

第 12 章

动态内存

内容

12.1 动态内存与智能指针.....	400
12.2 动态数组.....	423
12.3 使用标准库：文本查询程序.....	430
小结	436
术语表.....	436

到目前为止，我们编写的程序中所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。动态分配的对象的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，这些对象才会销毁。

动态对象的正确释放被证明是编程中极其容易出错的地方。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。

450

我们的程序到目前为止只使用过静态内存或栈内存。静态内存用来保存局部 `static` 对象（参见 6.6.1 节，第 185 页）、类 `static` 数据成员（参见 7.6 节，第 268 页）以及定义在任何函数之外的变量。栈内存用来保存定义在函数内的非 `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（dynamically allocate）的对象——即，那些在程序运行时分配的对象。动态对象的生存期由程序来控制，也就是说，当动态对象不再使用时，我们的代码必须显式地销毁它们。



WARNING 虽然使用动态内存有时是必要的，但众所周知，正确地管理动态内存是非常棘手的。

12.1 动态内存与智能指针

在 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，我们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。

动态内存的使用很容易出问题，因为确保在正确的时间释放内存是极其困难的。有时我们会忘记释放内存，在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我们就释放了它，在这种情况下就会产生引用非法内存的指针。

C++ 11

为了更容易（同时也更安全）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，重要的区别是它负责自动释放所指向的对象。新标准库提供的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”所指向的对象。标准库还定义了一个名为 `weak_ptr` 的伴随类，它是一种弱引用，指向 `shared_ptr` 所管理的对象。这三种类型都定义在 `memory` 头文件中。



12.1.1 shared_ptr 类

C++ 11

类似 `vector`，智能指针也是模板（参见 3.3 节，第 86 页）。因此，当我们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。与 `vector` 一样，我们在尖括号内给出类型，之后是所定义的这种智能指针的名字：

```
451> shared_ptr<string> p1;           // shared_ptr, 可以指向 string
      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（参见 2.3.2 节，第 48 页）。在 12.1.3 节中（见第 412 页），我们将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：

```
// 如果 p1 不为空，检查它是否指向一个空 string
if (p1 && p1->empty())
    *p1 = "hi"; // 如果 p1 指向一个空 string，解引用 p1，将一个新值赋予 string
```

表 12.1 列出了 `shared_ptr` 和 `unique_ptr` 都支持的操作。只适用于 `shared_ptr` 的

操作列于表 12.2 中。

表 12.1: `shared_ptr` 和 `unique_ptr` 都支持的操作

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

表 12.2: `shared_ptr` 独有的操作

<code>make_shared<T>(args)</code>	返回一个 <code>shared_ptr</code> ，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
<code>shared_ptr<T>p(q)</code>	p 是 <code>shared_ptr q</code> 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 <code>T*</code> （参见 4.11.2 节，第 143 页）
<code>p = q</code>	p 和 q 都是 <code>shared_ptr</code> ，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
<code>p.unique()</code>	若 <code>p.use_count()</code> 为 1，返回 <code>true</code> ；否则返回 <code>false</code>
<code>p.use_count()</code>	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`。与智能指针一样，`make_shared` 也定义在头文件 `memory` 中。

当要用 `make_shared` 时，必须指定想要创建的对象的类型。定义方式与模板类相同，在函数名之后跟一个尖括号，在其中给出类型：

```
// 指向一个值为 42 的 int 的 shared_ptr
shared_ptr<int> p3 = make_shared<int>(42);
// p4 指向一个值为"9999999999"的 string
shared_ptr<string> p4 = make_shared<string>(10, '9');
// p5 指向一个值初始化的(参见 3.3.1 节，第 88 页)int，即，值为 0
shared_ptr<int> p5 = make_shared<int>();
```

类似顺序容器的 `emplace` 成员（参见 9.3.1 节，第 308 页），`make_shared` 用其参数来构造给定类型的对象。例如，调用 `make_shared<string>` 时传递的参数必须与 `string` 的某个构造函数相匹配，调用 `make_shared<int>` 时传递的参数必须能用来初始化一个 `int`，依此类推。如果我们不传递任何参数，对象就会进行值初始化（参见 3.3.1 节，第 88 页）。

当然，我们通常用 `auto`（参见 2.5.2 节，第 61 页）定义一个对象来保存 `make_shared` 的结果，这种方式较为简单：

