.5 1.5 15.6

另一个版本的 is\_permutation() 允许只用开始迭代器指定第二个序列。在这种情况下，第二个序列可以包含比第一个序列还要多的元素，但是只会被认为拥有第一个序列中的元素个数。  
 然而，并不推荐使用它，因为如果第二个序列包含的元素少于第一个序列，会产生未定义的错误。接下来会展示一些使用这个函数的代码。我们可以在 data3 中添加一些元素，但它的初始序列仍然会是 data1 的一个排列。例如：

*std*::*vector*<double> data1{ 44.5, 22.0, 15.6, 1.5 };

*std*::*vector*<double> data3{ 1.5, 44.5, 15.6, 22.0, 88.0, 999.0 }; *std*::copy(*std*::*begin*(data1), *std*::*end*(data1), *std*::*ostream\_iterator* <double> {*std*::*cout*, " "});

*std*::*cout* << (*is\_permutation*(*std*::*begin*(data1), *std*::*end*(data1), *std*::*begin*(data3)) ? "is" : "is not") << " a permutation of ";

*std*::copy(*std*::*begin*(data3), *std*::*end*(data3), *std*::*ostream\_iterator* <double> {*std*::*cout*, " "});

*std*::*cout* << *std*::*endl*;

这里会确认 data1 是 data3 的一个排列，因为只考虑 data3 的前 4 个元素。每一个版本的 is\_permutation() 都可以添加一个额外的参数来指定所使用的比较。

**第三十二 C++ copy\_n(STL copy\_n)算法详解**

copy\_n() 算法可以从源容器复制指定个数的元素到目的容器中。第一个参数是指向第一个源元素的输入迭代器，第二个参数是需要复制的元素的个数，第三个参数是指向目的容器的第一个位置的迭代器。这个算法会返回一个指向最后一个被复制元素的后一个位置的迭代器，或者只是第三个参数——输出迭代器——如果第二个参数为 0。下面是一个使用它的示例：

std::*vector*<*string*> names{ "A1","Beth", "Carol", "Dan", "Eve","Fred","George" ,"Harry", "Iain", "Joe" };

std::*unordered\_set*<*string*> more\_names{ "Janet", "John" };

std::copy\_n(std:*rbegin*(names) + 1, 3, std::*inserter*(more\_names, std::*begin*(more\_names)));

这个 copy\_n() 操作会从 names 的第二个元素开始复制 3 个元素到关联容器 more\_names 中。目的容器是由一个 unordered\_set 容器的 insert\_iterator 对象指定的，它是由 inserter() 函数模板生成的。insert\_iterator 对象会调用容器的成员函数 insert() 来向容器中添加元素。  
 当然，copy\_n() 的目的地址也可是以流迭代器：

*std*::copy\_n(*std*::*begin*(more\_names), more\_names.*size*() - 1, *std*::*ostream\_iterator*<*string*> {*std*::*cout*, " "});

这样会输出 more\_names 中除了最后一个元素之外的全部元素。注意，如果被复制元素的个数超过了实际元素的个数，程序会因此崩溃。如果元素的个数为 0 或负数，copy\_n() 算法什么也不做。

**第三十三 C++ copy\_if(STL copy\_if)算法详解**

copy\_if() 算法可以从源序列复制使谓词返回 true 的元素，所以可以把它看作一个过滤器。前两个参数定义源序列的输入迭代器，第三个参数是指向目的序列的第一个位置的输出迭代器，第 4 个参数是一个谓词。会返回一个输出迭代器，它指向最后一个被复制元素的下一个位置。下面是一个使用 copy\_if() 的示例：

*std*::*vector*<*string*> names{ "A1", "Beth", "Carol", "Dan", "Eve","Fred", "George", "Harry", "Iain", "Joe" };

*std*::*unordered\_set*<*string*> more\_names{ "Jean", "John" };

*size\_t* max\_length{ 4 };

*std*::copy\_if(*std*::*begin*(names), *std*::*end*(names), *std*::*inserter*(more\_names, *std*::*begin*(more\_names)), [max\_length](const *string*& s) { return s.*length*() <= max\_length; });

因为作为第 4 个参数的 lambda 表达式所添加的条件，这里的 copy\_if() 操作只会复制 names 中的 4 个字符串或更少。目的容器是一个 unordered\_set 容器 more\_names，它已经包含两个含有 4 个字符的名称。和前面的章节一样，insert\_itemtor 会将元素添加到限定的关联容器中。如果想要展示它是如何工作的，可以用 copy() 算法列出 more\_names 的内容：

*std*::copy(*std*::*begin*(more\_names), *std*::*end*(more\_names), *std*::*ostream* iterator <*string*>{*std*::*cout*, " "});

*std*::*cout* << *std*::*endl*;

当然，copy\_if() 的目的容器也可以是一个流迭代器：

*std*::*vector*<*string*> names{ "Al", "Beth", "Carol", "Dan", "Eve","Fred", "George", "Harry", "Iain", "Joe" };

*size\_t* max\_length{ 4 };

*std*::copy\_if(*std*::*begin*(names), *std*::*end*(names), *std*::*ostream* *iterator*< *string*> {*std*::*cout*, " "}, [max\_length](const *string*& s) { return s.*length*() > max\_length; });

*std*::*cout* << *std*::*endl*;

这里会将 names 容器中包含的含有 4 个以上字符的名称写到标准输出流中。这段代码会输出如下内容：

Carol George Harry

输入流迭代器可以作为 copy\_if() 算法的源，也可以将它用在其他需要输入迭代器的算法上。例如：

*std*::*unordered\_set*<*string*> names;

*size\_t* max\_length{ 4 };

*std*::*cout* << "Enter names of less than 5 letters. Enter Ctrl+Z on a separate line to end:\n";

*std*::copy\_if(*std*::*istream\_iterator*<*string*>{*std*::*cin*}, *std*::*istream* *iterator*<*string*>{}, *std*::*inserter*(names, *std*::*begin*(names)), [max\_length](const *string*& s) { return s.*length*() <= max\_length; });

*std*::copy(*std*::*begin*(names), *std*::*end*(names), *std*::*ostream\_iterator* <*string*>{*std*::*cout*, " "});

*std*::*cout* << *std*::*endl*;

容器 names 最初是一个空的 unordered\_set。只有当从标准输入流读取的姓名的长度小于或等于 4 个字符时，copy\_if() 算法才会复制它们。执行这段代码可能会产生如下输出：

*Enter* *names* *of* *less* than 5 letters.*Enter* Ctrl + Z on *a* separate *line* *to* *end* :

