容器类共享一个公共接口，每个容器都以自己的方式扩展该接口。这种通用接口使库更容易学习;我们对一种容器的了解适用于另一种容器。每种容器都提供了一组不同的性能和功能权衡。

容器保存指定类型的对象的集合。顺序容器允许程序员控制元素的存储和访问顺序。该顺序不依赖于元素的值。相反，顺序对应于元素放入容器的位置。相比之下，有序和无序的关联容器（我们将在第11章中介绍）根据键的值存储它们的元素。

文本

描述已自动生成

字符串和向量将其元素保存在连续内存中。因为元素是连续的，所以从元素的索引计算元素的地址是快速的。

list和forward\_list容器的设计目的是使在容器中的任何位置添加或删除元素变得更快。作为交换，这些类型不支持对元素的随机访问：我们只能通过迭代容器来访问元素。此外，与vector、deque和array相比，这些容器的内存开销通常很大。

双端队列是一种更复杂的数据结构。像字符串和向量一样，双端队列支持快速随机访问。与字符串和向量一样，在双端队列中间添加或删除元素是一个（潜在的）昂贵的操作。然而，在双端队列的两端添加或删除元素是一个快速的操作，与将元素添加到列表或forward\_list相当。

文本

描述已自动生成

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

由于我们将在§ 13.6（p.531）中解释的原因，新的库容器比以前的版本快得多。几乎可以肯定，库容器的性能与最精心设计的替代品一样好（通常更好）。现代C++程序应该使用库容器，而不是更原始的结构，如数组。

·除非你有理由使用另一个容器，否则请使用vector。

·如果你的程序有很多小元素和空间开销问题，不要使用list或forward\_list。

·如果程序需要随机访问元素，请使用vector或deque。

如果程序需要在容器中间插入或删除元素，请使用list或forward\_list。

如果程序需要插入或删除前面和后面的元素，而不是中间的元素，请使用双端队列。

·如果程序只需要在阅读输入时在容器的中间插入元素，并且随后需要随机访问元素：-首先，决定是否实际需要在容器的中间添加元素。当您完成输入时，添加到向量中，然后调用库排序函数（我们将在§ 10.2.3（第384页）中介绍）来重新排序容器通常会更容易。

- 如果必须插入到中间，请考虑使用输入阶段的列表。一旦输入完成，将列表复制到向量中。

图片包含 图示

描述已自动生成

文本

描述已自动生成

文本

描述已自动生成

如果程序需要随机访问，需要在容器中间插入和删除元素，该怎么办？这个决定将取决于访问列表或forward\_list中的元素的相对成本与插入或删除vector或deque中的元素的成本。一般来说，应用程序的主要操作（无论是更多的访问还是更多的插入或删除）将决定容器类型的选择。在这种情况下，可能需要使用两个容器对应用程序进行性能测试

如果你不确定使用哪个容器，那么编写代码时，让它只使用向量和列表通用的操作：使用迭代器，而不是下标，并避免对元素的随机访问。这样就可以很容易地根据需要使用vector或list。

假如不确定，那么使用迭代器，代替特性

**1. 类型别名**

* **iterator**: 容器的迭代器类型。
* **const\_iterator**: 只能读取但不能修改元素的迭代器类型。
* **size\_type**: 无符号整数类型，足以容纳该容器类型的最大可能大小。
* **difference\_type**: 有符号整数类型，足以容纳两个迭代器之间的距离。
* **value\_type**: 容器中元素的类型。
* **reference**: 元素的左值类型，是 value\_type 的同义词。
* **const\_reference**: 元素的常量左值类型（即 const value\_type）。

**2. 构造函数**

* **C c;**: 默认构造函数，创建一个空容器（不适用于 array）。
* **C c1(c2);**: 构造 c1 作为 c2 的副本。
* **C c(b, e);**: 从迭代器 b 和 e 表示的范围复制元素（不适用于 array）。
* **C c{a, b, c...};**: 使用列表初始化 c。

**3. 赋值和交换**

* **c1 = c2**: 用 c2 中的元素替换 c1 中的元素。
* **c1 = {a, b, c...}**: 用列表中的元素替换 c1 中的元素（不适用于 array）。
* **a.swap(b)**: 交换 a 和 b 中的元素。
* **swap(a, b)**: 等同于 a.swap(b)。

**4. 大小操作**

* **c.size()**: 返回 c 中元素的数量（不适用于 forward\_list）。
* **c.max\_size()**: 返回 c 可以容纳的最大元素数量。
* **c.empty()**: 如果 c 为空，则返回 true，否则返回 false。

