二、C++ 标准库

8. IO库

C++不直接处理输入输出，而是通过一组定义在标准库中的类型来处理IO。这些类型支持从设备读取数据、向设备写入数据的 IO 操作，设备可以是文件、控制台窗口等。还有一些类型允许内存 IO，即，从 string 读取数据，向 string 写入数据。

IO 库定义了读写内置类型值的操作。此外，一些类如 string，也会定义类似 IO 操作，来读写自己的对象。

本章介绍 IO 库的基本内容。14章介绍如何编写自己的输入输出运算符，17章介绍如何控制输出格式以及如何对文件进行随机访问。

8.1 IO类

之前使用的IO类型和对象都是操纵char数据的。默认下，这些对象都是关联到用户的控制台窗口的。不能限制应用仅从控制台窗口进行IO操作，应用常常需要读写命名文件。而且，使用IO操作处理string中的字符会很方便。此外应用还可能读写需要宽字符支持的语言。

为了支持这些不同种类的IO处理操作，在istream和ostream之外，标准库还定义了其他一些IO类型。表8.1列出了这些类型，分别定义在三个独立的头文件中：iostream用于读写流的基本类型，fstream定义了读写命名文件的类型，sstream定义了读写内存string对象的类型。

表8.1：IO库类型和头文件

头文件	类型
iostream	istream, wistream 从流读取数据 ostream，wostream 向流写入数据 iostream，wiostream 读写流
fstream	ifstream，wifstream 从文件读取数据 ofstream，wofstream fstream，wfstream
sstream	istringstream, wistringstream 从string读取数据 ostringstream, wostringstream 向string写入数据 stringstream, wstringstream 读写string

> 以下标准库流特性可以无差别应用于普通流、文件流和string流，以及char或宽字符流版本

8.1.1 IO对象无拷贝或赋值

不能拷贝或对IO对象赋值。因此我们也不能将形参或返回类型设置成流类型。进行IO操作的函数通常以引用方式进行传递和返回流。读写一个IO对象会改变其状态，因此传递和返回的引用不能是const的。

8.1.2 条件状态

IO可能发生错误。一些错误是可恢复的，而其他错误则发生在系统深处，已经超出了应用程序可以修正的范围。表8.2列出了IO类所定义的一些函数和标志，可以帮助我们访问和操纵流的条件状态（condition state）。

表8.2：IO库条件状态

strm::iostate	strm是一种IO类型。iostate是一种机器相关的类型，提供了表达式条件状态的完整功能
strm::badbit	strm::badbit 用来指出流已崩溃
strm::failbit	strm::failbit 用来指出一个IO操作失败了
strm::eofbit	用来指出流到达了文件结束
str::goodbit	用来指出流未处于错误状态。此值保证为零
s.eof()	若流s的eofbit置位，则返回true
s.fail()	若流s的failbit或badbit置位，则返回true
s.bad()	若流s的badbit置位，则返回true
s.good()	若流处于有效状态，则返回true
s.clear()	
s.clear(flags)	
s.setstate(flags)	
s.rdstate()	

一个流一旦发生错误，其上后续的IO操作都会失败。只有当一个流处于无错状态时，才可以从它读取数据，向它写入数据。由于流可能处于错误状态，因此代码通常应该在使用一个流之前检查它是否处于良好状态。确定一个流对象的状态的最简单的方法是将它当作一个条件来使用：

```
1 while (cin>>word)
2     //ok: 读操作成功
```

while 循环检查>>表达式返回的流的状态。如果输入操作成功，流保持有效状态，则条件为真。

查询流的状态

将流作为条件使用，只能告诉我们流是否有效，无法知道具体发生了什么。有时需要知道流为什么失败。例如，在键入文件结束标识后我们的应对措施，可能与遇到一个IO设备错误的处理方式是不同的。

IO库定义了一个与机器无关的iostate类型，它提供了表达流状态的完整功能。这个类型作为一个位集合来使用。IO 库定义了4个iostate类型的constexpr值，表示特定的位模式。这些值用来表示特定类型的IO条件，可以与位运算符一起使用来一次性检测或设置多个标志位。

badbit表示系统级错误，如不可恢复的读写错误。

管理条件状态

8.1.3 管理输出缓冲

每个输出流都管理一个缓冲区，用来保存程序读写的数据。

```
1 os << "please enter a value:";
```

文本串可能立即打印出来，也有可能被操作系统保存在缓冲区中，随后再打印。有了缓冲机制，操作系统就可以将程序的多个输出操作组合为单一的设备写操作可以带来很大的性能提升。

导致缓冲刷新（数据真正写到输出设备或文件）的原因有很多：

- 程序正常结束，作为 main函数的 return操作的一部分，缓冲刷新被执行
- 缓冲区满，需要刷新缓冲，而后新的数据才能写入到缓冲区
- 可以使用操纵符如 endl来显式刷新缓冲区

- 每个输出操作之后，可以用操纵符unitbuf设置流的内部状态，来清空缓冲区。默认下，对 cerrr是设置unitbuf的，因此写到cerr的内容都是立即刷新的。
- 一个输出流可能被关联到另一个流。这种情况下，当读写被关联的流时，关联到的流的缓冲区会被刷新。例如，默认下，cin和cerr都关联到cout。因此，读cin或写cerr都会导致cout的缓冲区被刷新。

刷新输出缓冲区

使用过操纵符 endl，它完成换行并刷新缓冲区的工作。IO库还有两个类似的操纵符：flush和ends。flush刷新缓冲区，但不输出任何额外的字符；ends向缓冲区插入一个空字符，然后刷新缓冲区。

unitbuf 操纵符

每次输出操作后都刷新缓冲区，可以使用unitbuf操纵符。它告诉流在接下来的每次写操作之后都进行一次flush操作。而nounitbuf操纵符则重置流，使其恢复使用正常的系统管理的缓冲区刷新机制：

```
1 cout << unibuf;          // 所有输出操作后都会立即刷新缓冲区
2 // 任何输出都立即刷新，无缓冲
3 cout << nounitbuf;       // 回到正常的缓冲方式
```

> **注意：如果程序崩溃，输出缓冲区不会被刷新**
>
> 如果程序异常终止，输出缓冲区是不会被刷新的。当一个程序崩溃后，它所输出的数据很可能停留在输出缓冲区中等待打印。
>
> 当调试一个已经崩溃的程序时，需要确认那些你认为已经输出的数据确实已经刷新了，否则，可能将大量时间浪费在追踪代码为什么没有执行上，而实际上代码已经执行了，只是程序崩溃后缓冲区没有被刷新，输出数据被挂起没有打印而已。

关联输入和输出流

当一个输入流被关联到一个输出流时，任何试图从输入流读取数据的操作都会先刷新关联的输出流。标准库将cout和cin关联在一起，

`cin >> ival;` 导致cout的缓冲区被刷新。

> 交互式系统通常应该关联输入流和输出流。意味着所有输出，包括用户提示信息，都会在读操作之前被打印出来。

8.2 文件输入输出

头文件 fstream 定义了三个类型来支持文件IO：ifsteam、ofstream、fstream。

除了继承自 iostream 类型的行为外，fstream中定义的类型还增加了新的成员来管理与流关联的文件。

表8.3：fstream 特有的操作	
fstream fstrm;	创建一个未绑定的文件流
fstream fstrm(s);	创建一个fstream，并打开名为s的文件。s可以是string类型，或者是一个指向C风格字符串的指针。这些构造函数都是explicit的。默认的文件模式mode依赖于fstream类型
fstream fstrm(s,mode);	
fstrm.open(s)	打开名为s的文件，并将文件与fstrm绑定。
fstrm.close()	关闭与 fstrm绑定的文件，返回void
fstrm.is_open()	

8.2.1 使用文件流对象

想要读写一个文件时，可以定义一个文件流对象，并将对象与文件关联起来。每个文件流类都定义了一个名为open的成员函数，它完成了一些系统相关的操作，来定位给定的文件，并视情况打开为读或写模式。

创建文件流时，可以提供文件名，则open会被自动调用。

用fstream代替iostream&

要求使用基类型对象的地方，可以用继承类型的对象替代。意味着，接受一个 iostream类型引用（或指针）参数的函数，可以用一个对应fstream（或sstream）类型来调用。

成员函数 open 和 close

```
1 ofstream out;
2 out.open(ifile + ".copy");
```

如果调用open失败，failbit会被置位。因为调用open可能失败，进行open是否成功的检测是一个好习惯。

```
1 if (out)
```

一个文件流已经打开，它就保持与对应文件的关联。对一个已经打开的文件流调用open会失败，并导致failbit被置位。要将文件流关联到另外一个文件，必须先关闭已经关联的文件。

自动构造和析构

当一个 fstream 对象离开其作用域时（该对象会被销毁），与之关联的文件会自动关闭

8.2.2 文件模式

每个流都有一个关联的文件模式，用来指出如何使用文件。

表8.4：文件模式

int	以读方式打开
out	以写方法打开
app	每次写操作前均定位到文件末尾
ate	打开文件后立即定位到文件末尾
trunc	截断文件
binary	二进制方式进行IO

指定文件模式有限制

以out模式打开文件会丢弃已有数据

> 保留被 ostream 打开的文件中已有数据的唯一方法是显示指定 app 或 in 模式

每次调用open时都会确定文件模式

8.3 string流

sstream 头文件定义了三个类型来支持内存IO，istringstream（从string读取数据），ostringstream（向string写入数据），stringstream（从string读写数据）。就像 string是一个IO流一样。