```
// p6 指向一个动态分配的空 vector<string>
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); // p 指向的对象只有 p 一个引用者
auto q(p); // p 和 q 指向相同对象，此对象有两个引用者
```

452 我们可以认为每个 shared_ptr 都有一个关联的计数器，通常称其为引用计数 (reference count)。无论何时我们拷贝一个 shared_ptr，计数器都会递增。例如，当用一个 shared_ptr 初始化另一个 shared_ptr，或将它作为参数传递给一个函数（参见 6.2.1 节，第 188 页）以及作为函数的返回值（参见 6.3.2 节，第 201 页）时，它所关联的计数器就会递增。当我们给 shared_ptr 赋予一个新值或是 shared_ptr 被销毁（例如一个局部的 shared_ptr 离开其作用域（参见 6.1.1 节，第 184 页）时，计数器就会递减。

一旦一个 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
```

此例中我们分配了一个 int，将其指针保存在 r 中。接下来，我们将一个新值赋予 r。在此情况下，r 是唯一指向此 int 的 shared_ptr，在把 q 赋给 r 的过程中，此 int 被自动释放。



到底是用一个计数器还是其他数据结构来记录有多少指针共享对象，完全由标准库的具体实现来决定。关键是智能指针类能记录有多少个 shared_ptr 指向相同的对象，并能在恰当的时候自动释放对象。

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。它是通过另一个特殊的成员函数——析构函数 (destructor) 完成销毁工作的。类似于构造函数，每个类都有一个析构函数。就像构造函数控制初始化一样，析构函数控制此类型的对象销毁时做什么操作。

453 析构函数一般用来释放对象所分配的资源。例如，string 的构造函数（以及其他 string 成员）会分配内存来保存构成 string 的字符。string 的析构函数就负责释放这些内存。类似的，vector 的若干操作都会分配内存来保存其元素。vector 的析构函数就负责销毁这些元素，并释放它们所占用的内存。

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

……shared_ptr 还会自动释放相关联的内存

当动态对象不再被使用时，shared_ptr 类会自动释放动态对象，这一特性使得动态内存的使用变得非常容易。例如，我们可能有一个函数，它返回一个 shared_ptr，指向

一个 `Foo` 类型的动态分配的对象，对象是通过一个类型为 `T` 的参数进行初始化的：

```
// factory 返回一个 shared_ptr，指向一个动态分配的对象
shared_ptr<Foo> factory(T arg)
{
    // 恰当地处理 arg
    // shared_ptr 负责释放内存
    return make_shared<Foo>(arg);
}
```

由于 `factory` 返回一个 `shared_ptr`，所以我们可以确保它分配的对象会在恰当的时刻被释放。例如，下面的函数将 `factory` 返回的 `shared_ptr` 保存在局部变量中：

```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // 使用 p
} // p 离开了作用域，它指向的内存会被自动释放掉
```

由于 `p` 是 `use_factory` 的局部变量，在 `use_factory` 结束时它将被销毁（参见 6.1.1 ◀454 节，第 184 页）。当 `p` 被销毁时，将递减其引用计数并检查它是否为 0。在此例中，`p` 是唯一引用 `factory` 返回的内存的对象。由于 `p` 将要销毁，`p` 指向的这个对象也会被销毁，所占用的内存会被释放。

但如果其他 `shared_ptr` 也指向这块内存，它就不会被释放掉：

```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // 使用 p
    return p; // 当我们返回 p 时，引用计数进行了递增操作
} // p 离开了作用域，但它指向的内存不会被释放掉
```

在此版本中，`use_factory` 中的 `return` 语句向此函数的调用者返回一个 `p` 的拷贝。拷贝一个 `shared_ptr` 会增加所管理对象的引用计数值。现在当 `p` 被销毁时，它所指向的内存还有其他使用者。对于一块内存，`shared_ptr` 类保证只要有任何 `shared_ptr` 对象引用它，它就不会被释放掉。

由于在最后一个 `shared_ptr` 销毁前内存都不会释放，保证 `shared_ptr` 在无用之后不再保留就非常重要了。如果你忘记了销毁程序不再需要的 `shared_ptr`，程序仍会正确执行，但会浪费内存。`shared_ptr` 在无用之后仍然保留的一种可能情况是，你将 `shared_ptr` 存放在一个容器中，随后重排了容器，从而不再需要某些元素。在这种情况下，你应该确保用 `erase` 删除那些不再需要的 `shared_ptr` 元素。



如果你将 `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

程序使用动态内存出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而使用动态内存的典型例子，我们将在第 15 章看到出于第二种原因而使用动态内存的例子。在本节中，我们将定义一个类，它使用动态内存是为了让多个对象能共享相同的底层数据。