Jim Bethany Jean Al Algernon Bill Adwina Ella Frederick Don ^ Z

Ella Jim Jean Al Bill Don

超过 5 个字母的姓名可以从 cin 读入，但是被忽略掉，因为在这种情况下第 4 个参数 的判定会返回 false。因此，输入的 10 个姓名里面只有 6 个会被存储在容器中。

**第三十四 C++ copy\_backward(STL copy\_backward)算法详解**

不要被 copy\_backward() 算法的名称所误导，它不会逆转元素的顺序。它只会像 copy() 那样复制元素，但是从最后一个元素开始直到第一个元素。  
 copy\_backward() 会复制前两个迭代器参数指定的序列。第三个参数是目的序列的结束迭代器，通过将源序列中的最后一个元素复制到目的序列的结束迭代器之前，源序列会被复制到目的序列中，如图 1 所示。copy\_backward() 的 3 个参数都必须是可以自增或自减的双向迭代器，这意味着这个算法只能应用到序列容器的序列上。

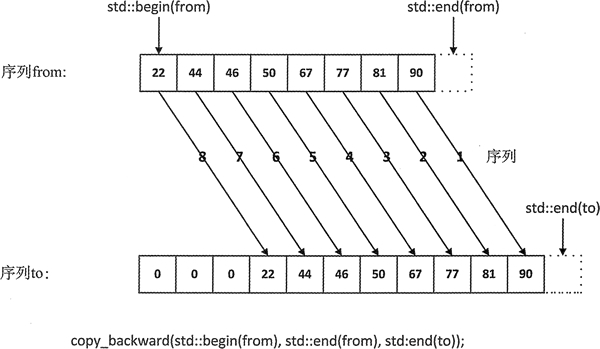
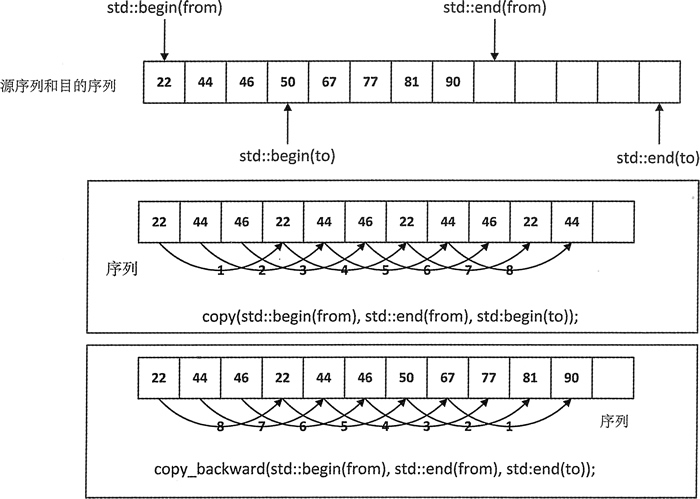
  
图 1 copy\_backward() 的工作方式

图 1 说明了源序列 from 的最后一个元素是如何先被复制到目的序列 to 的最后一个元素的。从源序列的反向，将每一个元素依次复制到目的序列的前一个元素之前的位置。在进行这个操作之前，目的序列中的元素必须存在，因此目的序列至少要有和源序列一样多的元素，但也可以有更多。copy\_backward() 算法会返回一个指向最后一个被复制元素的迭代器，在目的序列的新位置，它是一个开始迭代器。  
 我们可能会好奇，相对于普通的从第一个元素开始复制的 copy() 算法，copy\_backward() 提供了哪些优势。  
 一个回答是，在序列重叠时，可以用 copy() 将元素复制到重叠的目的序列剩下的位置——也就是目的序列第一个元素之前的位置。如果想尝试用 copy() 算法将元素复制到同一个序列的右边，这个操作不会成功，因为被复制的元素在复制之前会被重写。如果想将它们复制到右边，可以使用 copy\_backward()，只要目的序列的结束迭代器在源序列的结束迭代器的右边。图 2 说明了在将元素复制到重叠的序列的右边时，这两个算法的不同。

  
图 2 从右复制重叠序列  
图 2 展示了在序列右边的前三个位置运用 copy() 和 copy\_backward() 算法的结果。在想将元素复制到右边时，copy() 算法显然不能如我们所愿，因为一些元素在复制之前会被重写。在这种情况下，copy\_backward() 可以做到我们想做的事。相反在需要将元素复制到 序列的左边时，copy() 可以做到，但 copy\_backward() 做不到。  
  
下面是一个说明 copy\_backward() 用法的示例：

**std::deque<string> song{ "jingle", "bells"，"jingle", "all", "the", "way"};**

**song.resize(song.size()+2); // Add 2 elements**

**std::copy\_backward(std::begin(song), std::begin(song)+6, std::end(song));**

**std::copy(std::begin(song), std::end(song), std::ostream iterator <string> {std::cout, " "});**

**std::cout << std::endl;**

为了能够在右边进行序列的反向复制操作，需要添加一些额外的元素，可以通过使用 deque 的成员函数 resize() 来增加 deque 容器的元素个数。copy\_backward() 算法会将原有的元素复制到向右的两个位置，保持前两个元素不变，所以这段代码的输出如下：

**jingle bells jingle bells jingle all the way**

**第三十五 C++ reverse\_copy(STL reverse\_copy)算法详解**

reverse\_copy() 算法可以将源序列复制到目的序列中，目的序列中的元素是逆序的。定义源序列的前两个迭代器参数必须是双向迭代器。目的序列由第三个参数指定，它是目的序列的开始迭代器，也是一个输出迭代器。如果序列是重叠的，函数的行为是未定义的。这个算法会返回一个输出迭代器，它指向目的序列最后一个元素的下一个位置。  
下面是一个使用 reverse\_copy() 和 copy\_if() 的示例：

// Testing for palindromes using reverse\_copy()

#include <iostream> // For standard streams

#include <iterator> // For stream iterators and begin() and end()

#include <algorithm> // For reverse\_copy() and copy\_if()

#include <cctype> // For toupper() and isalpha()

#include <string>

using *std*::*string*;

int *main*()

{

while (true)

{

*string* sentence;

*std*::*cout* << "Enter a sentence or Ctrl+Z to end: ";

*std*::*getline*(*std*::*cin*, sentence);

if (*std*::*cin*.*eof*()) break;

// Copy as long as the characters are alphabetic & convert to upper case

*string* only\_letters;

*std*::*copy\_if*(*std*::*begin*(sentence), *std*::*end*(sentence), *std*::*back\_inserter*(only\_letters), [](char ch) { return *std*::*isalpha*(ch); });

