模板类vector类似于string类，是一种动态数组，可以在运行的阶段设置vector对象的长度，可在末尾添加新的数据，还可以在中间添加新的数据，基本上，它是使用new创建动态数组的替代品。实际上，vector类确实使用new和delete来管理内存。

我们不深入研究vector到底是什么，只简单的讨论他的用法。首先，使用vector时要引入头文件#include,其次，vector包含在std空间中，第三，模板类用不同的语法来指出他的类型。第四，用不同的语法来表示不同的元素数

---------------------

vector可以理解为动态数组，其特点和基本操作方式与array相似，但vector可以动态调整大小。

        vector中的元素是连续存储的，这意味着我们不仅可以通过迭代器std::vector<T>::iterator，还可以像array一样使用下标来访问任意位置的元素。

vector支持的基本操作：点击打开链接

vector的元素存储机制

        我们可以考虑一下，vector中的元素是连续存储的，且vector的大小是可变的，那么如果向vector中添加元素会发生什么：如果没有空间容纳新元素，容器不可能简单地将它添加到内存中其他位置——因为元素必须连续存储。容器必须分配新的内存空间来保存已有元素和新元素，将已有元素从旧位置移到新空间中，然后添加新元素，释放旧存储空间。如果我们每添加一个新元素，vector就执行一次这样的内存分配和释放操作，就会严重影响性能。

        事实上，vector对象采用自动预分配存储空间的方式来减少存储空间重新分配的次数以提高性能。因此存储相同元素的vector往往比array要占用更多空间，因为vector预分配更多的内存来处理未来的容量增长。这种分配策略显然比每次添加新元素时都重新分配容器内存空间的策略要高效得多。虽然vector在每次重新分配内存空间时都要移动所有元素到新位置，但使用此策略后，其扩张操作通常比list和deque还要快。

        同时，vector还为我们提供了一些成员函数来有限地参与vector的内存分配活动：

        上述操作中，reserve并不改变容器中元素的数量，它仅影响vector预先分配多大的内存空间。并且只有当需要的容量超过当前容量时，调用reserve才会改变vector的容量。如果需求大小大于当前容量，reserve至少分配与需求一样大的内存空间（可能更大）。如果需求大小不大于当前容量，reserve什么也不做。特别是当需求大小小于当前容量时，容器不会退回内存空间，即在调用reserve后capacity将会大于或等于传递给reserve的参数。这样，调用reserve永远也不会减少容器所占的空间。

成员函数shrink\_to\_fit()从C++11开始引入，调用它可以退回不需要的内存空间，但并不保证一定退回内存空间。

#include <iostream>

#include <vector>

int main()

{

std::vector<int> ivec;

for (int i = 0; i < 5; ++i)

ivec.push\_back(i);

std::cout << "当前ivec中有" << ivec.size() << "个元素\t" << "当前最多能存储" << ivec.capacity() << "个元素" << std::endl;

ivec.shrink\_to\_fit();

std::cout << "调用shrink\_to\_fit()函数后的capacity为：" << ivec.capacity() << std::endl;

ivec.reserve(3);//reserve无法缩小vector的容量

std::cout << ivec.capacity() << std::endl;

ivec.reserve(10);

std::cout << ivec.capacity() << std::endl;

return 0;

}

本文部分内容摘自《C++ Primer（中文版）第五版》

---------------------

容器是一些特定类型对象的集合，容器类分为序列式容器和关联容器两种。

**容器基本操作**

 容器类的一些基本操作如下图：



**定义和初始化**

    每个容器都定义了一个默认构造函数。除array之外，其他容器的默认构造函数都会创建一个指定类型的空容器，且都可以接受指定容器大小和元素初始值的参数。



**将一个容器初始化为另一个容器的拷贝**

    将一个新容器创建为另一个容器的拷贝的方法有两种：可以直接拷贝整个容器，或者拷贝由一个迭代器对指定的元素范围。后者对array不适用。  
    用第一种方式创建一个容器为另一个容器的拷贝时，两个容器的类型及其元素类型必须匹配。用第二种方式——传递迭代器参数，来拷贝时，就不要求容器类型是相同的了。而且新容器和原容器中的元素类型也可以不同，只要能将要拷贝的元素转换为要初始化的容器的元素类型即可（例如可以将const char\*元素转换为string）。