到目前为止，我们使用过的类中，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。当我们拷贝一个 `vector` 时，原 `vector` 和副本 `vector` 中的元素是相互分离的：

```
455> vector<string> v1; // 空 vector
{ // 新作用域
    vector<string> v2 = {"a", "an", "the"};
    v1 = v2; // 从 v2 拷贝元素到 v1 中
} // v2 被销毁，其中的元素也被销毁
// v1 有三个元素，是原来 v2 中元素的拷贝
```

由一个 `vector` 分配的元素只有当这个 `vector` 存在时才存在。当一个 `vector` 被销毁时，这个 `vector` 中的元素也都被销毁。

但某些类分配的资源具有与原对象相独立的生存期。例如，假定我们希望定义一个名为 `Blob` 的类，保存一组元素。与容器不同，我们希望 `Blob` 对象的不同拷贝之间共享相同的元素。即，当我们拷贝一个 `Blob` 时，原 `Blob` 对象及其拷贝应该引用相同的底层元素。

一般而言，如果两个对象共享底层的数据，当某个对象被销毁时，我们不能单方面地销毁底层数据：

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

在此例中，`b1` 和 `b2` 共享相同的元素。当 `b2` 离开作用域时，这些元素必须保留，因为 `b1` 仍然在使用它们。



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

最终，我们会将 `Blob` 类实现为一个模板，但我们直到 16.1.2 节（第 583 页）才会学习模板的相关知识。因此，现在我们先定义一个管理 `string` 的类，此版本命名为 `StrBlob`。

实现一个新的集合类型的最简单方法是使用某个标准库容器来管理元素。采用这种方法，我们可以借助标准库类型来管理元素所使用的内存空间。在本例中，我们将使用 `vector` 来保存元素。

但是，我们不能在一个 `Blob` 对象内直接保存 `vector`，因为一个对象的成员在对象销毁时也会被销毁。例如，假定 `b1` 和 `b2` 是两个 `Blob` 对象，共享相同的 `vector`。如果此 `vector` 保存在其中一个 `Blob` 中——例如 `b2` 中，那么当 `b2` 离开作用域时，此 `vector` 也将被销毁，也就是说其中的元素都将不复存在。为了保证 `vector` 中的元素继续存在，

我们将 `vector` 保存在动态内存中。

为了实现我们所希望的数据共享，我们为每个 `StrBlob` 设置一个 `shared_ptr` 来管理动态分配的 `vector`。此 `shared_ptr` 的成员将记录有多少个 `StrBlob` 共享相同的 `vector`，并在 `vector` 的最后一个使用者被销毁时释放 `vector`。456

我们还需要确定这个类应该提供什么操作。当前，我们将实现一个 `vector` 操作的小的子集。我们会修改访问元素的操作（如 `front` 和 `back`）：在我们的类中，如果用户试图访问不存在的元素，这些操作会抛出一个异常。

我们的类有一个默认构造函数和一个构造函数，接受单一的 `initializer_list<string>` 类型参数（参见 6.2.6 节，第 198 页）。此构造函数可以接受一个初始化器的花括号列表。

```
class StrBlob {
public:
    typedef std::vector<std::string>::size_type size_type;
    StrBlob();
    StrBlob(std::initializer_list<std::string> il);
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // 添加和删除元素
    void push_back(const std::string &t) {data->push_back(t);}
    void pop_back();
    // 元素访问
    std::string& front();
    std::string& back();
private:
    std::shared_ptr<std::vector<std::string>> data;
    // 如果 data[i] 不合法，抛出一个异常
    void check(size_type i, const std::string &msg) const;
};
```

在此类中，我们实现了 `size`、`empty` 和 `push_back` 成员。这些成员通过指向底层 `vector` 的 `data` 成员来完成它们的工作。例如，对一个 `StrBlob` 对象调用 `size()` 会调用 `data->size()`，依此类推。

StrBlob 构造函数

两个构造函数都使用初始化列表（参见 7.1.4 节，第 237 页）来初始化其 `data` 成员，令它指向一个动态分配的 `vector`。默认构造函数分配一个空 `vector`：

```
StrBlob::StrBlob(): data(make_shared<vector<string>>()) {}
StrBlob::StrBlob(initializer_list<string> il):
    data(make_shared<vector<string>>(il)) {}
```

接受一个 `initializer_list` 的构造函数将其参数传递给对应的 `vector` 构造函数（参见 2.2.1 节，第 39 页）。此构造函数通过拷贝列表中的值来初始化 `vector` 的元素。