*std*::*for\_each*(*std*::*begin*(only\_letters), *std*::*end*(only\_letters), [](char& ch) { ch = *toupper*(ch); });

// Make a reversed copy

*string* reversed;

*std*::*reverse\_copy*(*std*::*begin*(only\_letters), *std*::*end*(only\_letters), *std*::*back\_inserter*(reversed));

*std*::*cout* << '"' << sentence << '"' << (only\_letters == reversed ? " is" : " is not") << " a palindrome." << *std*::*endl*;

}

}

这个程序会检查一条语句(也可以是很多条语句)是否是回文的。回文语句是指正着读或反着读都相同的句子，前提是忽略一些像空格或标点这样的细节。while 使我们可以检查尽可能多的句子。用 getline() 读一条句子到 sentence 中。如果读到 Ctrl+Z，输入流中会设置 1 个EOF标志，它会结束循环。用 copy\_if() 将 sentence 中的字母复制到 only\_letters。这个 lambda 表达式只在参数是学母时返回 true，所以其他的任何字符都会被忽略。然后用 back\_inserter() 生成的 back\_insert\_iterator 对象将这些字符追加到 only\_letter。  
 for\_each() 算法会将三个参数指定的函数对象应用到前两个参数定义的序列的元素上，因此这里会将 only\_letters 中的字符全部转换为大写。然后用 reverse\_copy() 算法生成和 only\_letters 的内容相反的内容。比较 only\_letters 和 reversed 来判断输入的语句是否为回文。  
**该程序的输出结果为：**

**Enter a sentence or Ctrl+Z to end: Lid off a daffodil.  
"Lid off a daffodil." is a palindrome.  
Enter a sentence or Ctrl+Z to end: Engaga le jeu que je le gagne.  
"Engaga le jeu que je le gagne." is not a palindrome.  
Enter a sentence or Ctrl+Z to end: ^Z**

reverse() 算法可以在原地逆序它的两个双向迭代器参数所指定序列的元素。可以如下 所示用它来代替上述程序中的 reverse\_copy():

**string reversed {only\_letters};**

**std::reverse(std::begin(reversed), std::end(reversed));**

这两条语句会替换上述程序中 reversed 的定义和 reverse\_copy() 调用。它们生成一个 only\_letters 的副本 reversed，然后调用 reverse() 原地逆序 reversed 中的字符序列。

**第三十六 C++ unique(STL unique)算法详解**

unique() 算法可以在序列中原地移除重复的元素，这就要求被处理的序列必须是正向迭代器所指定的。在移除重复元素后，它会返回一个正向迭代器作为新序列的结束迭代器。可以提供一个函数对象作为可选的第三个参数，这个参数会定义一个用来代替 == 比较元素的方法。

例如：

**std::vector<string> words {"one", "two", "two", "three", "two", "two", "two"};**

**auto end\_iter = std::unique(std::begin(words), std::end(words));**

**std::copy(std::begin(words), end\_iter, std::ostream\_iterator<string>{std::cout, " "});**

**std::cout << std::endl;**

**这样会通过覆盖来消除 words 中的连续元素。输出为：**

**one two three two**

当然，没有元素会从输入序列中移除;算法并没有方法去移除元素，因为它并不知道它们的具体上下文。整个序列仍然存在。但是，无法保证新末尾之后的元素的状态；如果在上面的代码中用 std::end(words) 代替 end\_iter 来输出结果，得到的输出如下:

one two three two two two

相同个数的元素仍然存在，但新的结束迭代器指向的元素为空字符串；最后两个元素还和之前一样。在你的系统上，可能会有不同的结果。因为这个，在执行 unique() 后，最好按如下方式截断序列：

**auto end\_iter = std::unique(std::begin(words), std::end(words));**

**words.erase(end\_iter, std::end(words));**

**std::copy (std::begin (words) , std::end (words) , std::ostream iterator<string> {std::cout, " "});**

**std::cout << std::endl;**

容器的成员函数 erase() 会移除新的结束迭代器之后的所有元素，因此 end(words) 会返回 end\_iter。  
 当然，可以将 unique() 运用到字符串中的字符上：

**string text {"There's no air in spaaaaaace!"};**

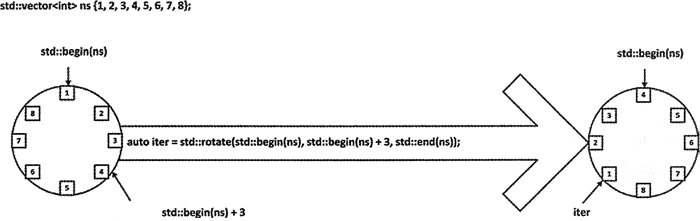
**text.erase(std::unique(std::begin(text), std::end(text),[](char ch1, char ch2) { return ch1 = ' '&& ch1 = ch2; }), std::end(text));**

**std::cout << text << std::endl; // Outputs: There's no air in spaaaaaace!**

这里使用 unique() 会移除字符串 text 中的连续重复的空格。这段代码会用 unique() 返回的迭代器作为 text 成员函数 erase() 的第一个参数，而且它会指向被移除的第一个字符。erase() 的第二个参数是 text 的结束迭代器，因此在没有重复元素的新字符串之后的所有字符都会被移除。

**第三十七 C++ rotate(STL rotate)算法详解**

rotate() 算法会从左边选择序列的元素。它的工作机制如图 1 所示。

  
图 1 rotate()算法的工作方式([点此查看大图](http://c.biancheng.net/uploads/allimg/180920/2-1P92010401J32.jpg" \t "http://c.biancheng.net/view/_blank))  
为了理解如何旋转序列，可以将序列中的元素想象成手镯上的珠子。rotate() 操作会导致一个新元素成为开始迭代器所指向的第一个元素。在旋转之后，最后一个元素会在新的第一个元素之前。  
 rotate() 的第一个参数是这个序列的开始迭代器；第二个参数是指向新的第一个元素的迭代器，它必定在序列之内。第三个参数是这个序列的结束迭代器。图 1 中的示例说明在容器 ns 上的旋转操作使值为 4 的元素成为新的第一个元素，最后一个元素的值为 3。元素的圆形序列会被维持，因此可以有效地旋转元素环，直到新的第一个元素成为序列的开始。这个算法会返回一个迭代器，它指向原始的第一个元素所在的新位置。例如：