**序列式容器**

        序列式容器为程序员提供了控制元素存储和顺序访问的能力。这种顺序不依赖于元素的值，而是与元素加入容器时的位置相对应。  
        C++的序列式容器有vector、deque、list、forward\_list、array和string几种。它们都提供了快速顺序访问元素的操作。但它们在访问顺序、添加/删除元素的代价方面存在差异。  
        string和vector都是可变大小的容器类，string专门用于保存字符，两者都将元素保存在连续的内存空间中。由于元素是连续存储的，由元素下标来计算其地址是非常快速的。但是，在这两种容器的中间位置添加/删除元素就会非常耗时：在一次插入/删除操作后，需要移动插入/删除之后的所有元素来保持连续存储。而且，添加一个元素有时可能还需要分配额外的存储空间，这种情况下每个元素都必须移动到新的存储空间中。  
        list和forward\_list两个容器支持容器任何位置的快速插入/删除操作，list是双向链表，forward\_list是单向链表。作为代价，这两个容器不支持元素的随机访问，只能够顺序访问。而且，与vector和deque、array相比，这两个容器的额外内存开销较大。  
        deque是一个双端队列，支持快速随机访问，但在除头尾以外的位置添加/删除元素的代价较大。但是，在deque的两端添加/删除元素的速度与list或forward\_list的速度相当。

        array是最接近原始数组的一个容器类，其对象的大小是固定的。因此，array不支持添加和删除元素以及改变容器大小的操作。与内置数组相比，array是一种更安全、更容易使用的数组类型。

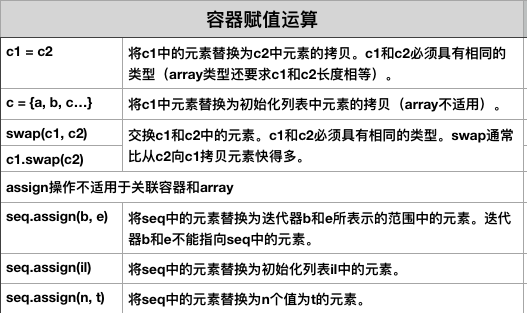
        序列式容器的特点决定我们应该在何种情况下使用何种容器。通常，使用vector是最好的选择，除非你有很好的理由选择其他容器。如果程序要求随机访问元素，不应使用list和forward\_list。如果程序只需要在头尾位置插入或删除元素，则使用deque。一般来说，应用中占主导地位的操作决定了容器类型的选择。

**相关操作**

**赋值和swap**

赋值运算符将其左边容器中的全部元素替换为右边容器中元素的拷贝：  
c1 = c2;  
c1 = {a, b, c};  
        第一个赋值运算后，左边容器将与右边容器相等。如果两个容器原来大小不同，赋值运算后两者的大小都与右边容器的原大小相同。第二个赋值运算后，c1的size变为3，即花括号列表中值的数目。  
        与内置数组不同，标准库array类型允许赋值。赋值号左右两边的运算对象必须具有相同的类型。并且，由于array类型大小固定，还要求赋值号左右两边的运算对象的大小相等。总之，所有试图改变原来array类型大小的操作都是不被允许的。  
使用assign（仅序列式容器）  
        赋值运算符要求左边和右边的运算对象具有相同的类型。它将右边运算对象中所有元素拷贝到左边运算对象中。顺序容器（array除外）定义了一个assign成员函数，允许我们从一个不同但相容的类型赋值，或者从容器的一个子序列赋值。assign操作用参数所指定元素的拷贝替换左边容器中的所有元素。  
list<string> names;  
vector<const char\*> oldstyle;  
names.assign(oldstyle.cbegin(), oldstyle.cend());