表8.5：stringstream特有的操作	
sstream strm;	
sstream strm(s);	
strm.str()	
strm.str(s)	

8.3.1 使用 istringstream

当某些工作是对整行文本进行处理，而其他一些工作是处理行内的单个单词时，通常可以使用istringstream。

8.3.2 使用 ostringstream

逐行构造输出，最后一起打印时，ostringstream是很有用的。

小结

C++使用标准库来处理面向流的输入和输出：

- iostream 处理控制台IO
- fstream 处理命名文件IO
- stringstream 完成内存string的IO

术语

9. 顺序容器

本章是第 3 章的扩展。在第 11 章介绍关联容器特有的操作。

所有容器类都共享公共的接口，不同容器按不同方式对其进行扩展。这个公共接口使容器的学习更加容易。每种容器提供了不同性能和功能的权衡。

一个容器就是一些特定类型对象的集合。顺序容器（sequential container）为程序员提供了控制元素存储和访问顺序的能力。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。与之相

对的，将在 11 章介绍的有序和无序关联容器，则根据关键字的值来存储元素。

标准库还提供了三种容器适配器，分别为容器操作定义了不同的接口，来与容器类型适配。

9.1 顺序容器概述

表9.1列出了标准库中的顺序容器，所有顺序容器都提供了快速顺序访问元素的能力。但是，这些容器在以下方面都有不同的性能折中：

- 向容器添加或从容器中删除元素的代价
- 非顺序访问容器中元素的代价

表9.1：顺序容器类型	
vector	可变大小数组。支持快速随机访问。在尾部之外的位置插入或删除元素可能很慢
deque	双端队列。支持快速随机访问。头尾位置插入、删除速度很快
list	双向链表。只支持双向顺序访问。在list中任何位置插入、删除操作速度都很快
forward_list	单向链表。只支持单向顺序访问。在链表任何位置进行插入、删除操作速度都很快
array	固定大小数组，支持快速随机访问。不能添加或删除元素
string	与 vector 相似的容器，但专门用于保存字符。随机访问快。尾部插入、删除速度快

除了固定大小的 array 外，其他容器都提供高效灵活的内存管理。可以添加和删除元素，扩张和收缩容器的大小。容器保存元素的策略对容器操作的效率有着固有的、重大的影响，某些情况下存储策略影响特定容器是否支持特定操作。

string 和 vector 将元素保存在连续的内存空间中。由于元素是连续存储的，元素下标计算地址非常快。但是，在这两种容器的中间位置添加或删除元素就会非常耗时：再一次插入或删除操作后，需要移动插入或删除位置之后的所有元素，以保持连续存储。而且添加一个元素有时可能需要分配额外的存储空间。这时，每个元素都必须移动到新的存储空间。

list 和 forwar_list 两个容器的设计目的是令容器任何位置的添加和删除操作都很快速。代价是这两个容器不支持元素的随机访问：为了访问一个元素，只能遍历整个容器。而且，与 vector、deque 和 array 相比，这两个容器的额外内存开销很大。

deque 是一个更复杂的数据结构。与 string 和 vecotr 类似，deque 支持快速的随机访问。与 stirng 和 vector 一样，在 deque 的中间位置添加或删除元素的代价很高。但在 deque 两端增删元素都很快。

forward_list 和 array 是新C++标准增加的类型。与内置数组相比，array是种更安全、更容易使用的数组类型。forwar_list 的设计目标是达到与最好的手写单向链表数据结构相当的性能。因此，forward_list 没有 size 操作，因为保存或计算其大小会比手写链表多出额外的开销。对其他容器而言，size 得保证是一个快速的常见时间的操作。

> 尽量使用标准库容器，而不是原始的数据结构，如内置数组。
>
> 新标准库容器的性能几乎与精心优化过的同类数据一样好（通常更好）。

确定使用哪种顺序容器 *

> 通常，使用 vector 是最好的选择，除非有充分的理由选择其他容器。

选择容器的基本原则：

- 除非有很好的理由选择其他容器，否则使用 vector
- 程序有很多小的元素，且空间的额外开销很重要，则不要使用链表 list 或 forward_list
- 要求随机访问元素，使用 vector 或 deque
- 在容器中间插入或删除元素，使用 list 或 forward_list
- 头尾位置插入或删除元素，但不会在中间位置增删，使用 deque
- 只有在读取输入时才需要在容器中间位置插入元素，随后需要随机访问元素，则
  - 首先，确定是否真的需要在容器中间位置添加元素。当处理输入数据时，通常可以很容易地向 vector 追加数据，然后再调用标准库的 sort 函数来重排容器中的元素，从而避免在中间位置添加元素
  - 如果必须在中间位置插入元素，考虑在输入阶段使用 list，一旦输入完成，将list中的内容拷贝到一个vector中

既要随机访问元素，又要在容器中间位置插入元素，怎么办？取决于 list 或 forward_list 中访问元素与 vector 或 deque 中插入、删除元素的相对性能。一般，应用中占主导地位的操作（执行的访问操作更多还是插入、删除更多）决定了容器类型的选择。这时，需要对两种容器分别测试应用的性能。

> 如果不确定使用哪种容器，可以在程序中只使用 vector 和 list 公共的操作：使用迭代器，不使用下标操作，避免随机访问。这样，在必要时选择使用 vector 和 list 都很方便。

9.2 容器库概览

容器类型上的操作：

- 某些操作是所有容器类型都提供的（表9.2）
- 一些操作仅针对顺序容器、关联容器或无序容器
- 一些操作只适用于小部分容器

介绍对所有容器都适用的操作，剩余部分聚焦于仅适用于顺序容器的操作。关联容器特有的操作在 11 章介绍。

对容器可以保存的元素类型的限制

表9.2：容器操作

类型别名	
iterator	此容器类型的迭代器类型
const_iterator	可以读取元素，但不能修改元素的迭代器类型
size_type	无符号整型，保存此种容器类型最大可能容器的大小
difference_type	带符号整数类型，保存两个迭代器之间的距离
value_type	元素类型
reference	元素的左值类型；与 value_type& 含义相同
const_reference	元素的 const 左值类型（即 const value_type&）
构造函数	
C c;	默认构造函数，构造空容器
C c1(c2);	构造 c2 的拷贝 c1
C c(b, e);	构造 c，将迭代器 b 和 e 指定范围内的元素拷贝到 c （array 不支持）
C c{a, b, c...};	列表初始化 c
赋值与 swap	
c1 = c2	
c1 = {a, b, c...}	将 c1 中的元素替换为列表中元素（不适用于 array）
a.swap(b)	交换 a b 的元素
swap(a,b)	
大小	

c.size()	不支持 forward_list
c.max_size()	可保存的最大元素数目
c.empty()	
添加/删除元素（不适用于 array）	在不同容器中，这些操作的接口都不同
c.insert(args)	将 args 中的元素拷贝进 c
c.emplace(inits)	使用 inits 构造 c 中的一个元素
c.erase(args)	删除 args 指定的元素
c.clear()	删除 c 中的所有元素
关系运算符	
==, !=	
<, <=, >, >=	（无序关联容器不支持，如 list 链表）
获取迭代器	
c.begin(), c.end()	返回指向 c 的首元素和尾元素之后位置的迭代器
c.cbegin(), c.cend()	返回 const_iterator
反向容器的额外成员	**（不支持 forward_list）**
reverse_iterator	按逆序寻址元素的迭代器
const_reverse_iterator	不能修改元素的逆序迭代器
c.rbegin(), c.rend()	返回指向 c 的尾元素和首元素之前位置的迭代器
c.crbegin(), c.crend()	返回 const_reverse_iterator

9.2.1 迭代器

与容器一样，迭代器有着公共的接口：如果一个迭代器提供某种操作，那么所有提供相同操作的迭代器对这个操作的实现方式都是相同的。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，而所有迭代器都是通过解引用运算符来实现这个操作的。类似，标准库容器的所有迭代器都定义了递增运算符，从当前元素移动到下一个元素。

表3.6列出了容器迭代器支持的所有操作，有一个不符合公共接口特点——forward_list 迭代器不支持递减运算符（--）。表3.7列出了迭代器支持的算术运算，只能应用于 string、vector、deque 和 array 的迭代器。

迭代器范围

> 迭代器范围的概念是标准库的基础

一个迭代器范围（iterator range）由一对迭代器表示，这两个迭代器通常为 begin 和 end，或者是 first 和 last，它们标记了容器中元素的一个范围。

这种元素范围称为左闭合区间（left-inclusive interval），其标准数学描述为 [begin, end)

end可以与 begin指向相同的位置，但不能指向 begin 之前的位置。

使用左闭合范围蕴含的编程假定

标准库使用左闭合范围是因为它有三种方便的性质。

- begin 与 end 相等，则范围为空
- begin 与 end 不等，则范围至少包含一个元素，且 begin 指向该范围中的第一个元素
- 可以对 begin 递增若干次，使得 begin == end

这些性质让我们可以用一个循环来处理一个元素的范围，这是安全的

```
while (begin != end) {
    *begin = val;    // 正确：范围非空，因此 begin 指向一个元素
    ++begin;
}
```

给定构成一个合法范围的迭代器 begin 和 end，若范围不为空，begin指向非空范围的一个元素，因此在 while 循环体中，可以安全解引用 begin，因为它必然指向一个元素。