**std::vector<string> words { "one", "two", "three", "four", "five","six", "seven", "eight"};**

**auto iter = std::rotate(std::begin(words), std::begin(words)+3, std::end(words));**

**std::copy(std::begin(words), std::end(words),std::ostream\_iterator<string> {std::cout, " "});**

**std::cout << std::endl << "First element before rotation: " << \*iter << std::endl;**

这段代码对 words 中的所有元素进行了旋转。执行这段代码会生成如下内容：

**four five six seven eight one two three  
First element before rotation: one**

**输出说明 "four" 成为新的第一个元素，而且 rotate() 返回的迭代器指向之前的第一个元素"one"。**  
当然，不需要对容器中的全部元素进行旋转。例如：

**std::vector<string> words { "one", "two", "three", "four", "five","six", "seven", "eight", "nine", "ten"};**

**auto start = std::find(std:rbegin(words), std::end(words), "two");**

**auto end\_iter = std::find(std::begin(words), std::end(words), "eight");**

**auto iter = std::rotate(start, std::find(std::begin(words), std::end (words), "five") , end\_iter);**

**std::copy(std::begin(words), std::end(words), std::ostream\_iterator<string>{std::cout, " "});**

**std::cout << std::endl << "First element before rotation: " << \*iter << std::endl;**

这里用 find() 算法分别获取了和"two"、"eight"匹配的元素的迭代器。它们定义了被旋转的序列，这个序列是容器元素的子集。这个序列会被旋转为使"five"成为第一个元素，输出说明它是按预期工作的：

one five six seven two three four eight nine ten  
First element before rotation: two

**第三十八 C++ rotate\_copy（STL rotate\_copy）算法详解**

rotate\_copy() 算法会在新序列中生成一个序列的旋转副本，并保持原序列不变。rotate\_copy() 的前 3 个参数和 copy() 是相同的；第 4 个参数是一个输出迭代器，它指向目的序列的第一个元素。这个算法会返回一个目的序列的输出迭代器，它指向最后一个被复制元素的下一个位置。例如：

**std::vector<string> words {"one", "two", "three", "four", "five","six", "seven", "eight", "nine"，"ten"};**

**auto start = std::find(std::begin(words), std::end(words), "two");**

**auto end\_iter = std::find (std::begin(words) , std::end (words) ,"eight");**

**std::vector<string> words\_copy;**

**std::rotate\_copy(start, std::find(std::begin(words), std::end(words),"five") , end\_iter, std::back\_inserter (words\_copy));**

**std::copy(std::begin(words\_copy), std::end(words\_copy),std::ostream\_iterator<string> {std::cout, " "});**

**std::cout << std::endl;**

这段代码会对 word 中从 "two" 到 "seven" 的元素生成一个旋转副本。通过使用 back\_insert\_iterator 将复制的元素追加到 words\_copy 容器中，back\_insert\_iterator 会调用 words\_copy 容器的成员函数 push\_back() 来插入每个元素。这段代码产生的输出如下：

five six seven two three four

这里 rotate\_copy() 返回的迭代器是 words\_copy 中元素的结束迭代器。在这段代码中，并没有保存和使用它，但它却很有用。例如：

**std::vector<string> words {"one”，"two", "three", "four", "five","six", "seven", "eight", "nine", "ten"};**

**auto start = std::find (std::begin(words) , std::end(words) ,"two");**

**auto end\_iter = std::find(std::begin(words) , std::end(words),"eight"); std::vector<string> words\_copy {20}; // vector with 20 default elements**

**auto end\_copy\_iter = std::rotate\_copy(start,std::find(std::begin(words), std::end(words), "five"), end\_iter, std::begin(words\_copy));**

**std::copy (std::begin (words\_copy),end\_copy\_iter, std::ostream\_iterator<string>{std::cout," "});**

**std::cout << std::endl;**

生成的 words\_copy 容器默认有 20 个元素。rotate\_copy() 算法现在会将现有元素的旋转序列保存到 words\_copy 中。在输出时，这个算法返回的迭代器可以用来确定 words\_copy 的尾部边界；如果没有它，就必须通过源序列的元素个数来计算出尾部边界。

**第三十九 C++ move(STL move)函数使用详解**

move() 算法会将它的前两个输入迭代器参数指定的序列移到第三个参数定义的目的序列的开始位置，第三个参数必须是输出迭代器。这个算法返回的迭代器指向最后一个被移动到目的序列的元素的下一个位置。  
 这是一个移动操作，因此无法保证在进行这个操作之后，输入序列仍然保持不变；源元素仍然会存在，但它们的值可能不再相同了，因此在移动之后，就不应该再使用它们。如果源序列可以被替换或破坏，就可以选择使用 move() 算法。如果不想扰乱源序列，可以使用 copy() 算法。下面是一个展示如何使用它的示例：

*std*::*vector*<int> srce{ 1, 2, 3, 4 };

*std*::*deque*<int> dest{ 5, 6, 7, 8 };

*std*::move(*std*::*begin*(srce), *std*::*end*(srce), *std*::*back\_inserter*(dest));

这里会将 data 的最后 6 个元素移到容器的开头。它能够正常工作是因为目的地址在源序列之外。在移动之后，无法保证最后两个元素的值。这里它们虽然被移除了，但同样可以将它们重置为已知的值一一例如 0。  
 最后一行中的注释展示了输出结果。当然也可以用 rotate() 算法来代替 move() 移动元素，在这种情况下，我们肯定知道最后两个元素的值。  
 如果一个移动操作的目的地址位于源序列之内，move() 就无法正常工作，这意味着移动需要从序列的右边开始。原因是一些元素在移动之前会被重写，但 move\_backward() 可以正常工作。它的前两个参数指定了被移动的序列，第三个参数是目的地址的结束迭代器。例如：

*std*::*vector*<int> data{ 1, 2, 3, 4, 5, 6, 7, 8 };

*std*::move(*std*::*begin*(data) + 2, *std*::*end*(data), *std*::*begin*(data));

data.erase(*std*::*end*(data) - 2, *std*::*end*(data)); // Erase moved elements

*std*::copy(*std*::*begin*(data), *std*::*end*(data), *std*::*ostream\_iterator*<int> {*std*::*cout*, " "});

*std*::*cout* << *std*::*endl*;