        上述代码将names中的元素替换为迭代器指定的范围中的元素的拷贝。assign的参数决定了容器中将有多少个元素以及值是什么。这一操作是不能用赋值运算符完成的，因为两个容器的类型不同。



**使用swap**

        swap操作交换两个相同类型容器的内容。调用swap后，两个容器中的元素将会交换。而且，用swap操作来交换两个容器的速度会很快，因为元素本身并未交换，swap只是交换了两个容器的内部数据结构。也即swap不对元素进行拷贝、删除或插入操作，因此可以保证在常数时间内完成。  
        元素不会被移动的事实意味着，除string外，指向容器的迭代器、引用和指针在swap后都不会失效，它们仍然指向swap操作之前所指向的元素。但是，在swap操作后这些元素已经属于不同的容器了。  
        swap操作有两个例外，一个是string，对string调用swap会导致迭代器、引用和指针失效。另一个是array，对array调用swap操作会真正交换它们的元素。因此，对于array，swap操作后指针、引用和迭代器所绑定的元素位置保持不变，但元素值已经与另一个array中对应元素的值进行了交换。  
        上述对swap操作的描述都只针对非成员版本的swap，即std::swap

**容器大小操作**

        出forward\_list不支持size外，每个容器类型都有三个与大小相关的操作，这三个成员函数分别是：size、empty和max\_size

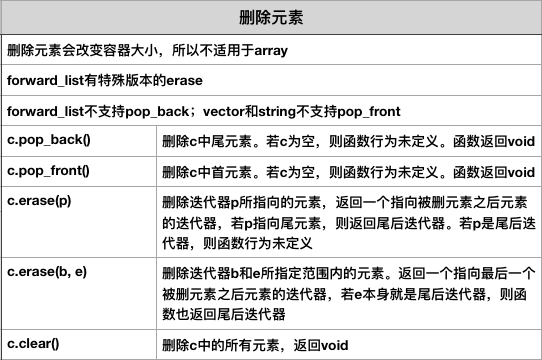
**序列式容器操作**

**添加元素**

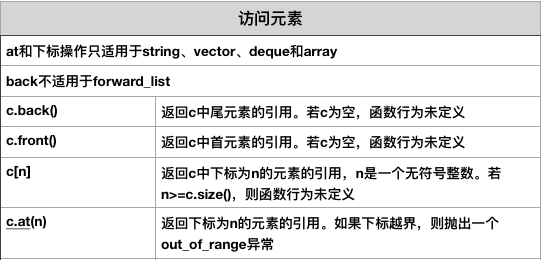
        从下表可以看到，虽然某些容器不支持push\_front操作，但它们对于insert操作并无类似的限制（插入开始位置）。因此我们可以使用insert将元素插入到容器的开始位置，而不必担心容器是否支持push\_front。但需要注意的是，虽然将元素插入到vector、deque和string中的任何位置都是合法的。然而，这样做可能很耗时。



**删除元素**



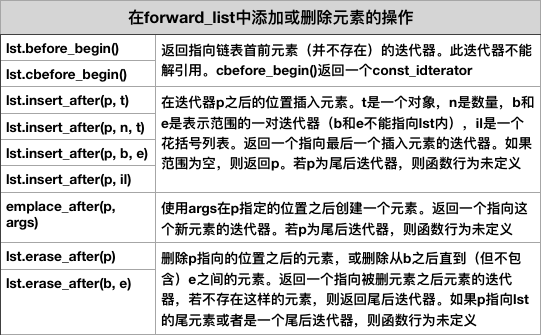
**访问元素**



**特殊的forward\_list操作**

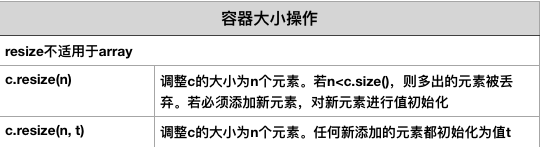
        当添加或删除一个元素时，删除或添加的元素之前的那个元素的后继会发生改变。为了添加或删除一个元素，我们需要访问其前驱，以便改变前驱的链接。但是，forward\_list是一个单向链表。在一个单向链表中，没有简单的方法来获取一个元素的前驱。出于这个原因，在一个forward\_list中添加或删除元素的操作是通过改变给定元素之后的元素来完成的。这样，我们总是可以访问到被添加或删除操作所影响的元素。