元素访问成员函数

`pop_back`、`front` 和 `back` 操作访问 `vector` 中的元素。这些操作在试图访问元素之前必须检查元素是否存在。由于这些成员函数需要做相同的检查操作，我们为 `StrBlob` 定义了一个名为 `check` 的 `private` 工具函数，它检查一个给定索引是否在合法范围内。457

除了索引，`check` 还接受一个 `string` 参数，它会将此参数传递给异常处理程序，这个 `string` 描述了错误内容：

```
void StrBlob::check(size_type i, const string &msg) const
{
    if (i >= data->size())
        throw out_of_range(msg);
}
```

`pop_back` 和元素访问成员函数首先调用 `check`。如果 `check` 成功，这些成员函数继续利用底层 `vector` 的操作来完成自己的工作：

```
string& StrBlob::front()
{
    // 如果 vector 为空，check 会抛出一个异常
    check(0, "front on empty StrBlob");
    return data->front();
}

string& StrBlob::back()
{
    check(0, "back on empty StrBlob");
    return data->back();
}

void StrBlob::pop_back()
{
    check(0, "pop_back on empty StrBlob");
    data->pop_back();
}
```

`front` 和 `back` 应该对 `const` 进行重载（参见 7.3.2 节，第 247 页），这些版本的定义留作练习。

StrBlob 的拷贝、赋值和销毁

类似 `Sales_data` 类，`StrBlob` 使用默认版本的拷贝、赋值和销毁成员函数来对此类型的对象进行这些操作（参见 7.1.5 节，第 239 页）。默认情况下，这些操作拷贝、赋值和销毁类的数据成员。我们的 `StrBlob` 类只有一个数据成员，它是 `shared_ptr` 类型。因此，当我们拷贝、赋值或销毁一个 `StrBlob` 对象时，它的 `shared_ptr` 成员会被拷贝、赋值或销毁。

如前所见，拷贝一个 `shared_ptr` 会递增其引用计数；将一个 `shared_ptr` 赋予另一个 `shared_ptr` 会递增赋值号右侧 `shared_ptr` 的引用计数，而递减左侧 `shared_ptr` 的引用计数。如果一个 `shared_ptr` 的引用计数变为 0，它所指向的对象会被自动销毁。因此，对于由 `StrBlob` 构造函数分配的 `vector`，当最后一个指向它的 `StrBlob` 对象被销毁时，它会随之被自动销毁。

458

12.1.1 节练习

练习 12.1：在此代码的结尾，`b1` 和 `b2` 各包含多少个元素？

```
StrBlob b1;
{
    StrBlob b2 = {"a", "an", "the"};
```

```

    b1 = b2;
    b2.push_back("about");
}

```

练习 12.2: 编写你自己的 StrBlob 类，包含 const 版本的 front 和 back。

练习 12.3: StrBlob 需要 const 版本的 push_back 和 pop_back 吗？如果需要，添加进去。否则，解释为什么不需要。

练习 12.4: 在我们的 check 函数中，没有检查 i 是否大于 0。为什么可以忽略这个检查？

练习 12.5: 我们未编写接受一个 initializer_list explicit（参见 7.5.4 节，第 264 页）参数的构造函数。讨论这个设计策略的优点和缺点。

12.1.2 直接管理内存

C++语言定义了两个运算符来分配和释放动态内存。运算符 new 分配内存，delete 释放 new 分配的内存。

相对于智能指针，使用这两个运算符管理内存非常容易出错，随着我们逐步详细介绍这两个运算符，这一点会更为清楚。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（参见 7.1.4 节，第 237 页）。因此，使用智能指针的程序更容易编写和调试。



在学习第 13 章之前，除非使用智能指针来管理内存，否则不要分配动态内存。

使用 new 动态分配和初始化对象

在自由空间分配的内存是无名的，因此 new 无法为其分配的对象命名，而是返回一个指向该对象的指针：

```
int *pi = new int; // pi 指向一个动态分配的、未初始化的无名对象
```

此 new 表达式在自由空间构造一个 int 型对象，并返回指向该对象的指针。

默认情况下，动态分配的对象是默认初始化的（参见 2.2.1 节，第 40 页），这意味着内置类型或组合类型的对象的值将是未定义的，而类类型对象将用默认构造函数进行初始化：

```
string *ps = new string; // 初始化为空 string
int *pi = new int; // pi 指向一个未初始化的 int
```