9.2.2 容器类型成员

每个容器都定义了多个类型，已经使用过 size_type、iterator 和 const_iterator。

大多数容器还提供反向迭代器。

类型别名。通过类型别名，我们可以在不了解容器中元素类型的情况下使用它。需要元素类型，可以使用容器的 value_type。需要元素类型的一个引用，可以使用 reference 或 const_reference。类型别名在泛型编程中非常有用。16 章介绍。

9.2.3 begin和end成员

> 当不需要写访问时，应使用 cbegin 和 cend

9.2.4 容器定义和初始化

将一个容器初始化为另一个容器的拷贝

可以直接拷贝整个容器，或者（array除外）拷贝有一个迭代器对指定的元素范围。

列表初始化

与顺序容器大小相关的构造函数

> 只有顺序容器的构造函数才接受大小参数，关联容器并不支持

标准库 array 具有固定大小

9.2.5 赋值和swap

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。

与内置数组不同，标准库 array 类型允许赋值。由于右边运算对象的大小可能与左边运算对象的大小不同，因此 array 类型不支持 assign，也不允许用花括号包围的值列表进行赋值。

> 赋值相关运算会导致指向左边容器内部的迭代器、引用和指针失效。而 swap 操作将容器内容交换不会导致指向容器的迭代器、引用和指针失效（容器类型为 array 和 string 的情况除外）

使用 assign（仅顺序容器）

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。

> 由于其旧元素被替换，因此传递给 assign 的迭代器不能指向调用 assign 的容器

使用 swap

swap 操作交换两个相同类型容器的内容。调用 swap 之后，两个容器中的元素将会交换：

```
1 vector<string> svec1(10);
2 vector<string> svec2(24);
```

```
swap(svec1, svec2);
```

除 array 外，交换两个容器内容的操作很快——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。

> 除array 外，swap 不对任何元素进行拷贝、删除或插入操作，可以保证在常数时间内完成。

元素不会被移动意味着，除 string 外，指向容器的迭代器、引用和指针在 swap 操作之后都不会失效。它们仍指向 swap 操作之前所指向的那些元素。但是，在 swap 之后，这些元素已经属于不同的容器了。

与其他容器不同，对一个 string 调用 swap 会导致迭代器、引用和指针失效。

swap 两个 array 会真正交换它们的元素，所需的时间与 array 中元素的数目成正比。对于 array，在 swap 操作之后，指针、引用和迭代器所绑定的元素保持不变，但元素值已经与另一个 array 中对应元素的值进行了交换。

非成员版本的 swap 在泛型编程中是非常重要的。建议统一使用非成员版本的 swap。

9.2.6 容器大小操作

每个容器都有三个与大小相关的操作：

- 成员函数 size 返回容器中元素的数目
- empty 当 size 为 0 时返回布尔值 true，否则返回 false
- max_size 返回一个大于或等于该类型容器所能容纳的最大元素数的值

forwar_list 支持 max_size 和 empty，但不支持 size。

9.2.7 关系运算符

每个容器类型都支持相等运算符（== 和 !=）；除了无需关联容器外的所有容器都支持关系运算符（>、>=、<、<=）。

比较两个容器实际上进行元素的逐对比较。

容器的关系运算符使用元素的关系运算符完成比较

> 只有当其元素类型也定义了相应的比较运算符时，才可以使用关系运算符来比较两个容器。

9.3 顺序容器操作

顺序容器和关联容器的不同之处在于两者组织元素的方式。这些不同直接关系到元素如何存储、访问、添加和删除。本章剩余部分介绍顺序容器特有的操作。

9.3.1 向顺序容器添加元素

除 array 外，所有标准库容器都提供灵活的内存管理。在运行时可以动态添加或删除元素来改变容器大小。

表9.5：向顺序容器添加元素的操作

这些操作会改变容器的大小；array 不支持这些操作 forward_list 有自己专有版本的 insert 和 emplace forward_list 不支持 push_back 和 emplace_back vector 和 string 不支持 push_front 和 emplace_front	
c.push_back(t) c.emplace_back(args)	在 c 的尾部创建一个值为 t 或由 args 创建的元素。返回 void
c.push_front(t) c.enplace_front(args)	在 c 的头部创建一个值为 t 或由 args 创建的元素。返回 void
c.insert(p,t) c.emplace(p, args)	在迭代器 p 指向的元素之前创建一个值为 t 或由 args 创建的元素。返回指向新添加的元素的迭代器
c.insert(p, n, t)	在迭代器 p 指向的元素之前插入 n 个值为 t 的元素。返回指向新添加的第一个元素的迭代器
c.insert(p, b, e)	将迭代器 b 和 e 指定范围内的元素插入到迭代器 p 指向的元素之前。b 和 e 不能指向 c 中的元素。返回指向新添加的第一个元素的迭代器；若范围为空，则返回 p
c.insert(p, il)	il 是一个花括号包围的元素值列表。将这些给定值插入到迭代器 p 指向的元素之前。返回指向新添加的第一个元素的迭代器；若列表为空，则返回 p
向一个 vector、string 或 deque 插入元素会使所有指向容器的迭代器、引用和指针失效	

使用这些操作时，必须记得不同容器使用不同的策略来分配元素空间，而这些策略直接影响性能。在一个 vector 或 string 的尾部之外的任何位置，或是一个 deque 的首尾之外的任何位置添加元素，都需要移动端元素。而且，向一个 vector 或 string 添加元素可能引起整个对象存储空间的重新分配。重新分配一个对象的存储空间需要分配新的内存，并将元素从旧的空间移动到新的空间。

使用 push_back

除 array 和 forwar_list 之外，每个顺序容器（包括 string）都支持 push_back。

> 容器元素是拷贝

使用 push_front

list、forward_list、deque 容器支持 push_front 操作。

在容器中的特定位置添加元素

insert 提供了更一般的添加功能，允许我们在容器中任意位置插入 0 个或多个元。vector、deque、list 和 string 都支持 insert。forward_list 提供了特殊版本的 insert。

> 将元素插入到 vector、deque 和 string 中的任何位置都是合法的，但是这样做可能很耗时。

插入范围内元素

使用 insert 的返回值

使用 emplace 操作

新标准引入了三个新成员——emplace_front、emplace 和 emplace_back，这些操作是构造而不是拷贝元素，分别对应 push_front、insert 和 push_back。

当调用 push 或 insert 成员函数时，我们将元素类型的对象传递给它们，这些对象被拷贝到容器中。当调用一个 emplace 成员函数时，将参数传递给元素类型的构造函数。emplace 成员使用这些参数在容器管理的内存空间中直接构造元素。

> emplace 函数在容器中直接构造函数。传递给 emplace 函数的参数必须与元素类型的构造函数相匹配。

9.3.2 访问元素

表9.6：在顺序容器中访问元素的操作

at 和 下标操作只适用于 string、vector、deque 和 array back 不适用于 forward_list	
c.back()	返回 c 中尾元素的引用。若 c 为空，函数行为 undefined
c.front()	返回 c 中首元素的引用
c[n]	

	返回 c 中下标为 n 的元素的引用，n 是一个无符号整数。若 n >= c.size()，则函数行为 undefined
c.at(n)	返回下标为 n 的元素的引用。如果下标越界，则抛出一 out_of_range 异常

> 对一个空容器调用front 和 back，就像使用一个越界的下标一样。

访问成员函数返回的是引用

下表操作和安全的随机访问

9.3.3 删除元素

与添加元素的多种方式类似，容器也有多种删除元素的方式。

表9.7：顺序容器的删除操作	
这些操作会改变容器的大小，不适用于 array forward_list 有特殊版本的 erase forward_list 不支持 pop_back；vector 和 string 不支持 pop_front	
c.pop_back()	
c.pop_front()	
c.erase(p)	
c.erase(b,e)	
c.clear()	

> 删除 deque 中除首尾位置之外的任何元素都会使所有迭代器、引用和指针失效。指向 vector 和 string 中删除点之后位置的迭代器、引用和指针都会失效。

pop_front 和 pop_back 成员函数

从容器内部删除一个元素

删除多个元素

9.3.4 特殊的 forward_list 操作

9.3.5 改变容器大小

> 如果 resize 缩小容器，则指向被删除元素的迭代器、引用和指针都会失效；对 vector、string 或 deque 进行 resize 可能导致迭代器、指针和引用失效。

9.3.6 容器操作可能使迭代器失效

向容器中添加和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。使用失效的指针、引用或迭代器是一个严重的程序设计错误，很可能引起与使用未初始化指针一样的问题。

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、指针和引用都会失效。
- 对于 deque，插入到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。

从一个容器删除元素后：

- 对于 list 和 forward_list，指向容器其他位置的迭代器、引用和指针仍有效
- 对于 deque，如果在首尾之外的任何位置删除元素，那么指向被删除元素外其他元素的迭代器、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，指向被删元素之前元素的迭代器、引用和指针仍有效。尾后迭代器失效。

编写改变容器的循环程序

添加、删除 vecotr、string 或 deque 元素的循环程序必须考虑迭代器、引用和指针可能失效的问题。程序必须保证每个循环步中都更新迭代器、引用或指针。