**5. 添加/删除元素（不适用于 array）**

* **c.insert(args)**: 根据 args 将元素插入到 c 中。
* **c.emplace(inits)**: 使用 inits 在 c 中构造一个元素。
* **c.erase(args)**: 根据 args 删除 c 中的元素。
* **c.clear()**: 删除 c 中的所有元素，返回 void。

**. 相等性和关系运算符**

* **==, !=**: 适用于所有容器类型的相等性比较。
* **<, <=, >, >=**: 关系运算符（不适用于无序关联容器）。

**7. 获取迭代器**

* **c.begin(), c.end()**: 返回指向 c 中第一个元素和最后一个元素之后位置的迭代器。
* **c.cbegin(), c.cend()**: 返回 const\_iterator。

**8. 可逆容器的额外成员（不适用于 forward\_list）**

* **reverse\_iterator**: 以逆序访问元素的迭代器。
* **const\_reverse\_iterator**: 不能修改元素的逆序迭代器。
* **c.rbegin(), c.rend()**: 返回指向 c 中最后一个元素和第一个元素之前位置的迭代器。
* **c.crbegin(), c.crend()**: 返回 const\_reverse\_iterator。

**1. 默认构造函数**

* **C c;**: 创建一个空容器。如果 C 是 array 类型，则 c 中的元素会被默认初始化；否则，c 是一个空容器。

**2. 拷贝构造函数**

* **C c1(c2);**: 创建一个容器 c1，它是容器 c2 的副本。c1 和 c2 必须具有相同的类型（即相同的容器类型和元素类型）。对于 array，它们还必须具有相同的大小。

**3. 列表初始化**

* **C c{a, b, c...};**: 使用初始化列表中的元素创建一个容器 c。初始化列表中的元素类型必须与 C 的元素类型兼容。对于 array，列表中的元素数量必须与 array 的大小相同或更少，缺少的元素会被值初始化。

**4. 范围构造函数**

* **C c(b, e);**: 使用迭代器 b 和 e 表示的范围内的元素创建一个容器 c。元素类型必须与 C 的元素类型兼容。此构造函数不适用于 array。

**5. 带大小的构造函数（仅适用于顺序容器，不包括 array）**

* **C seq(n);**: 创建一个包含 n 个值初始化元素的顺序容器 seq。此构造函数是显式的（explicit），不适用于 string。
* **C seq(n, t);**: 创建一个包含 n 个值为 t 的元素的顺序容器 seq。

图形用户界面, 文本, 应用程序

描述已自动生成

文本

描述已自动生成

与容器一样，迭代器有一个公共接口：如果一个迭代器提供了一个操作，那么每个提供该操作的迭代器都以相同的方式支持该操作。例如，标准容器类型上的所有迭代器都允许我们访问容器中的元素，它们都通过提供解引用操作符来实现。类似地，库容器的迭代器都定义了从一个元素移动到下一个元素的递增操作符。

除了一个例外，容器迭代器支持表3.6中列出的所有操作。例外是forward\_list迭代器不支持递减（--）运算符。表3.7中列出的迭代器算术运算只适用于字符串、向量、双端队列和数组的迭代器。我们不能在任何其他容器类型的迭代器上使用这些操作。

迭代器范围的概念是标准库的基础。

每个迭代器引用同一容器中的一个元素或最后一个元素之后的元素。这两个迭代器，通常被称为开始和结束--或者（有点误导）称为第一个和最后一个--标记容器中的元素范围。

范围中的元素包括由第一个表示的元素和从第一个直到但不包括最后一个的每个元素。

迭代器开始和结束必须引用同一个容器。迭代器end可以等于开始，但不能引用开始所表示的元素之前的元素。

文本

描述已自动生成

当不需要写访问时，使用cbegin和dbind

**1. 通过拷贝创建容器**

* **要求**:
  + 容器的类型必须完全匹配。
  + 元素的类型也必须完全匹配。

**2. 通过迭代器范围创建容器**

* **要求**:
  + 容器类型可以不同。
  + 元素类型可以不同，只要能够进行隐式转换。

具有大小的构造函数仅对顺序容器有效;关联容器不支持它们。

文本, 信件

描述已自动生成

文本

描述已自动生成

因为现有元素被替换，所以传递给assign的迭代器不能引用调用assign的容器。

std::vector<int> vec = {1, 2, 3, 4, 5};

vec.assign(vec.begin(), vec.end()); // 错误：迭代器指向调用 assign 的容器本身

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

每个容器类型都支持相等运算符（==和！=）;除了无序关联容器之外的所有容器也支持关系运算符（>，>=，<，<=）。

* **相等性比较**:
  + 两个容器相等当且仅当它们的大小相同且所有对应位置的元素相等。
* **大小比较**:
  + 如果两个容器的大小不同，但较小容器的所有元素与较大容器的对应元素相等，则较小容器小于较大容器。
  + 如果存在第一个不相等的元素，则比较结果取决于这两个元素的比较结果。