459

我们可以使用直接初始化方式（参见 3.2.1 节，第 76 页）来初始化一个动态分配的对象。我们可以使用传统的构造方式（使用圆括号），在新标准下，也可以使用列表初始化（使用花括号）：

```
int *pi = new int(1024); // pi 指向的对象的值为 1024
string *ps = new string(10, '9'); // *ps 为"9999999999"
// vector 有 10 个元素，值依次从 0 到 9

vector<int> *pv = new vector<int>{0,1,2,3,4,5,6,7,8,9};
```

C++
11

也可以对动态分配的对象进行值初始化（参见 3.3.1 节，第 88 页），只需在类型名之

后跟一对空括号即可：

```
string *ps1 = new string;      // 默认初始化为空 string
string *ps = new string();    // 值初始化为空 string
int *pi1 = new int;           // 默认初始化；*pi1 的值未定义
int *pi2 = new int();         // 值初始化为 0；*pi2 为 0
```

对于定义了自己的构造函数（参见 7.1.4 节，第 235 页）的类类型（例如 `string`）来说，要求值初始化是没有意义的；不管采用什么形式，对象都会通过默认构造函数来初始化。但对于内置类型，两种形式的差别就很大了：值初始化的内置类型对象有着良好定义的值，而默认初始化的对象的值则是未定义的。类似的，对于类中那些依赖于编译器合成的默认构造函数的内置类型成员，如果它们未在类内被初始化，那么它们的值也是未定义的（参见 7.1.4 节，第 236 页）。

Best Practices

出于与变量初始化相同的原因，对动态分配的对象进行初始化通常是个好主意。

C++ 11 如果我们提供了一个括号包围的初始化器，就可以使用 `auto`（参见 2.5.2 节，第 61 页）从此初始化器来推断我们想要分配的对象的类型。但是，由于编译器要用初始化器的类型来推断要分配的类型，只有当括号中仅有单一初始化器时才可以使用 `auto`：

```
auto p1 = new auto(obj);      // p 指向一个与 obj 类型相同的对象
                               // 该对象用 obj 进行初始化
auto p2 = new auto{a,b,c};    // 错误：括号中只能有单个初始化器
```

`p1` 的类型是一个指针，指向从 `obj` 自动推断出的类型。若 `obj` 是一个 `int`，那么 `p1` 就是 `int*`；若 `obj` 是一个 `string`，那么 `p1` 是一个 `string*`；依此类推。新分配的对象用 `obj` 的值进行初始化。

动态分配的 `const` 对象

用 `new` 分配 `const` 对象是合法的：

```
// 分配并初始化一个 const int
const int *pci = new const int(1024);
// 分配并默认初始化一个 const 的空 string
const string *pcs = new const string;
```

460 类似其他任何 `const` 对象，一个动态分配的 `const` 对象必须进行初始化。对于一个定义了默认构造函数（参见 7.1.4 节，第 236 页）的类类型，其 `const` 动态对象可以隐式初始化，而其他类型的对象就必须显式初始化。由于分配的对象是 `const` 的，`new` 返回的指针是一个指向 `const` 的指针（参见 2.4.2 节，第 56 页）。

内存耗尽

虽然现代计算机通常都配备大容量内存，但是自由空间被耗尽的情况还是有可能发生。一旦一个程序用光了它所有可用的内存，`new` 表达式就会失败。默认情况下，如果 `new` 不能分配所要求的内存空间，它会抛出一个类型为 `bad_alloc`（参见 5.6 节，第 173 页）的异常。我们可以改变使用 `new` 的方式来阻止它抛出异常：

```
// 如果分配失败，new 返回一个空指针
int *p1 = new int; // 如果分配失败，new 抛出 std::bad_alloc
int *p2 = new (nothrow) int; // 如果分配失败，new 返回一个空指针
```

我们称这种形式的 new 为定位 new (placement new)，其原因我们将在 19.1.2 节（第 729 页）中解释。定位 new 表达式允许我们向 new 传递额外的参数。在此例中，我们传递给它一个由标准库定义的名为 noexcept 的对象。如果将 noexcept 传递给 new，我们的意图是告诉它不能抛出异常。如果这种形式的 new 不能分配所需内存，它会返回一个空指针。bad_alloc 和 noexcept 都定义在头文件 new 中。

释放动态内存

为了防止内存耗尽，在动态内存使用完毕后，必须将其归还给系统。我们通过 delete 表达式（delete expression）来将动态内存归还给系统。delete 表达式接受一个指针，指向我们想要释放的对象：

```
delete p; // p 必须指向一个动态分配的对象或是一个空指针
```