不要保存 end 返回的迭代器

> 如果在一个循环中插入、删除 deque、string 或 vector 中的元素，不要缓存 end 返回的迭代器

9.4 vector 对象是如何增长的

标准库实现者采用了可以减少容器空间重新分配次数的策略。当不得不获取新的内存空间时，vector 和 string 的实现通常会分配比新的空间需求更大的内存空间。容器预留这些空间作为备用，可用来保存更多的新元素。不需要每次添加新元素都重新分配容器的内存空间了。

虽然 vector 在每次重新分配内存空间时都要移动所有元素，但使用此策略后，其扩张操作通常比 list 和 deque 还要快。

管理容量的成员函数

表9.10：容器大小管理操作	
shrink_to_fit 只适用于 vector、string 和 deque capacity 和 reserve 只适用于 vector 和 string	
c.shrink_to_fit	请将 capacity() 减少为与 size() 相同大小
c.capacity()	不重新分配内存空间的话，c 可以保存多少元素
c.reserve(n)	分配至少能容纳 n 个元素的内存空间

shrink 指出不再需要任何多余的内存空间。但是具体实现可以忽略此请求。它并不保证一定退回内存空间。

capacity 和 size

size 是指容器已经保存的元素的数目；capacity 是在不分配新的内存空间下它最多可以保存多少元素。

9.5 额外的 string 操作

9.5.1 构造string的其他方法

substr 操作

9.5.2 改变string的其他方法

append 和 replace 函数

改变 string 的多种重载函数

9.5.3 string搜索操作

指定在哪里开始搜索

逆向搜索

9.5.4 compare函数

9.5.5 数值转换

9.6 容器适配器

除了顺序容器外，标准库还定义了三个顺序容器适配器：stack、queue 和 priority_queue。

适配器（adaptor）是标准库中的一个通用概念。容器、迭代器和函数都有适配器。本质上，一个适配器是一种机制，能使某种事物的行为看起来像另外一种事物一样。一个容器适配器接受一个已有的容器类型，使其行为看起来像一种不同的类型。例如，stack 适配器接受一个顺序容器（除 array 或 forwar_list 外），并使其操作起来像一个 stack 一样。

定义一个适配器

栈适配器

队列适配器

小结

标准库是模板类型，用来保存给定类型的对象。在一个顺序容器中，元素是按顺序存放的，通过位置来访问。顺序容器有公共的标准接口：如果两个顺序容器都提供一个特定的操作，那么这个操作在两个容器中具有相同的接口和含义。

所有容器（除 array 外）都提供高效的动态内存管理。我们可以向容器中添加元素而不必担心元素存储值在哪里。容器负责管理自身的存储。vector 和 string 都提供更细致的内存管理机制，这是通过它们的 reserve 和 capacity 成员函数来实现的。

很大程度上，容器只定义了极少的操作。每个容器都定义了构造函数、添加和删除元素的操作、确定容器大小的操作以及返回指向特定元素的迭代器的操作。其他一些有用的操作，如排序或搜索，并不是由容器类型定义的，而是由标准库算法实现的，我们将在第 10 章介绍这些内容。

当我们使用添加和删除元素的容器操作时，必须注意这些操作可能使指向容器中元素的迭代器、指针或引用失效。很多会使迭代器失效的操作，如 insert 和 erase，都会返回一个新的迭代器，帮助程序员维护容器中的位置。如果循环程序中使用了改变容器大小的操作，就要尤其小心其中迭代器、指针和引用的使用。

术语

10. 泛型算法

标准库容器定义的操作集合很小。标准库没有给每个容器添加大量功能，而是提供了一组算法，这些算法大多数都独立于任何特定的容器。这些算法是通用的（generic，或称泛型的）：它们可用于不同类型的容器和不同类型的元素。

顺序容器只定义了很少的操作：多数情况下，可以添加和删除元素、访问首尾元素、确定容器是否为空以及获得指向首元素或尾元素之后位置的迭代器。

用户可能还希望做其他很多有用的操作：查找特定元素、替换或删除一个特定值、重排元素顺序等。

标准库没有给每个容器都定义成员函数来实现这些操作，而是定义了一组泛型算法（generic algorithm）：称 “算法”，是因为它们实现了一些经典算法的公共接口，如排序和搜索；称 “泛型”，是因为它们可以用于不同类型的元素和多种容器类型（不仅包括标准库类型，如 vector 和 list，还包括内置的数组类型），还能用于其他类型的序列。

10.1 概述

一般情况下，这些算法不直接操作容器，而是遍历由两个迭代器指定的一个元素范围来进行操作。

```
1 int val = 42;
2 auto result = find(vec.cbegin(), vec.cend(), val);
3 cout << "The value" << val
4      << (result == vec.end() ? " is not present" : "is present") << endl;
```

find 将范围中每个元素与给定值进行比较。它返回指向第一个等于给定值的元素的迭代器。如果无匹配元素，则返回第二个参数来表示搜索失败。

算法如何工作

为了弄清楚这些算法如何用于不同类型的容器，观察一下 find。find 的工作是在一个未排序的元素序列中查找一个特定元素。

。。。

这些步骤都不依赖于容器所保存到元素类型。因此，只要有一个迭代器可用来访问元素，find 就完全不依赖于容器类型。

迭代器令算法不依赖于容器，...

...，但算法依赖于元素类型的操作

虽然迭代器的使用令算法不依赖于容器类型，但大多数算法都使用了一个（或多个）元素类型上的操作。find 用元素类型的 == 运算符完成每个元素与给定值的比较。大多数算法提供了一种方法，允许使用自定义的操作来代替默认的运算符。

> 算法永远不会执行容器的操作，它们只会运行于迭代器之上，执行迭代器的操作。
>
> 带来的编程假定：算法永远不会改变底层容器的大小。算法可能改变容器中保存的元素的值，也可能在容器内移动元素，但永远不会直接添加或删除元素。
>
> 10.4.1节，标准库定义了一类特殊的迭代器，称插入器。给这类迭代器赋值时，它们会在底层的容器上执行插入操作。当算法操作这样的迭代器时，迭代器可以完成向容器添加元素的效果，但算法自身不会做这样的操作。

10.2 初始泛型算法

与容器类似，这些算法有一致的结构。

除了少数例外，标准库算法都对一个范围内的元素进行操作。

虽然大多数算法遍历输入范围的方式相似，但它们使用范围中元素的方式不同。理解算法的最基本的方法就是了解它们是否读取元素、改变元素或是重排元素顺序。

10.2.1 只读算法

算法和元素类型

> 对于只读取而不改变元素的算法，通常最好使用 cbegin() 和 cend()。如果使用算法返回的迭代器来改变元素的值，则使用 begin() 和 end()。

操作两个序列的算法

10.2.2 写容器元素的算法

算法不检查写操作

> 向目的位置迭代器写入数据的算法假定目的位置足够大，能容纳要写入的元素。

介绍 back_insert

一种保证算法有足够元素空间来容纳输出数据的方法是使用插入迭代器（insert iterator）。插入迭代器是一种向容器中添加元素的迭代器。

拷贝算法

10.2.3 重排容器元素的算法

某些算法会重排容器中元素的顺序。调用 sort 会重排输入序列中的元素，使之有序，它是利用元素类型的 < 运算符来实现排序的。

分析一系列儿童故事中所用的词汇。已有一个 vector，保存了多个故事的文本。希望简化这个 vector，使得每个单词只出现一次，而不管单词在任意给定文档中到底出现了多少次。

```
// 输入
the quick red fox jumps over the slow red turtle
// 程序应该生成如下 vector
fox jumps over quick red slow the turtle
```

消除重复单词

为了消除重复单词，首先将 vector 排序，使得重复的单词都相邻出现。一旦 vector 排序完毕，可以使用 unique 算法来重排 vector，使得不重复的元素出现在 vector 的开始部分。由于算法不能执行容器的操作，将使用 vector 的 erase 成员来完成真正的删除操作：

```
void elimDups(vector<string> &words)
{
    // 按字典排序words，以便查找重复单词
    sort(words.begin(), words.end());
    // unique 重排输入范围，使得每个单词只出现一次
    // 排列在范围的前部，返回指向不重复区域之后一个位置的迭代器
    auto end_unique = unique(words.begin(), words.end());
    // 使用向量操作 erase 删除重复单词
    words.erase(end_unique, words.end());
}
```

使用 unique

使用容器操作删除元素

10.3 定制操作

很多算法都会比较输入序列中的元素。默认这类算法使用元素类型的 < 或 == 运算符完成比较。标准库还为这些算法定义了额外的版本，允许我们提供自己定义的操作来代替默认运算符。

例如，sort 算法默认使用元素类型的 < 运算符。但可能我们希望的排序顺序与 < 所定义的顺序不同，或是我们的序列可能保存的是未定义 < 运算符的元素类型（如 Sales_data）。这两种情况下，都需要重载 sort 的默认行为。

10.3.1 向算法传递函数

举例，希望在调用 elimDups（10.2.3节）后打印 vector 的内容。此外还希望单词按其长度排序，大小相同的再按字典序排序。为了按长度重排 vector，将使用 sort 的第二个版本，此版本是重载过的，它接受第三个参数，此参数是一个谓词（predicate）。