**2. forward\_list 的特殊性**

* forward\_list 是一个单向链表，因此它不支持 push\_back 和 emplace\_back，但提供了特殊版本的 insert 和 emplace

**3. push\_back 和 emplace\_back**

* **c.push\_back(t)**:
  + 在容器 c 的末尾添加一个值为 t 的元素。
* **c.emplace\_back(args)**:
  + 在容器 c 的末尾直接构造一个元素，使用 args 作为构造函数的参数。
* **适用容器**:
  + 适用于 vector、deque、list 和 string。
  + 不适用于 forward\_list 和 array。

**4. push\_front 和 emplace\_front**

* **c.push\_front(t)**:
  + 在容器 c 的开头添加一个值为 t 的元素。
* **c.emplace\_front(args)**:
  + 在容器 c 的开头直接构造一个元素，使用 args 作为构造函数的参数。
* **适用容器**:
  + 适用于 deque、list 和 forward\_list。
  + 不适用于 vector 和 array。
* 向 vector、string 或 deque 添加元素可能会导致所有现有的迭代器、引用和指针失效。
* 这是因为这些容器在添加元素时可能需要重新分配内存，导致元素的地址发生变化。

此外，向向量或字符串添加元素可能会导致整个对象重新分配。重新分配对象需要分配新的内存并将元素从旧空间移动到新空间。

当向 vector 或 string 添加元素时，可能会触发以下步骤：

**步骤 1：检查容量**

* 如果当前容量足够（即 size + 1 <= capacity），则直接将新元素添加到末尾。
* 如果当前容量不足（即 size + 1 > capacity），则需要重新分配内存。

**步骤 2：重新分配内存**

* 容器会分配一块更大的内存空间（通常是当前容量的两倍）。
* 将旧空间中的所有元素复制到新空间中。
* 将新元素添加到新空间的末尾。
* 释放旧空间的内存。

**步骤 3：更新迭代器、引用和指针**

* 由于内存地址发生了变化，所有指向旧空间的迭代器、引用和指针都会失效。

当我们使用一个对象初始化一个容器，或者将一个对象插入到一个容器中时，该对象的值的副本被放置在容器中，而不是对象本身。就像我们把一个对象传递给一个非引用参数一样（第6.2.1节），容器中的元素和产生该值的对象之间没有任何关系。对容器中元素的后续更改对原始对象没有影响，反之亦然。

在向量、双端队列或字符串中的任何位置插入都是法律的。然而，这样做可能是一个昂贵的操作。

**3. 与 vector 的对比**

* **vector**:
  + vector 是基于动态数组实现的，所有元素在内存中是连续存储的。
  + 在 vector 的前面插入元素会导致所有元素整体后移，时间复杂度为 **O(n)**。
* **deque**:
  + deque 是基于分段连续内存块实现的，元素可以分散存储在多个内存块中。
  + 在 deque 的前面插入元素不会导致所有元素整体后移，时间复杂度为 **O(1)**。

emplace函数在容器中构造元素。这些函数的参数必须与元素类型的构造函数匹配。

当我们调用push或insert成员时，我们传递元素类型的对象，这些对象被复制到容器中。当我们调用emplace成员时，我们将参数传递给元素类型的构造函数。emplace成员使用这些参数直接在容器管理的空间中构造元素。

* **at 和下标运算符（[]）**:
  + 适用于 string、vector、deque 和 array。
  + at 会检查索引范围，[] 不会。
* **back 函数**:
  + 适用于 string、vector、deque、list 和 array。
  + 不适用于 forward\_list。
* **front 函数**:
  + 适用于所有顺序容器。
* **未定义行为**:
  + 访问空容器或超出范围的元素会导致未定义行为。
* **副本的创建**:
  + 所有这些操作（\*c.begin()、c.front()、\*(--last)、c.back()）都会返回元素的引用或值。
  + 使用 auto val 或 auto val2 等变量时，会创建元素的副本。

访问容器中元素的成员（即，front、back、subscribe和at）返回引用。如果容器是一个const对象，返回值是对const的引用。如果容器不是const对象，返回值是一个普通的引用，我们可以使用它来改变获取的元素的值