        由于这些操作与其他容器上的操作的实现方式不同，forward\_list并未定义insert、emplace和erase，而是定义了名为insert\_after、emplace\_after和erase\_after的操作。



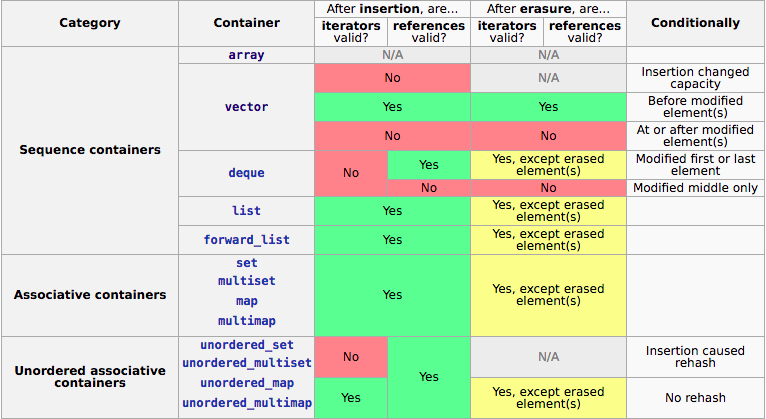
**改变容器的大小**

    我们可以用resize来增大或缩小容器。如果当前大小大于所要求的大小，容器后部的元素会被删除；如果当前大小小于所要求的大小，会将新元素添加到容器后部。



**容器操作可能使迭代器失效**

        向容器中添加元素和从容器中删除元素的操作可能会使指向容器元素的指针、引用或迭代器失效。一个失效的指针、引用或迭代器将不再表示任何元素。



        在向容器添加元素后：  
·如果容器是vector或string，且存储空间被重新分配，则指向容器的迭代器、指针和引用都会失效。如果存储空间未重新分配，指向插入位置之前的元素的迭代器、指针和引用仍有效，但指向插入位置之后元素的迭代器、指针和引用将会失效。  
·对于deque，插入到除首尾位置之外的任何位置都会导致迭代器、指针和引用失效。如果在首尾位置添加元素，迭代器会失效，但指向存在的元素的引用和指针不会失效。  
·对于list和forward\_list，指向容器的迭代器（包括尾后迭代器和首前迭代器）、指针和引用仍有效。  
        从容器中删除元素后：  
·所有容器中指向被删除元素的迭代器、指针和引用会失效。  
·对于list和forward\_list，指向容器其他位置的迭代器（包括尾后迭代器和首前迭代器）、引用和指针仍有效。  
·对于deque，如果在首尾位置之外的任何位置删除元素，那么指向被删除元素外的其他元素的迭代器、引用或指针也会失效。如果是删除deque的尾元素，则尾后迭代器也会失效，但其他迭代器、引用和指针不受影响；如果是删除首元素，这些也不会受影响。  
·对于vector和string，指向被删除元素之前的元素的迭代器、引用和指针仍有效。  
        由于向迭代器添加元素和从迭代器删除元素的代码可能会使迭代器失效，因此必须保证每次改变容器的操作之后都正确地重新定位迭代器。另外，最小化要求迭代器必须保持有效的程序片段也是一个便于管理迭代器的好方法。  
  
  
        C++11开始的新标准库比旧版本快得多，原因是新标准库支持移动对象的操作，而旧标准库只能拷贝对象。因此新标准库容器的性能相较于旧标准库有比较大的提升。

本文摘自《C++ Primer（中文版）第五版》