谓词（predicate）

谓词是一个可调用的表达式，其返回结果是一个能用做条件的值。标准库算法所使用的谓词分为两类：一元谓词（unary predicate，只接受单一参数）和二元谓词（binary predicate，它们有两个参数）。接受谓词参数的算法对输入序列中的元素调用谓词，因此，元素类型必须能转换为谓词的参数类型。

```
1 // 比较函数，用来按长度排序单词
2 bool isShorter(const string &s1, const string &s2)
3 {
4     return s1.size() < s2.size();
5 }
6
7 // 按长度由短至长排序 words
8 sort(words.begin(), words.end(), isShorter);
```

排序算法

在将 words 按大小重排的同时，还希望具有相同长度的元素按字典序排列。为了保持相同长度的单词按字典序排序，可以使用 stable_sort 算法。这种稳定排序算法维持相等元素的原有顺序。

10.3.2 lambda表达式

有时希望进行的操作需要更多参数，超出了算法对谓词的限制。

介绍 lambda

可以向一个算法传递任何类别的可调用对象（callable object）。对于一个对象或一个表达式，如果可以对其使用调用运算符，则称它为可调用的。

使用过的两种可调用对象是函数和函数指针。还有两种可调用对象：重载了函数调用运算符的类（14.8节），以及 lambda 表达式（lambda expression）。

一个 lambda 表达式表示一个可调用的代码单元。可以将其理解为一个未命名的内联函数。与任何函数类似，一个 lambda 具有一个返回类型、一个参数列表和一个函数体。与函数不同，lambda 可能定义在函数内部。一个 lambda 表达式具有如下形式：

```
1 [capture list] (parameter list) -> return type { function body }
```

可忽略参数列表和返回类型，但必须包含捕获列表和函数体。

向 lambda 传递参数

编写一个与 isShorter 函数完成相同功能的 lambda来调用 stable_sort:

```
1 stable_sort(words.begin(), words.end(),
2             [](const string &a, const string &b)
3             { return a.size() < b.size(); });
```

使用捕获列表

> 一个 lambda 只有在其捕获列表中捕获一个它所在函数中的局部变量，才能在函数体中使用该变量

调用 find_if

for_each 算法

完整的 biggies

10.3.3 lambda捕获和返回

当定义一个lambda时，编译器生成一个与 lambda 对应的新的（未命名的）类类型。将在14.8.1节介绍这种类是如何生成的。当向一个函数传递一个 lambda 时，同时定义了一个新类型和该类型的一个对象：传递的参数就是此编译器生成的类类型的未命名对象。类似，当使用 auto 定义一个用 lambda 初始化的变量时，定义了一个从 lambda 生成的类型的对象。

默认情况下，从 lambda 生成的类都包含一个对应该 lambda 所捕获的变量的数据成员。lambda 数据成员在lambda 对象创建时被初始化。

值捕获

类似参数传递，变量的捕获方式可以是值或引用。采用值捕获的前提是变量可以拷贝。与参数不同，被捕获的变量的值是在 lambda 创建时拷贝，而不是调用时拷贝。

引用捕获

如果采用引用方式捕获一个变量，就必须确保被引用的对象在 lambda 执行的时候是存在的。

> 尽量保持 lambda 的变量捕获简单化

隐式捕获

可以让编译器根据lambda 体中的带am来推断我们要使用哪些变量，在捕获列表中写一个&或=，告诉编译器采用引用捕获还是值捕获。

可变 lambda

指定 lambda 返回类型

10.3.4 参数绑定

标准库 bind 函数

绑定 check_size 的 sz 参数

使用 placeholders 名字

bind 的参数

用 bind 重排参数顺序

绑定引用参数

10.4 再探迭代器

除了为每个容器定义的迭代器外，标准库谁来头文件 iterator 中还定义了额外几种迭代器。

- 插入迭代器（insert iterator）：这些迭代器被绑定到一个容器上，可用来向容器插入元素
- 流迭代器（stream iterator）：这些迭代器被绑定到输入或输出流上，可用来遍历所关联的 IO 流
- 反向迭代器（reverse iterator）：这些迭代器向后而不是向前移动。除了 forward_list 之外的标准库容器都有反向迭代器
- 移动迭代器（move iterator）：这些专用的迭代器不是拷贝其中的元素，而是移动它们。13.6.2节

10.4.1 插入迭代器

10.4.2 iostream迭代器

istream_iterator 操作

使用算法操作流迭代器

istream_iterator 允许使用懒惰求值

ostream_iterator 操作

使用流迭代器处理类类型

10.4.3 反向迭代器

反向迭代器需要递减运算符

反向迭代器和其他迭代器间的关系

10.5 泛型算法结构

任何算法的最基本的特性是它要求其迭代器提供哪些操作。

表 10.5：迭代器类别

输入迭代器	只读，不写；单遍扫描，只能递增
输出迭代器	只写，不读；单遍扫描，只能递增

前向迭代器	可读写；多遍扫描，只能递增
双向迭代器	可读写；多遍扫描，可递增递减
随机访问迭代器	可读写，多遍扫描，支持全部迭代器运算

> 向一个算法传递错误类别的迭代器问题，很多编译器不会给出任何警告或提示

10.5.1 5类迭代器

迭代器类别

算法 sort 要求随机访问迭代器。array、deque、string 和 vector 的迭代器都是随机访问迭代器，用于访问内置数组元素的指针也是。

10.5.2 算法形参模式

接受单个目标迭代器的算法

接受第二个输入序列的算法

10.5.3 算法命名规范

一些算法使用重载形式传递一个谓词

_if 版本的算法

区分拷贝元素的版本和不拷贝的版本

10.6 特定容器算法

splice 成员

链表特有的操作会改变容器

小结

标准库定义了大约 100 个类型无关的对序列进行操作的算法。序列可以是标准库容器类型中的元素、一个内置数组或者是通过读写一个流来生成的。算法通过在迭代器上进行操作来实现类型无关。多数算法接受的前两个参数是一对迭代器，表示一个元素的范围。额外的迭代器参数可能包括一个表示目的位置的输出迭代器，或是表示第二个输入范围的另一个或另一队迭代器。

根据支持的操作不同，迭代器可分为五类：输入、输出、前向、双向以及随机访问迭代器。

算法从不直接改变它们所操作的序列的大小。它们会将元素从一个位置拷贝到另一个位置，但不会直接添加或删除元素。

插入迭代器可以向序列添加元素。

forward_list 和 list 对一些通用算法定义了自己特有的版本。这些链表特有版本会修改给定的链表。

术语

11. 关联容器

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

关联容器支持高校的关键字查找和访问。两个主要的关联容器（associative-container）类似是 map 和 set。map 中的元素是一些关键字-值（key-value）对：关键字起到索引作用，值则表示与索引相关联的数据。set 中每个元素只包含一个车关键字；set 指出高效的关键字查询操作——检查一个给定关键字是否在 set 中。

标准库提供 8 个关联容器，表11.1。这 8 个容器间的不同体现在三个维度上：

1. 每个容器是一个set 或 map
2. 要求不重复的关键字，或允许重复的关键字
3. 按顺序保存元素，或无序保存

表11.1：关联容器类型

按关键字有序保存元素	
map	关联数组，保存关键字-值对
set	关键字即值，只保存关键字的容器
multimap	关键字可重复的map
multiset	关键字可重复的set
无序集合	
unordered_map	用哈希函数组织的 map
unordered_set	用哈希函数组织的 set

unordered_multimap	哈希组织的map，关键字可重复
unordered_multiset	哈希组织的 set，关键字可重复

11.1 使用关联容器

使用 map

使用 set

11.2 关联容器概述

关联容器（有序的和无序的）都支持9.2节普通容器操作。关联容器不支持顺序容器的位置相关的操作，例如 push_fornt或push_back。因为关联容器中元素是根据关键字存储的，这些操作对关联容器没有意义。而且关联容器也不支持构造函数或插入操作这些接受一个元素值和一个数量值的操作。

除了与顺序容器相同的操作之外，关联容器还支持一些顺序容器不支持的操作和类型别名。此外了，无序容器还提供一些用来调整哈希性能的操作。

关联容器的迭代器都是双向的。

11.2.1 定义关联容器

初始化 multimap 或 multiset

11.2.2 关键字类型的要求

有序容器的关键字类型

使用关键字类型的比较函数

11.2.3 pair 类型

创建 pair 对象的函数

11.3 关联容器操作

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

表11.3：关联容器额外的类型别名	
key_type	此容器类型的关键字类型
mapped_type	每个关键字关联的类型；只适用于map
value_type	对于 set，与 key_type 相同 对于 map，为 pair<const key_type, mapped_type>

11.3.1 关联容器迭代器

> 一个 map 的 value_type 是一个 pair，我们可以改变 pair 的值，但不能改变关键字成员的值

set 的迭代器是 const 的

遍历关联容器

关联容器和算法

11.3.2 添加元素

向 map 添加元素

检测 insert 的返回值

展开递增语句

向 multiset 和 multimap 添加元素

11.3.3 删除元素

11.3.4 map 的下标操作

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

使用下标操作的返回值

> 与 vector 和 string 不同，map 的下标运算符返回的类型与解引用 map 迭代器得到的类型不同。对一个 map 进行下标操作时，会获得一个 mapped_type 对象；但当解引用一个 map 迭代器时，会得到一个 value_type 对象。