// 3, 4, 5, 6, 7, 8这里使用 deque 容器只

是为了换个容器使用。将前 6 个元素向右移动两个位置。在移动操作后，值无法得到保证的元素会被重置为 0。最后一行展示了这个操作的结果

**第四十 C++ swap\_ranges(STL swap\_ranges)函数使用详解**

可以用 swap\_ranges() 算法来交换两个序列。这个算法需要 3 个正向迭代器作为参数。前两个参数分别是第一个序列的开始和结束迭代器，第三个参数是第二个序列的开始迭代器。显然，这两个序列的长度必须相同。这个算法会返回一个迭代器，它指向第二个序列的最后一个被交换元素的下一个位置。例如：

using Name = std::*pair*<*string*, *string*>; // First and second name

std::*vector*<Name> people{ Name{"Al", "Bedo" }, Name { "Ann", "Ounce"}, Name{"Jo"，"King"} };

std::*list*<Name> folks{ Name{"Stan", "Down"}, Name{"Dan","Druff"},Name {"Bea", "Gone"} };

std::swap\_ranges(std::*begin*(people), std::*begin*(people) + 2, ++std::*begin*(folks));

std::for\_each(std::*begin*(people), std::*end*(people), [](const Name& name) {std: : *cout* << '"' << name.*first* << " " << name.*second* << "\" "; });

std::*cout* << std::*endl*; // "Dan Druff" "Bea Gone" "Jo King"

std::for\_each(std::*begin*(folks), std::*end*(folks), [](const Name& name) {std::*cout* << '"' << name.*first* << " " << name.*second* << "\" "; });

std::*cout* << std::*endl*;// "Stan Down" "Al Bedo" "Ann Ounce"

这里使用 vector 和 list 容器来保存 pair<string,string> 类型的元素，pair<string,string> 用来表示名称。swap\_ranges() 算法被用来交换 people 的前两个元素和 folks 的后两个元素。这里并没有为了将 pair 对象写入流而重载 operator<<() 函数，因此 copy() 无法用输出流迭代器来列出容器的内容。为了生成输出，选择使用 for\_each() 算法将 lambda 表达式运用到容器的每个元素上。这个 lambda 表达式只会将传给它的 Name 元素的成员变量写入标准输出流。注释展示了执行这段代码后输出的结果：  
定义在 utility 头文件中的 swap() 算法的重载函数的模板原型为：

**template<typename T1, typename T2> void swap(std::pair<T1,T2> left, std::pair<T1,T2> right);**

这段代码会对 pair<T1,T2> 对象进行交换，在前面的代码段中也可以用 swap\_ranges() 来交换元素。  
 用来交换两个 T 类型对象的 swap() 模板也被定义在 utility 头文件中。除了 pair 对象的重载之外，utility 文件头中也有可以交换任何类型的容器对象的模板的重载。也就是说，可以交换两个 list<T> 容器或者两个 set<T> 容器但不能是一个 list<T> 和 vector<T>，也不能是一个 list<T1> 和一个 list<T2>。  
 另一个 swap() 模板的重载可以交换两个相同类型的数组。也有其他几个 swap() 的重载，它们可以用来交换其他类型的对象，包含元组和智能[指针](http://c.biancheng.net/c/80/" \t "http://c.biancheng.net/view/_blank)类型，正如本章前面所述。iter\_swap() 算法有一些不同，它会交换两个正向迭代器所指向的元素。

**第四十一 C++ remove、remove\_copy、remove\_if和remove\_copy\_if函数使用详解**

如果不知道具体的场景，即元素保存在什么样的容器中，是不能从序列中移除元素的。因此，“移除元素的”算法也无法做到这一点，它们只会重写被选择的元素或者忽略复制的元素。移除操作不会改变被“移除”元素的序列的元素个数。  
  
有 4 种移除算法：

（1）remove() 可以从它的前两个正向迭代器参数指定的序列中移除和第三个参数相等的对象。基本上每个元素都是通过用它后面的元素覆盖它来实现移除的。它会返回一个指向新的最后一个元素之后的位置的迭代器。

（2）remove\_copy() 可以将前两个正向迭代器参数指定的序列中的元素复制到第三个参数指定的目的序列中，并忽略和第 4 个参数相等的元素。它返回一个指向最后一个被复制到目的序列的元素的后一个位置的迭代器。序列不能是重叠的。

（3）remove\_if() 可以从前两个正向迭代器指定的序列中移除能够使作为第三个参数的谓词返回 true 的元素。

（4）4remove\_copy\_if() 可以将前两个正向迭代器参数指定的序列中，能够使作为第 4 个参数的谓词返回 true 的元素，复制到第三个参数指定的目的序列中。它返回一个指向最后一个被复制到目的序列的元素的后一个位置的迭代器。序列不能是重叠的。

可以按如下方式使用 remove():

*std*::*deque*<double> samples{ 1.5, 2.6, 0.0, 3.1, 0.0, 0.0, 4.1, 0.0, 6.7, 0.0 };

samples.erase(*std*::*remove*(*std*::*begin*(samples), *std*::*end*(samples), 0.0), *std*::*end*(samples));

*std*::copy(*std*::*begin*(samples), *std*::*end*(samples), *std*::*ostream* iterator <double> {*std*::*cout*, " "});

*std*::*cout* << *std*::*endl*;

// 1.5 2.6 3.1 4.1 6.7

sample 中不应包含为 0 的物理测量值。remove() 算法会通过左移其他元素来覆盖它们，通过这种方式就可以消除杂乱分布的 0。remove() 返回的迭代器指向通过这个操作得到的新序列的尾部，所以可以用它作为被删除序列的开始迭代器来调用 samples 的成员函数 erase()。注释说明容器中的元素没有被改变。  
 如果想保留原始序列，并生成一个移除选定元素之后的副本，可以使用 remove\_copy()。 例如：

*std*::*deque*<double> samples{ 1.5, 2.6, 0.0, 3.1, 0.0, 0.0, 4.1, 0.0, 6.7, 0.0 }; *std*::*vector*<double> edited\_samples;

*std*::remove\_copy(*std*::*begin*(samples), *std*::*end*(samples), *std*::*back\_inserter*(edited\_samples), 0.0);

samples 容器中的非零元素会被复制到 edited\_samples 容器中，edited\_samples 正好是一个 vector 容器。通过 back\_insert\_iterator 对象将这些元素添加到 edited\_samples，因此这个容器只包含从 sample 中复制的元素。  
 remove\_if() 提供了更强大的能力，它能够从序列中移除和给定值匹配的元素。谓词会决定一个元素是否被移除；它接受序列中的一个元素为参数，并返回一个布尔值。例如：

using Name = *std*::*pair*<*string*, *string*>； // First and second name

*std*::*set*<Name> blacklist{ Name {"Al", "Bedo"}, Name {"Ann", "Ounce"}, Name {"Jo","King"} };