如果我们使用auto来存储这些函数之一的返回值，并且我们想使用该变量来更改元素，我们必须记住将变量定义为引用类型。

提供快速随机访问的容器（字符串、向量、双端队列和数组）也提供下标操作符（第3.3.3节）。

* **pop\_back**:
  + 删除最后一个元素，适用于 vector、deque、list 和 string。
* **pop\_front**:
  + 删除第一个元素，适用于 deque、list 和 forward\_list。
* **erase**:
  + 删除指定位置或范围内的元素，适用于 vector、deque、list 和 string。
* **clear**:
  + 删除所有元素，适用于所有顺序容器（除了 array）。
* **注意事项**:
  + 删除操作可能会导致迭代器、引用和指针失效。

删除元素的成员不检查其参数。程序员在删除元素之前必须确保它们存在

* **before\_begin 和 cbefore\_begin**:
  + 返回指向链表开头之前位置的迭代器，用于在链表开头插入元素。
* **insert\_after**:
  + 在指定位置之后插入元素。
* **emplace\_after**:
  + 在指定位置之后直接构造元素。
* **erase\_after**:
  + 删除指定位置之后的元素。
* **c.resize(n)**:
  + 将容器大小调整为 n，新添加的元素会被值初始化。
* **c.resize(n, t)**:
  + 将容器大小调整为 n，新添加的元素会被初始化为 t。
* **影响**:
  + 缩小容器会使指向被删除元素的迭代器、引用和指针失效。
  + 扩大容器可能会使所有迭代器、引用和指针失效（对于 vector、string 和 deque）

从容器中添加或删除元素的操作可以使指向容器元素的指针、引用或迭代器无效。无效的指针、引用或迭代器是不再表示元素的指针、引用或迭代器。使用无效的指针、引用或迭代器是一个严重的编程错误，可能会导致与使用未初始化的指针相同的问题（第2.3.2节）。

如果容器被重新分配，则迭代器、指针和对向量或字符串的引用无效。如果没有发生重新分配，则插入之前对元素的间接引用仍然有效;插入之后对元素的间接引用无效。

**1. vector 和 string**

* **内存重新分配**:
  + 如果添加元素导致容器重新分配内存（即容量不足），则所有迭代器、指针和引用都会失效。
  + 这是因为重新分配内存后，元素的地址发生了变化。
* **未重新分配内存**:
  + 如果没有重新分配内存，则指向插入位置之前的元素的迭代器、指针和引用仍然有效。
  + 指向插入位置之后的元素的迭代器、指针和引用会失效。

**2. deque**

* **在开头或结尾添加元素**:
  + 如果添加元素在 deque 的开头或结尾，则迭代器会失效，但指向现有元素的引用和指针仍然有效。
* **在中间添加元素**:
  + 如果添加元素在 deque 的中间，则所有迭代器、指针和引用都会失效

**. list 和 forward\_list**

* **添加元素**:
  + 在 list 或 forward\_list 中添加元素不会使任何迭代器、指针或引用失效。
  + 包括尾后迭代器（end）和开头之前的迭代器（before\_begin）仍然有效。

**1. 删除元素的基本规则**

* **指向被删除元素的迭代器、指针和引用**:
  + 无论哪种容器，指向被删除元素的迭代器、指针和引用都会失效，因为这些元素已经被销毁。

**3. deque**

* **删除中间元素**:
  + 如果删除 deque 中间的元素，则所有迭代器、指针和引用都会失效。
* **删除开头或结尾元素**:
  + 如果删除 deque 开头的元素，则所有迭代器、指针和引用仍然有效。
  + 如果删除 deque 结尾的元素，则尾后迭代器 end 会失效，但其他迭代器、指针和引用仍然有效。

**4. vector 和 string**

* **删除元素后**:
  + 指向删除位置之前的元素的迭代器、指针和引用仍然有效。
  + 指向删除位置之后的元素的迭代器、指针和引用会失效。
  + **尾后迭代器 end 总是会失效**，因为删除元素会改变容器的大小。

当你使用迭代器（或者容器元素的引用或指针）时，最好尽量减少迭代器必须保持有效的程序部分。因为向容器添加或移除元素的代码可能会使迭代器无效，所以需要确保在每次更改容器的操作之后，适当地重新定位迭代器。这个建议对于vector、string和deque尤其重要。

当我们添加或删除向量或字符串中的元素时，或者添加或删除双端队列中除第一个元素之外的任何元素时，end返回的迭代器总是无效的。因此，添加或删除元素的循环应该总是调用end，而不是使用存储的副本。部分出于这个原因，C++标准库通常被实现为调用end（）是一个非常快的操作