11.3.5 访问元素

表11.7：在一个关联容器中查找元素的操作

lower_bound 和 upper_bound 不适用于无序容器 下标和 at 操作只适用于非 const 的 map 和 unordered_map	
c.find(k)	返回一个迭代器，指向第一个关键字为 k 的元素，若 k 不在容器中，则返回尾后迭代器
c.count(k)	返回关键字等于 k 的元素的数量。对于不允许重复关键字的容器，返回值永远是 0 或 1
c.lower_bound(k)	返回一个迭代器，指向第一个关键字不小于 k 的元素
c.upper_bound(k)	返回一个迭代器，指向第一个关键字大于 k 的元素
c.equal_range(k)	返回一个迭代器 pair，表示关键字等于 k 的元素的范围。若 k 不存在，pair 的两个成员均等于 c.end()

对 map 使用 find 代替下标操作

在 multimap 或 multiset 中查找元素

一种不同的，面向迭代器的解决方法

equal_range 函数

11.3.6 一个单词转换的map

单词转换程序

建立转换映射

生成转换文本

11.4 无序容器

新标准定义了 4 个无序关联容器（unordered associative container）。这些容器不是使用比较运算符来组织元素，而是使用一个哈希函数（hash function）和关键字类型的 == 运算符。在某些应用中，维护元素的序代价非常高，此时无序容器也很有容。

理论上哈希能获得更好的平均性能，但实际中想要达到很好地效果还需要进行性能测试和调优工作。使用无序容器通常更简单，也会有更好的性能。

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

使用无序容器

管理桶

无序容器在存储上组织为一组桶，每个桶保存零个或多个元素。无序容器使用一个哈希函数将元素映射到桶。为了访问一个元素，容器首先计算元素的哈希值，它指出应该搜索哪个桶。容器将具有一个特定哈希值的所有元素都保存在相同的桶中，因此，无序容器的性能依赖于哈希函数的质量和桶的数量和大小。

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

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

表11.8：无序容器管理操作

桶接口	
c.bucket_count()	正在使用的桶的数目
c.max_bucket_count()	容器能容纳的最多的桶的数量
c.bucket_size(n)	第 n 个桶中有多少个元素
c.bucket(k)	关键字为 k 的元素在哪个桶中
桶迭代	
local_iterator	可以用来访问桶中元素的迭代器类型
const_local_iterator	桶迭代器的 const 版本

c.begin(n), c.end(n)	桶 n 的首元素迭代器和尾后迭代器
c.cbegin(n), c.cend(n)	与前两个函数类似，但返回 float 值
哈希策略	
c.load_factor()	每个桶的平均元素数量，返回 float 值
c.max_load_factor()	C 试图维护的平均桶大小，返回 float 值。c 会在需要时添加新的桶，以使得 load_factor <= max_load_factor
c.rehash(n)	重组存储，使得 bucket_count >= n 且 bucket_count>size/max_load_factor
c.reserve(n)	重组存储，使得 c 可以保存 n 个元素且不必 rehash

无序容器对关键字类型的要求

小结

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

标准库定义了 8 个关联容器，每个容器

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

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

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

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

术语

12. 动态内存

目前为止，我们编写的程序中使用的对象都有着严格定义的生存期。全局对象在程序启动时分配，在程序结束时销毁。对于局部自动对象，当我们进入其定义所在的程序块时被创建，在离开块时销毁。局部 static 对象在第一次使用前分配，在程序结束时销毁。

除了自动和 static 对象外，C++还支持动态分配对象。动态分配的对象生存期与它们在哪里创建是无关的，只有当显式地被释放时，这些对象才会被销毁。

动态对象的正确释放被证明是编程中极容易出错的地方。为了更安全地使用动态对象，标准库定义了两个智能指针类型来管理动态分配的对象。当一个对象应该被释放时，指向它的智能指针可以确保自动地释放它。

目前为止只使用过静态内存和栈内存。静态内存用来保存局部 static 对象、类 static 数据成员已产出定义在任何函数之外的变量。栈内存用来保存在函数内的非 static 对象。分配在静态或栈内存中的对象由编译器自动创建和销毁。对于栈对象，仅在其定义的程序块运行时才存在：static 对象在使用之前分配，在程序结束时销毁。

除了静态内存和栈内存，每个程序还拥有一个内存池，这部分内存被称作自由空间（free store）或堆（heap）。程序用堆来存储动态分配

12.1 动态内存与智能指针

C++ 中，动态内存的管理是通过一对运算符来完成的：new，在动态内存中为对象分配空间并返回一个指向该对象的指针，可以选择对对象进行初始化；delete，接受一个动态对象的指针，销毁该对象，并释放与之关联的内存。

动态内存使用容易出问题，因为确保在正确的时间释放内存是极其困难的。有时忘记释放内存，这种情况下就会产生内存泄漏；有时在尚有指针引用内存的情况下就释放了它，就会产生引用非法内存的指针。

为了更容易、更安全地使用动态内存，新的标准库提供了两种智能指针（smart pointer）类型来管理动态对象。智能指针的行为类似常规指针，区别是它负责自动释放指向的对象。这两种智能指针的区别在于管理底层指针的方式：shared_ptr 允许多个指针指向同一个对象；unique_ptr 则独占所致想到e对象。标准库还定义了一个 weak_ptr 的伴随类，它是一种弱引用，指向 shared_ptr 所管理的对象。三种类型都定义在 memory 头文件。

12.1.1 shared_ptr 类

类似 vector，智能指针也是模板。因此，创建一个智能指针时，必须提供额外的信息——指针可以指向的类型。

```
1 shared_ptr<string> p1;        // shared_ptr，可以指向 string
```

```
2 shared_ptr<list<int>> p2;     // shared_ptr，可以指向 int 的 list
```

默认初始化的智能指针中保存着一个空指针。12.1.3节介绍初始化智能指针的其他方法。

智能指针的使用方式与普通指针类似。解引用一个智能指针返回它指向的对象。如果在一个条件判断中使用智能指针，效果就是检测它是否为空：

```
1 // 如果 p1 不为空，检查它是否指向一个空 string
2 if (p1 && p1->empty())
3     *p1 = "hi";
```

表12.1：shared_ptr 和 unique_ptr 都支持的操作

shared_ptr<T> sp	空智能指针，可以指向类型为 T 的对象
unique_ptr<T> up	
p	将 p 用作一个条件判断，若 p 指向一个对象，则为 true
*p	解引用 p，获得它指向的对象
p->mem	等价于 (*p).mem
p.get()	返回 p 中保存的指针。若智能指针释放了其对象，返回的智能指针所指向的对象也会消失
swap(p, q)	交换 p 和 q 中的指针
p.swap(q)	

表12.2：shared_ptr 独有的操作

make_shared<T>(args)	返回一个 shared_ptr，指向一个动态分配的类型为 T 的对象。使用 args 初始化此对象
shared_ptr<T>p(q)	p 是 shared_ptr q 的拷贝；此操作会递增 q 中的计数器。q 中的指针必须能转换为 T*
p = q	p 和 q 都是 shared_ptr，所保存的指针必须能相互转换。此操作会递减 p 的引用计数，递增 q 的引用计数；若 p 的引用计数变为 0，则将其管理的原内存释放
p.unique()	若 p.use_count() 为 1，返回 true；否则返回 false

p.use_count()	返回与 p 共享对象的智能指针数量；可能很慢，主要用于调试

make_shared 函数

最安全的分配和使用动态内存的方法是调用一个名为 make_shared 的标准库函数。此函数在动态内存中分配一个对象并初始化它，返回指向此对象的 shared_ptr。

```
// 指向一个值为 42 的 int 的 shared_ptr
shared_ptr<int> p3 = make_shared<int>(42);

auto p6 = make_shared<vector<string>>();
```

shared_ptr 的拷贝和赋值

当进行拷贝或赋值操作时，每个 shared_ptr 都会记录有多少个其他 shared_ptr 指向相同的对象

```
auto p = make_shared<int>(42);
auto q(p);    // p 和 q 指向相同对象，此对象有两个引用者
```

可以认为每个 shared_ptr 都有一个关联的计数器，称为引用计数（reference count）。拷贝一个 shared_ptr、用一个 shared_ptr 初始化另一个 shared_ptr，或将它作为参数传递给另一个函数以及作为函数的返回值时，它所关联的计数器就会递增。当我们给 shared_ptr 赋予一个新值或时 shared_ptr 被销毁（例如一个局部的shared_ptr 离开其作用域）时，计数器就会递减。

一个 shared_ptr 的计数器变为0，它就会自动释放自己所管理的对象：

```
auto r = make_shared<int>(42);    // r指向的int只有一个引用者
r = q;    // 给 r 赋值，令它指向另一个地址
          // 递增 q 指向的对象的引用计数
          // 递减 r 原来指向的对象的引用计数
          // r 原来指向的对象已没有引用者，会自动释放
```

shared_ptr 自动销毁所管理的对象...