*std*::*deque*<Name> candidates{ Name{"Stan", "Down"}, Name {"Al", "Bedo"}, Name {"Dan", "Druff"},Name {"Di", "Gress"}, Name {"Ann", "Ounce"}, Name {"Bea", "Gone"} }; candidates.erase(*std*::*remove\_if*(*std*::*begin*(candidates), *std*::*end*(candidates), [&blacklist](const Name& name) { return blacklist.*count*(name); }), *std*::*end*(candidates)); *std*::for\_each(*std*::*begin*(candidates), *std*::*end*(candidates), [](const Name& name) {*std*::*cout* << '"' << name.*first* << " " << name.*second* << "\" "; });

*std*::*cout* << *std*::*endl*; // "Stan Down" "Dan Druff" "Di Gress" "Bea Gone"

这段代码用来模拟候选人申请成为倶乐部会员。那些众所周知的不安分人士的姓名被保存在 blacklist 中，它是一个集合。当前申请成为会员的候选人被保存在 candidates 容器中，它是一个 deque 容器。用 remove\_if() 算法来保证不会有 blacklist 中的姓名通过甄选过程。这里的谓词是一个以引用的方式捕获 blacklist 容器的 lambda 表达式。当参数在容器中存在时，set 容器的成员函数 count() 会返回 1。谓词返回的值会被隐式转换为布尔值，因此对于每一个出现在 blacklist 中的候选人，谓词都会返回 true，然后会将它们从 candidates 中移除。注释中显示了通过甄选的候选人。  
remove\_copy\_if() 之于 *remove\_copy*()，就像 *remove\_if*() 之于 remove。下面展示它是如何工作的：

*std*::*set*<Name> blacklist{ Name {"Al", "Bedo"}, Name {"Ann", "Ounce"}, Name {"Jo", ,"King" } };

*std*::*deque*<Name> candidates{ Name {"Stan", "Down"}, Name { "Al", "Bedo"},Name {"Dan", "Druff"}, Name {"Di", "Gress"}, Name {"Ann", "Ounce"},Name {"Bea", "Gone"} };

*std*::*deque*<Name> validated;

*std*::remove\_copy\_if(*std*::*begin*(candidates), *std*::*end*(candidates), *std*::*back* inserter(validated), [&blacklist](const Name& name) { return blacklist.*count*(name); });

这段代码实现了和前一段代码同样的功能，除了结果被保存在 validated 容器中和没有修改 candidates 容器之外。

**第四十二 C++ fill和fill\_n函数用法详解**

fill() 和 fill\_n() 算法提供了一种为元素序列填入给定值的简单方式，fill() 会填充整个序列； fill\_n() 则以给定的迭代器为起始位置，为指定个数的元素设置值。下面展示了 fill() 的用法：

*std*::*vector*<*string*> data{ 12 }; // Container has 12 elements

*std*::fill(*std*::*begin*(data), *std*::*end*(data), "none"); // Set all elements to "none"

fill 的前两个参数是定义序列的正向迭代器，第三个参数是赋给每个元素的值。当然这个序列并不一定要代表容器的全部元素。例如：

*std*::*deque*<int> values(13); //Container has 13 elements

int n{ 2 }; // Initial element value

const int step{ 7 }; // Element value increment

const *size\_t* count{ 3 }; // Number of elements with given value

auto iter = *std*::*begin*(values);

while (true)

{

auto t0\_end = *std*::*distance*(iter, *std*::*end*(values)); // Number of elements remaining

if (to\_end < count) //In case no. of elements not a multiple of count

{

*std*::*fill*(iter, iter + to\_end, n); // Just fill remaining elements and end the loop

break;

}

else

{

*std*::*fill*(iter, *std*::*end*(values), n); // Fill next count elements

}

iter = *std*::*next*(iter, count); // Increment iter

n += step;

}

上面创建了具有 13 个元素的 value 容器。在这种情况下，必须用圆括号将值传给构造函数；使用花括号会生成一个有单个元素的容器，单个元素的值为 13。在循环中，fill() 算法会将 values 赋值给 count 个元素。以 iter 作为容器的开始迭代器，如果还有足够的元素剩下，每次遍历中，它会被加上 count，因此它会指向下个序列的第一个元素。执行这段代码会将 values 中的元素设置为：

2 2 2 9 9 9 16 16 16 23 23 23 30

fill\_n() 的参数分别是指向被修改序列的第一个元素的正向迭代器、被修改元素的个数以及要被设置的值。distance() 和 next() 函数定义在 iterator 头文件中。前者必须使用输入迭代器，而后者需要使用正向迭代器。

**第四十三 C++（STL）generate和generate\_n函数用法详解**

你已经知道可以用for\_each()算法将一个函数对象应用到序列中的每一个元素上。函数对象的参数是for\_each()的前两个参数所指定序列中元素的引用，因此它可以直接修改被保存的值。generate()算法和它有些不同，它的前两个参数是指定范围的正向迭代器，第三个参数是用来定义下面这种形式的函数的函数对象：

*T* fun (); // T is a type that can be assigned to an element in the range

无法在函数内访问序列元素的值。generate() 算法只会保存函数为序列中每个元素所返回的值，而且 genemte() 没有任何返回值。为了使这个算法更有用，可以将生成的不同的值赋给无参数函数中的不同元素。也可以用一个可以捕获一个或多个外部变量的函数对象作为 generate() 的第三个参数。例如：

*string* chars(30, ' ');// 30 space characters

char ch{ 'a' };

int incr{};

*std*::generate(*std*::*begin*(chars), *std*::*end*(chars), [ch, &incr]

{

incr += 3;

return ch + (incr % 26); })；

*std*::*cout* << chars << *std*: : *endl*;

// chars is: dgjmpsvybehknqtwzcfiloruxadgjm

变量 chars 被初始化为了个有 30 个空格的字符串。作为 generate() 的第三个参数的 lambda 表达式的返回值会被治存到 chars 的连续字符中。lambda 表达式以值的方式捕获 ch，以引用的方式捕获 incr，因此会在 lambda 的主体中对后者进行修改。lambda 表达式会返回 ch 加上 incr 后得到的字符，增加的值是 26 的模，因此返回的值总是在 'a' 到 'z' 之间，给定的起始值为 'a'。这个操作的结果会在注释中展示出来。可以对 lambda 表达式做一些修改， 使它可以用于任何大写或小写字母，但只生成保存在 ch 中的这种类型的字母。  
 generate\_n() 和 generate() 的工作方式是相似的。不同之处是，它的第一个参数仍然是序列的开始迭代器，第二个参数是由第三个参数设置的元素的个数。为了避免程序崩溃，这个序列必须至少有第二个参数定义的元素个数。例如：

*string* chars(30, ' '); // 30 space characters

char ch{ 'a' } / int incr{};