不要在插入或删除双端队列、字符串或向量中的元素的循环中缓存end（）返回的迭代器。

如果没有空间容纳新元素，容器不能只是在内存中的其他地方添加元素-元素必须是连续的。相反，容器必须分配新的内存来保存现有元素和新元素，将元素从旧位置移动到新空间，添加新元素，并释放旧内存。如果vector在我们每次添加元素时都进行这种内存分配和释放，那么性能将慢得无法接受。

**2. 内存分配策略**

为了减少内存重新分配的次数，标准库实现者采用了以下策略：

**（1）预分配额外容量**

* 当 vector 或 string 需要重新分配内存时，它们通常会分配比当前需求更多的容量。
* 例如，如果当前容量为 4，添加第 5 个元素时，容器可能会将容量扩展到 8 或更多。
* 这种额外的容量被称为**预留容量（reserved capacity）**。

**（2）使用预留容量**

* 当添加新元素时，容器会使用预留容量，而不需要立即重新分配内存。
* 只有在预留容量用尽时，容器才会再次重新分配内存。

**3. 性能优势**

* **减少重新分配次数**:
  + 通过预分配额外容量，vector 和 string 显著减少了内存重新分配的次数。
  + 例如，如果每次容量翻倍，添加 n 个元素的时间复杂度为 **O(n)**，而不是每次重新分配的 **O(n^2)**。
* **高效的内存使用**:
  + 尽管每次重新分配时需要移动所有元素，但由于重新分配次数减少，整体性能仍然很高。
* **与 list 和 deque 的对比**:
  + list 是基于链表的实现，每次插入元素都需要分配新节点，因此插入操作的时间复杂度是 **O(1)**，但由于内存碎片化和缓存不友好，实际性能可能不如 vector。
  + deque 是基于分段连续内存块的实现，插入操作的时间复杂度是 **O(1)**，但由于其复杂的内存管理，实际性能也可能不如 vector。
* **reserve**:
  + 用于预分配内存空间，避免频繁的内存重新分配。
  + 不会减少容器的容量。
* **resize**:
  + 用于调整容器中元素的数量。
  + 不会减少容器的容量。
* **shrink\_to\_fit**:
  + 用于请求容器释放未使用的内存空间。
  + 不保证容器会释放内存。

调用shrink\_to\_fit只是一个请求;不能保证库会返回内存。

每个向量实现都可以选择自己的分配策略。但是，在强制它分配新内存之前，它不能这样做。“

* **vector 的内存分配策略**:
  + 按需分配内存，只有在必要时才分配新的内存。
  + 不同的实现可以选择自己的分配策略，但必须遵循这一基本原则。

除了顺序容器之外，该库还定义了三个顺序容器适配器：stack、queue和priority\_queue。适配器是库中的一般概念。这里有容器、迭代器和函数适配器。从本质上讲，适配器是一种使一个事物像另一个事物那样工作的机制。容器适配器采用现有的容器类型，并使其像其他类型一样工作。例如，堆栈适配器采用顺序容器（而不是array或forward\_list），并使其像堆栈一样操作。表9.17列出了所有容器适配器通用的操作和类型。

容器适配器包括：

* **std::stack**（栈）
* **std::queue**（队列）
* **std::priority\_queue**（优先队列）

这些适配器基于其他容器（如 std::deque 或 std::vector）实现，但提供了特定的接口（如栈的 push 和 pop 操作）。

**基于现有容器**:

* 容器适配器是基于其他容器实现的，例如：
  + std::stack 默认基于 std::deque。
  + std::queue 默认基于 std::deque。
  + std::priority\_queue 默认基于 std::vector

库容器是保存给定类型的对象的模板类型。在顺序容器中，元素按位置排序和访问。顺序容器共享一个公共的标准化接口：如果两个顺序容器提供一个特定的操作，那么这个操作对两个容器来说具有相同的接口和含义。

所有的容器（除了数组）都提供了有效的动态内存管理。我们可以将元素添加到容器中，而不必担心将元素存储在哪里。容器本身管理其存储。vector和string都通过它们的reserve和capacity成员对内存管理提供更详细的控制。

在大多数情况下，容器定义的操作少得惊人。容器定义构造函数、添加或删除元素的操作、确定容器大小的操作以及返回特定元素的迭代器的操作。其他有用的操作，如排序或搜索，不是由容器类型定义的，而是由标准算法定义的，我们将在第10章中介绍。

当我们使用容器操作来添加或删除元素时，必须记住这些操作可以使迭代器，指针或对容器中元素的引用无效。许多使迭代器无效的操作，如插入或擦除，返回一个新的迭代器，允许程序员在容器中维护一个位置。使用更改容器大小的容器操作的循环在使用迭代器、指针和引用时应特别小心。