当指向一个对象的最后一个 shared_ptr 被销毁时，shared_ptr 类会自动销毁此对象。它是通过析构函数（destructor）完成销毁工作的。

shared_ptr 的析构函数会递减它所指向对象的引用计数。如果引用计数变为 0，shared_ptr 的析构函数就会销毁对象，并释放它所占用的内存。

...shared_ptr 还会自动释放相关联的内存

> 如果将 shared_ptr 存放于一个容器中，而后不再需要全部元素，而只使用其中一部分，要记得用 erase 删除不再需要的那些元素。

使用了动态生存期的资源的类

程序使用动态内存处于以下三种原因之一

1. 程序不知道自己需要使用多少对象
2. 程序不知道所需对象的准确类型
3. 程序需要在多个对象间共享数据

容器类是出于第一种原因使用动态内存。15 章有出于第二种原因使用动态内存的例子。本节定义一个类，它使用动态内存来让多个对象能共享相同的底层数据。

使用过的类中，分配的资源都与对应对象生存期一致。例如，每个 vector 拥有其自己的元素。拷贝一个 vecotr 时，原 vecotr 和副本 vector 中的元素是分离的。

希望定义一个 Blob 的类，保存一组元素。Blog 对象的不同拷贝之间共享相同的元素。

如果两个对象共享底层的数据，当某个对象被销毁时，不能单方面地销毁底层数据：

```
Blob<string> b1;
{   // 新作用域
    Blob<sting> b2 = {"a", "an", "the"};
    b1 = b2;    // b1 和 b2 共享相同的元素
}    // b2 被销毁了，但 b2 的元素不能被销毁
     // b1 指向最初由 b2 创建的元素
```

> 使用动态内存的一个常见原因是允许多个对象共享相同的状态。

定义 StrBlob 类

StrBlob 构造函数

元素访问成员函数

StrBlob 的拷贝、赋值和销毁

12.1.2 直接管理内存

使用 new、delete来分配和释放动态内存非常容易出错。而且，自己直接管理内存的类与使用智能指针的类不同，它们不能依赖类对象拷贝、赋值和销毁操作的任何默认定义。

使用 new 动态分配和初始化对象

动态分配的 const 对象

内存耗尽

释放动态内存

指针值和 delete

释放一块并非 new 分配的内存，或者将相同的指针值释放多次，其行为是未定义的

动态对象的生存期直到被释放时为止

> 由内置指针（而不是智能指针）管理的动态内存在被显式释放前一直都会存在。

> 动态内存的管理非常容易出错
>
> 1. 忘记 delete 内存。导致内存泄漏，查找内存泄漏是非常困难的，通常程序运行很长时间后，真正耗尽内存时，才能检测到这种错误。
> 2. 使用已经释放掉的对象。
> 3. 同一块内存释放两次。
>
> 坚持只使用智能指针，可以避免这些问题。对于一块内存，只有 在没有任何智能指针指向它的情况下，智能指针才会自动释放它。

delete 之后重置指针值...

delete 一个指针后，指针值就变为无效了。但在很多机器上指针仍然保存着（已经释放了的）动态内存的地址。delete 后，指针就变成了空悬指针，即，指向一块曾经保存数据对象但现在已经无效的内存的指针。

避免空悬指针的方法：在指针即将离开其作用域之前释放掉它所关联的内存。这样，在指针关联的内存被释放掉后，就没有机会继续使用指针了。如果需要保留指针，可以在delete后将nullptr赋予指针，指出指针不指向任何对象。

...这只是提供了有限的保护

动态内存的一个基本问题是可能有多个指针指向相同的内存，在delete内存之后重置指针的方法只对这个指针有效，对其他任何仍指向（已释放的）内存的指针是没有作用的。

实际系统中，查找指向相同内存的所有指针是异常困难的。

12.1.3 shared_ptr 和 new 结合使用

默认情况下，一个用来初始化智能指针的普通指针必须指向动态内存，因为智能指针默认使用 delete 释放它所关联的对象。我们可以将智能指针绑定到一个指向其他类型的资源的指针上，但是必须提供自己的操作来替代 delete。12.1.4节介绍如何定义自己的释放操作。

表12.3：定义和改变 shared_ptr 的其他方法

shared_ptr<T> p(q)	p 管理内置指针 q 所指向的对象；q 必须指向 new 分配的内存，且能够转换为 T* 类型
shared_ptr<T> p(u)	p 从 unique_ptr u 那里接管了对象的所有权；将 u 置为空
shared_ptr<T> p(q, d)	p 接管了内置指针 q 所指向的对象的所有权。q 必须能转换为 T*类型。p 将使用可调用对象 d 来代替 delete
shared_ptr<T> p(p2, d)	p 是 shared_ptr p2 的拷贝，区别是 p 将用可调用对象 d 来代替 delete
p.reset() p.reset(q) p.reset(q, d)	若 p 是唯一指向其对象的 shared_ptr，reset 会释放此对象。 若传递了可选的参数内置指针q，会令 p 指向 q，否则会将 p 置为空。 若还传递了参数 d，将会调用 d 而不是 delete 来释放 q

不要混合使用普通指针和智能指针...

> 使用一个内置指针来访问一个智能指针所负责的对象是很危险的，因为我们无法知道对象何时会被销毁。

...也不要使用 get 初始化另一个智能指针或为智能指针赋值

其他 shared_ptr 操作

12.1.4 智能指针和异常

使用异常处理的程序能在异常发生后令程序流程继续。这种程序需要确保在异常发生后资源能被正确地释放。一个简单的确保资源被释放的方法是使用智能指针。

```
1 void f() {
2     shared_ptr<int> sp(new int (42));
3     // 抛出一个异常，且在 f 中未被捕获
4 }   // 在函数结束时 shared_ptr 自动释放内存
```

函数的退出有两种可能，正常处理结束或者发生了异常，无论哪种情况，局部对象都会被销毁。sp 是一个 shared_ptr，因此 sp 销毁时会检查引用计数。此例中 sp 是指向这块内存的唯一指针，因此内存会被释放掉。

相反，当发生异常时，直接管理的内存是不会自动释放的。

```
void f() {
    int *ip = new int(42);
    // 抛出异常，且在 f 中未被捕获
    delete ip;    // 退出之前释放内存
}
```

如果在 new 和 delete 之间发生异常，且异常未在 f 中被捕获，则内存就永远不会释放了。在函数 f 之外没有指针指向这块内存，因此无法释放它了。

智能指针和哑类

包括所有标准库类在内的很多 C++类都定义了析构函数负责清理对象使用的资源。但是，不是所有的类都是这样良好定义的。特别是那些为 C 和 C++ 设计的类，通常要求用户显式地释放所使用的任何资源。

那些分配了资源，有没有定义析构函数来释放这些资源的类，可能会遇到与使用动态内存相同的错误——程序员容易忘记释放资源。类似，如果在资源分配和释放之间发生了异常，程序也会发生资源泄露。

与管理动态内存类似，可以使用类似技术来管理不具有良好定义的析构函数的类。例如使用一个 C 和 C++ 都使用的网络库：

```
struct destination;                 // 正在连接什么
struct connection;                  // 使用连接所需的信息
connection connect(destination*);   // 打开连接
void disconnect(connection);        // 关闭给定的连接
void f(destination &d /* 其他参数 */)
{
    // 获得一个连接；记住使用完后要关闭它
    connection c = connect(&d);
    // 使用连接
    // 如果在 f 退出前忘记调用 disconnect，就无法关闭 c 了
}
```

如果 connection 有一个析构函数，就可以在 f 结束时由析构函数自动关闭连接。但是，connection 没有析构函数，这个问题与我们上一个程序中使用 shared_ptr 避免内存泄漏几乎是等价的。使用

shared_ptr 来保证 connection 被正确关闭，是一种有效的方法。

使用我们自己的释放操作

默认 shared_ptr 假定它们指向的是动态内存。因此，当一个 shared_ptr 被销毁时，它默认地对它管理的指针进行delete 操作。为了用 shared_ptr 来管理一个 connection，必须先定义一个函数来代替 delete。这个删除器（deleter）函数必须能够完成对 shared_ptr 中保存的指针进行释放的操作。

```
void end_connection(connection *p) { disconnect(*p); }

void f(destination &d /* 其他函数 */)
{
    connection c = connect(&d);
    // 创建一个 shared_ptr 时，可以传递一个（可选的）指向删除器函数的参数
    shared_ptr<connection> p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由于异常退出），connection 会被正确关闭
}
```

当 p 被销毁时，它不会对自己保存的指针执行 delete，而是调用 end_connection。end_connection 会调用 disconnect，从而确保连接被关闭。如果 f 正常跳出，那么 p 的销毁会作为结束处理的一部分。如果发生了异常，p通常会被销毁，从而连接被关闭。

> 智能指针陷阱
>
> 正确使用智能指针的一些基本规范：
>
> - 不使用相同的内置指针初始化（或 reset）多个智能指针
> - 不 delete get() 返回的指针
> - 不使用 get() 初始化或 reset 另一个智能指针
> - 如果使用 get() 返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就无效了
> - 如果使用智能指针管理的资源不是new分配的内存，记住传递给它一个删除器

12.1.5 unique_ptr

一个 unique_ptr 拥有它所指向的对象，因此 unique_ptr 不支持普通的拷贝或赋值操作

表12.4：unique_ptr 操作（表12.1）

unique_ptr<T> u1	空 unique_ptr，可以指向类型为 T 的对象。u1 会使用 delete 来释放它的指针；u2 会使用一个类型为 D 的可调用对象来释放它的指针

unique_ptr<T, D> u2	
unique_ptr<T, D> u(d)	空 unique_ptr，指向类型为 T 的对象，用类型为 D 的对象 d 代替delete
u = nullptr	释放 u 指向的对象，将 u 置为空
u.release()	u 放弃对指针的控制权，返回指针，并将 u 置为空
u.reset() u.reset(q) u.reset(nullptr)	释放 u 指向的对象 如果提供了内置指针 q，令 u 指向这个对象；否则将 u 置为空