*std*::generate\_n(*std*::*begin*(chars), chars.*size*() / 2, [ch, &incr]

{

incr += 3;

return ch + (incr % 26);

})；

这里，chars 中只有一半的元素会被算法设为新的值，剩下的一半仍然为空格。

**第四十四 C++ transform(STL transform)函数用法详解**

transform() 可以将函数应用到序列的元素上，并将这个函数返回的值保存到另一个序列中，它返回的迭代器指向输出序列所保存的最后一个元素的下一个位置。  
 这个算法有一个版本和 for\_each() 相似，可以将一个一元函数应用到元素序列上来改变它们的值，但这里有很大的区别。for\_each() 中使用的函数的返回类型必须为 void，而且可以通过这个函数的引用参数来修改输入序列中的值；而 transform() 的二元函数必须返回一个值，并且也能够将应用函数后得到的结果保存到另一个序列中。  
 不仅如此，输出序列中的元素类型可以和输入序列中的元素类型不同。对于 for\_each()，函数总是会被应用序列的元素上，但对于 transform()，这一点无法保证。  
 第二个版本的 transform() 允许将二元函数应用到两个序列相应的元素上，但先来看一下如何将一元函数应用到序列上。在这个算法的这个版本中，它的前两个参数是定义输入序列的输入迭代器，第 3 个参数是目的位置的第一个元素的输出迭代器，第 4 个参数是一个二元函数。这个函数必须接受来自输入序列的一个元素为参数，并且必须返回一个可以保存在输出序列中的值。例如：

std::*vector*<double> deg\_C{ 21.0, 30.5, 0.0, 3.2, 100.0 };

std::*vector*<double> deg\_F(deg\_C.*size*());

std::transform(std::*begin*(deg\_C), std::*end*(deg\_C), std:*rbegin*(deg\_F), [](double temp) { return 32.0 + 9.0 \* temp / 5.0; });

//Result 69.8 86.9 32 37.76 212

这个 transform() 算法会将 deg\_C 容器中的摄氏温度转换为华氏温度，并将这个结果保存到 deg\_F 容器中。为了保存全部结果，生成的 deg\_F 需要一定个数的元素。因此第三个参数是 deg\_F 的开始迭代器。通过用 back\_insert\_iterator 作为 transform() 的第三个参数，可以将结果保存到空的容器中：

*std*::*vector*<double> deg\_F; // Empty container

*std*::transform(*std*::*begin*(deg\_C), *std*::*end*(deg\_C), *std*::*back\_inserter*(deg\_F), [](double temp) { return 32.0 + 9.0 \* temp / 5.0; });

// Result 69.8 86.9 32 37.76 212

用 back\_insert\_iterator 在 deg\_F 中生成保存了操作结果的元素；结果是相同的。第三个参数可以是指向输入容器的元素的迭代器。例如：

*std*::*vector*<double> temps{ 21.0, 30.5, 0.0, 3.2, 100.0 }; // In Centigrade

*std*::transform(*std*::*begin*(temps), *std*::*end*(temps), *std*::*begin*(temps), [](double temp) { return 32.0 + 9.0 \* temp / 5.0; });

// Result 69.8 86.9 32 37.76 212

这里将 temp 容器中的值从摄氏温度转换成了华氏温度。第三个参数是输入序列的开始迭代器，应用第 4 个参数指定的函数的结果会被存回它所运用的元素上。  
  
下面的代码展示了目的序列和输入序列是不同类型的情况：

*std*::*vector*<*string*> words{ "one", "two", "three", "four","five" };

*std*::*vector*<*size\_t*> hash\_values;

*std*::transform(*std*::*begin*(words), *std*::*end*(words), *std*::*back\_inserter*(hash\_values), *std*::*hash*<*string*>()); // string hashing function

*std*::copy(*std*::*begin*(hash\_values), *std*::*end*(hash\_values), *std*::*ostream\_iterator*<*size\_t*> {*std*::*cout*, " "});

*std*::*cout* << *std*::*endl*;

输入序列包含 string 对象，并且应用到元素的函数是一个定义在 string 头文件中的标准的哈希函数对象。这个哈希函数会返回 size\_t 类型的哈希值，并且会用定义在 iterator 头文件中的辅助函数 back\_inserter() 返回的 back\_insert\_iterator 将这些值保存到 hash\_values 容器中。在笔者的系统上，这段代码产生的输出如下：

**3123124719 3190065193 2290484163 795473317 2931049365**

你的系统可能会产生不同的输出。注意，因为目的序列是由 back\_insert\_iterator 对象指定的，这里 transform() 算法会返回一个 back\_insert\_iterator<vector<size\_T>> 类型的迭代器，因此不能在 copy() 算法中用它作为输入序列的结束迭代器。为了充分利用 transform() 返回的迭代器，这段代码可以这样写：

std::vector<string> words {"one", "two", "three", "four", "five"}; std::vector<size\_t> hash\_values(words.size());

auto end\_iter = std::transform(std::begin(words),std::end(words), std::begin(hash\_values), std::hash<string>()); // string hashing function

std::copy(std::begin(hash\_values) , end\_iter, std::ostream iterator<size t>{std::cout," "});

std::cout << std::endl;

现在，transform() 返回的是 hash\_values 容器中元素序列的结束迭代器。  
  
可以在 transform() 所运用的函数中为元素序列调用一个算法。下面举例说明：

std::deque<string> names {"S[tan](http://c.biancheng.net/ref/tan.html" \t "http://c.biancheng.net/view/_blank) Laurel", "Oliver Hardy", "Harold Lloyd"};

std::transform(std::begin(names), std::end(names), std::begin(names),[](string& s) { std::transform(std::begin(s), std::end(s), std::begin(s), ::[toupper](http://c.biancheng.net/ref/toupper.html" \t "http://c.biancheng.net/view/_blank));return s;})；

std::copy(std::begin(names), std::end(names), std::ostream iterator<string>{std::cout," "});

std::cout << std::endl;

transform() 算法会将 lambda 定义的函数应用到 names 容器中的元素上。这个 lambda 表达式会调用 transform()，将定义在 cctype 头文件中的 toupper() 函数应用到传给它的字符串的每个字符上。它会将 names 中的每个元素都转换为大写，因此输出为：