虽然不能拷贝或赋值 unique_ptr，但可以通过调用 release 或 reset 将指针的所有权从一个（非 const）unique_ptr 转移给另一个 unique。

传递 unique_ptr 参数和返回 unique_ptr

不能拷贝 unique_ptr 的规则有一个例外：可以拷贝或赋值一个将要被销毁的 unique_ptr。常见例子是从函数返回一个 unique_ptr

向 unique_ptr 传递删除器

12.1.6 weak_ptr

weak_ptr 是一种不控制所指向对象生存期的智能指针，它指向由一个 shared_ptr 管理的对象。将一个 weak_ptr 绑定到一个 shared_ptr 不会改变 shared_ptr 的引用计数。一旦最后一个指向对象的 shared_ptr 被销毁，对象就会被释放。即使有 weak_ptr 指向对象，对象也还是会被释放。弱共享对象。

表 12.5：weak_ptr

weak_ptr<T> w	空 weak_ptr 可以指向类型为 T 的对象
weak_ptr<T> w(sp)	与 shared_ptr sp 指向相同对象的 weak_ptr。T 必须能转换为 sp 指向的类型
w = p	p 可以是一个 shared_ptr 或一个 weak_ptr。赋值后 w 与 p 共享对象
w.reset()	将 w 置为空
w.use_count()	与 w 共享对象的 shared_ptr 的数量
w.expired()	若 w.use_count() 为 0，返回 true，否则返回 false

w.lock()	如果 expired 为 true，返回一个空 shared_ptr；否则返回一个指向 w 的对象的 shared_ptr

创建一个 weak_ptr 时，要用一个 shared_ptr 来初始化它：

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p);     // wp 弱共享 p; p 的引用计数未改变
```

由于对象可能不存在，不能使用 weak_ptr 直接访问对象，必须调用lock。

```
if (shared_ptr<int> np = wp.lock()) {     // 如果 np 不为空则条件成立
    // 在 if 中，np 与 p 共享对象
}
```

核查指针类

指针操作

12.2 动态数组

为了一次给很多对象分配内存，C++ 和标准库提供了两种一次分配一个对象数组的方法。new 分配并初始化一个对象数组。标准库中有个 allocator 的类，允许将分配和初始化分离，使用 allocator 通常会提供更好的性能和更灵活的内存管理能力。

> 大多数应用应该使用标准库容器而不是动态分配的数组。使用容器更为简单，不容易出现内存管理错误并且有更好的性能。

使用容器的类可以使用默认版本的拷贝、赋值和析构操作。分配动态数组的类必须定义自己版本的操作，在拷贝、复制以及销毁对象时管理所关联的内存。

> 13章前，不要在类内的代码中分配动态内存

12.2.1 new 和数组

分配一个数组会得到一个元素类型的指针

分配的内存并不是一个数组类型，因此不能对动态数组调用 begin 或 end。这些函数使用数组维度来返回指向首元素和尾后元素的指针。相同原因，也不能范围 for 语句来处理动态数组中的元素。

> 动态数组并不是数组类型

初始化动态分配对象的数组

动态分配一个空数组是合法的

```
1 char arr[0];            // 错误：不能定义长度为 0 的数组
2 char *cp = new char[0]; // 正确：但 cp 不能解引用
```

释放动态数组

```
1 delete p;        // p 必须指向一个动态分配的对象或为空
2 delete [] pa;    // pa 必须指向一个动态分配的数组或为空
```

智能指针和动态数组

标准库提供了一个可以管理 new 分配的数组的 unque_ptr 版本。为了用一个 unique_ptr 管理动态数组，必须在对象类型后面跟一对空方括号

```
1 // up 指向一个包含 10 个未初始化 int 的数组
2 unique_ptr<int[]> up(new int [10]);
3 up.release();    // 自动用 delete[] 销毁其指针
```

表12.6：指向数组的 unique_ptr

指向数组的 unique_ptr 不支持成员访问运算符（点和箭头运算符） 其他 unique_ptr 操作不变	
unique_ptr<T[]> u	u 可以指向一个动态分配的数组，数组元素类型为 T
unique_ptr<T[]> u(p)	u 指向内置指针 p 所指向的动态分配的数组。p 必须能转换为类型 T*
u[i]	返回 u 拥有的数组中位置 i 处的对象 u 必须指向一个数组

shared_ptr 不直接支持管理动态数组。希望使用 shared_ptr 管理一个动态数组，必须提供自己定义的删除器。shared_ptr 未定义下标运算符，为了访问数组中的元素，需要用 get 获取一个内置指针，用它来访问数组元素。

12.2.2 allocator 类

new 在灵活性上的局限，一方面它将内存分配和对象构造组合在了一起。delete 将对象析构和内存释放组合在了一起。分配单个对象时，通常希望将内存分配和对象初始化组合在一起，该情况下，基本知道对象应有什么值。

当分配一大块内存时，通常计划在这块内存上按需构造对象。在此情况下，我们希望将内存分配和对象构造分离。这意味着我们可以分配大块内存，但只在真正需要时才真正执行对象创建操作（同时付出一定开销）。

将内存分配和对象构造组合在一起可能会导致不必要的浪费。

allocator 类

标准库 allocator 类帮助我们将内存分配和对象构造分离开来。它提供一种类型感知的内存分配方法，它分配的内存是原始的、未构造的。本节介绍 allocator 支持的操作，13.5节介绍如何使用这个类的典型例子。

类似 vector，allocator 是一个模板。定义一个 allocator 对象，必须指明这个 allocator 可以分配的对象类型。当一个 allocator 对象分配内存时，它会根据给定的对象类型来确定恰当的内存大小和对齐位置：

```
allocator<string> alloc;            //可以分配 string 的 allocator 对象
auto const p = alloc.allocate(n);   //分配 n 个未初始化的 string
```

表12.7：标准库 allocator 类及其算法

allocator<T> a	定义一个名为 a 的 allocator 对象，它可以为类型为 T 的对象分配内存
a.allocate(n)	分配一段原始的、未构造的内存，保存 n 个类型为 T 的对象
a.deallocate(p, n)	释放从 T* 指针 p 中地址开始的内存，这个内存保存了 n 个类型为 T 的对象；p 必须是一个先前由 allocate 返回的指针，且 n 必须是 p 创建时所要求的大小。再调用 deallocate 之前，用户必须对每个在这块内存中创建的对象调用 destroy
a.construct(p, args)	p 必须是一个类型为 T* 的指针，指向一块原始内存；arg 被传递给类型为 T 的构造函数，用来在 p 指向的内存中构造一个对象

a.destroy(p)	p 为 T* 类型的指针，此算法对 p 指向的对象执行析构函数

allocator 分配未构造的内存

allocator 分配的内存是未构造的（unconstructed）。需要在此内存中构造对象。新标准库中，construct 成员函数接受一个指针和零个或多个额外参数，在给定位置构造一个元素。额外参数用来初始化构造的对象。

```
auto q = p;     // q指向最后构造的元素之后的位置
alloc.construct(q++);             // *q为空字符串
alloc.construct(q++, 10, 'c');    // *q为 cccccccccc
alloc.construct(q++, "hi");
```

> 为了使用 allocate 返回的内存，必须用 construct 构造对象。使用未构造的内存，其行为是未定义的。

用完对象后，必须对每个构造的元素调用 destroy 来销毁它们。函数 destroy 接受一个指针，对指向的对象执行析构函数。

```
while (q != p)
    alloc.destroy(--q);    // 释放我们真正构造的 string
```

元素被销毁后，可以重新使用这部分内存来保存其他 string，也可以将其归还给系统。释放内存通过调用 deallocate 来完成：

```
alloc.deallocate(p, n);
```

拷贝和填充未初始化内存的算法

标准库为 allocator 类定义了两个伴随算法，可以在未初始化内存中创建对象。

表12.8：allocator 算法

这些函数在给定目的位置创建元素，而不是由系统分配内存给它们	
uninitialized_copy(b, e, b2)	从迭代器 b 和 e 指出的输入范围中拷贝元素到迭代器 b2 指定的未构造的原始内存中。b2 指向的内存必须足够大，能容纳输入序列中元素的拷贝

uninitialized_copy_n(b, n, b2)	从迭代器 b 指向的元素开始，拷贝 n 个元素到 b2 开始的内存中
uninitialized_fill(b, e, t)	在迭代器 b 和 e 指定的原始内存范围中创建对象，对象的值均为 t 的拷贝
uninitialized_fill_n(b, n, t)	从迭代器 b 指向的内存地址开始创建 n 个对象，b 必须指向足够大的未初始化的原始内存，能够容纳给定数量的对象

12.3 使用标准库：文本查询程序

12.3.1 文本查询程序设计

数据结构

在类之间共享数据

使用 TextQuery 类

12.3.2 文本查询程序类的定义

TextQuery 构造函数

QueryResult 类

query 函数

小结

C++ 中，内存是通过 new 表达式分配，通过 delete 表达式释放的。标准库还定义了一个 allocator 类来分配动态内存块。

分配动态内存的程序应负责释放它所分配的内存。内存的正确释放是非常容易出错的地方：要么内存永远不会被释放，要么在仍有指针引用它时就被释放了。新的标准库定义了智能指针类型——shared_ptr、unique_ptr 和 weak_ptr，可令动态内存管理更为安全。对于一块内存，当没有任何用户使用它时，智能指针会自动释放它。尽可能使用智能指针。

术语