STAN LAUREL OLIVER HARDY HAROLD LLOYD

当然，也有其他更简单的方式可以得到相同的结果。  
  
应用二元函数的这个版本的 transform() 含有 5 个参数：

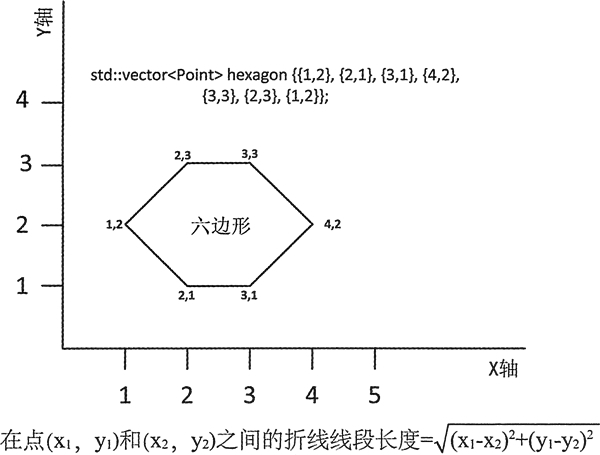
前两个参数是第一个输入序列的输入迭代器。

第3个参数是第二个输入序列的开始迭代器，显然，这个序列必须至少包含和第一个输入序列同样多的元素。

第4个参数是一个序列的输出迭代器，它所指向的是用来保存应用函数后得到的结果的序列的开始迭代器。

第5个参数是一个函数对象，它定义了一个接受两个参数的函数，这个函数接受来自两个输入序列中的元素作为参数，返回一个可以保存在输出序列中的值。

让我们来思考一个关于几何计算的简单示例。一条折线是由点之间连续的线组成的。折线可以表示为一个 Point 对象的 vector，折线线段是加入连续点的线。如果最后一个点和前一个点相同，折线就是闭合的一个多边形。

  
图 1 一条表示六边形的折线  
Point 被定义为一个类型别名，图 1 展示了一个示例：

u[sin](http://c.biancheng.net/ref/sin.html" \t "http://c.biancheng.net/view/_blank)g Point = std::pair<double, double>; // pair<x,y> defines a point

这里有 7 个点，因此图 1 中的六边形对象有 6 个折线段。因为第一个点和最后一个 点是相同的，这 6 条线段实际上组成了一个多边形——六边形。可以用 transform() 算法来 计算这些线段的长度：

*std*::*vector*<*Point*> hexagon{ {1,2}, {2,1}, {3,1}, {4,2}, {3,3}, {2,3}, {1,2} };

*std*::*vector*<double> segments; // Stores lengths of segments

*std*::transform(*std*::*begin*(hexagon), *std*::*end*(hexagon) — 1, *std*::*begin*(hexagon) + 1, *std*::*back\_inserter*(segments), [](const *Points* p1, const *Points* p2) {return *st* *d*::*sqrt*((p1.*first* - p2.*first*) \* (p1.*first* - p2.*first*) + (p1.*second* - p2.*second*) \* (p1.*second* - p2.*second*)); });

transform() 的第一个输入序列包含六边形中从第一个到倒数第二个 Point 对象。第二个输入序列是从第二个 Point 对象开始的，因此这个二元函数调用的连续参数为点 1 和 2、点 2 和 3、点 3 和 4，依此类推，直到输入序列的最后两个点 6 和 7。图 1 展示了计算 (x1,y1) 和 (x2,y2) 两点之前距离的公式，作为 transform() 最后一个参数的 lambda 表达式实现的就是这个公式。线段的长度是由 lambda 表达式计算的，它们会被保存在 segments 容器中。我们可以用两种以上的算法来输出线段的长度和这个六边形的周长。例如：

std::cout << "Segment lengths: ";

std::copy(std::begin(segments), std::end(segments),std::ostream\_iterator<double> {std::cout," "});

std::cout << std::endl;

std::cout << "Hexagon perimeter: "<< std::accumulate(std::begin(segments), std::end(segments), 0.0) << std::endl;

这里使用 copy() 算法来输出线段的长度。accumulate() 函数可以求出 segments 中元素值之和，从而得到周长。

**第四十五 C++ replace,replace\_if和replace\_copy函数用法详解**

replace() 算法会用新的值来替换和给定值相匹配的元素。它的前两个参数是被处理序列的正向迭代器，第 3 个参数是被替换的值，第 4 个参数是新的值。下面展示了它的用法:

*std*::*deque*<int> data{ 10, -5, 12, -6, 10, 8, -7, 10, 11 };

*std*::replace(*std*::*begin*(data), *std*::*end*(data), 10, 99);

// Result: 99 -5 12 -6 99 8 -7 99 11

这里，data 容器中和 10 匹配的全部元素都会被 99 替代。  
 replace\_if() 会将使谓词返回 true 的元素替换为新的值。它的第 3 个参数是一个谓词，第 4 个参数是新的值。参数的类型一般是元素类型的 const 引用；const 不是强制性的，但谓词不应该改变元素。下面是一个使用 replace\_if() 的示例：

*string* password{ "This is a good choice !" };

*std*::replace\_if(*std*::*begin*(password), *std*::*end*(password), [](char ch) {return *std*::*isspace*(ch); }, '\_');

//Result:This\_is\_a\_good\_choice!

这个谓词会为任何是空格字符的元素返回 true，因此这里的空格都会被下划线代替。  
 replace\_copy() 算法和 replace() 做的事是一样的，但它的结果会被保存到另一个序列中，而不会改变原始序列。它的前两个参数是输入序列的正向迭代器，第 3 个参数是输入序列的开始迭代器，最后两个参数分别是要被替换的值和替换值。例如：

*std*::*vector*<*string*> words{ "one","none", "two", "three", "none", "four" };

*std*::*vector*<*string*> new\_words;

*std*::replace\_copy(*std*::*begin*(words), *std*::*end*(words), *std*::*back\_inserter*(new\_words), *string*{ "none" }, *string*{ "0" });

// Result:"one", "0", "two","three","0","four"

在执行这段代码后，new\_words 会包含注释中的 string 元素。  
可以在序列中有选择地替换元素的最后一个算法是 replace\_copy\_if()，它和 replace\_if() 算法是相同的，但它的结果会被保存到另一个序列中。它的前两个参数是输入序列的迭代器，第 3 个参数是输出序列的开始迭代器，最后两个参数分别是谓词和替换值。例如：

*std*::*deque*<int> data{ 10, -5, 12, -6, 10, 8, -7, 10，11 }; *std*::*vector*<int> data\_copy;

*std*::replace\_copy\_if(*std*::*begin*(data), *std*::*end*(data), *std*::*back\_inserter*(data\_copy), [](int value) {return value == 10; }, 99);

// Result:99 -5 12 -6 99 8 -7 99 11

data\_copy 是一个 vector 容器，这里使用它只是为了说明输出容器可以和输入容器不同。这段代码执行后，它会包含注释中所示的元素。