与 new 类型类似，delete 表达式也执行两个动作：销毁给定的指针指向的对象；释放对应的内存。

指针值和 delete

我们传递给 delete 的指针必须指向动态分配的内存，或者是一个空指针（参见 2.3.2 节，第 48 页）。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：

```
int i, *pi1 = &i, *pi2 = nullptr;
double *pd = new double(33), *pd2 = pd;
delete i;      // 错误: i 不是一个指针
delete pi1;    // 未定义: pi1 指向一个局部变量
delete pd;     // 正确
delete pd2;    // 未定义: pd2 指向的内存已经被释放了
delete pi2;    // 正确: 释放一个空指针总是没有错误的
```

对于 delete i 的请求，编译器会生成一个错误信息，因为它知道 i 不是一个指针。执行 delete pi1 和 pd2 所产生的错误则更具潜在危害：通常情况下，编译器不能分辨一个指针指向的是静态还是动态分配的对象。类似的，编译器也不能分辨一个指针所指向的内存是否已经被释放了。对于这些 delete 表达式，大多数编译器会编译通过，尽管它们是错误的。

虽然一个 const 对象的值不能被改变，但它本身是可以被销毁的。如同任何其他动态对象一样，想要释放一个 const 动态对象，只要 delete 指向它的指针即可：

```
const int *pci = new const int(1024);
delete pci; // 正确: 释放一个 const 对象
```

动态对象的生存期直到被释放时为止

如 12.1.1 节（第 402 页）所述，由 shared_ptr 管理的内存在线程最后一个 shared_ptr 销毁时会被自动释放。但对于通过内置指针类型来管理的内存，就不是这样了。对于一个由内置指针管理的动态对象，直到被显式释放之前它都是存在的。

返回指向动态内存的指针（而不是智能指针）的函数给其调用者增加了一个额外负担——调用者必须记得释放内存：

```
// factory 返回一个指针，指向一个动态分配的对象
Foo* factory(T arg)
```

```

{
    // 视情况处理 arg
    return new Foo(arg); // 调用者负责释放此内存
}

```

类似我们之前定义的 `factory` 函数（参见 12.1.1 节，第 403 页），这个版本的 `factory` 分配一个对象，但并不 `delete` 它。`factory` 的调用者负责在不需要此对象时释放它。不幸的是，调用者经常忘记释放对象：

```

void use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p 但不 delete 它
} // p 离开了它的作用域，但它所指向的内存没有被释放！

```

此处，`use_factory` 函数调用 `factory`，后者分配一个类型为 `Foo` 的新对象。当 `use_factory` 返回时，局部变量 `p` 被销毁。此变量是一个内置指针，而不是一个智能指针。

与类类型不同，内置类型的对象被销毁时什么也不会发生。特别是，当一个指针离开其作用域时，它所指向的对象什么也不会发生。如果这个指针指向的是动态内存，那么内存将不会被自动释放。



由内置指针(而不是智能指针)管理的动态内存存在被显式释放前一直都会存在。

WARNING

462>

在本例中，`p` 是指向 `factory` 分配的内存的唯一指针。一旦 `use_factory` 返回，程序就没有办法释放这块内存了。根据整个程序的逻辑，修正这个错误的正确方法是在 `use_factory` 中记得释放内存：

```

void use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p
    delete p; // 现在记得释放内存，我们已经不需要它了
}

```

还有一种可能，我们的系统中的其他代码要使用 `use_factory` 所分配的对象，我们就应该修改此函数，让它返回一个指针，指向它分配的内存：

```

Foo* use_factory(T arg)
{
    Foo *p = factory(arg);
    // 使用 p
    return p; // 调用者必须释放内存
}

```

小心：动态内存的管理非常容易出错

使用 `new` 和 `delete` 管理动态内存存在三个常见问题：

1. 忘记 `delete` 内存。忘记释放动态内存会导致人们常说的“内存泄漏”问题，因为这种内存永远不可能被归还给自由空间了。查找内存泄露错误是非常困难的，因为通常应用程序运行很长一段时间后，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。通过在释放内存后将指针置为空，有时可以检测出这

种错误。

3. 同一块内存释放两次。当有两个指针指向相同的动态分配对象时，可能发生这种错误。如果对其中一个指针进行了 `delete` 操作，对象的内存就被归还给自由空间了。如果我们随后又 `delete` 第二个指针，自由空间就可能被破坏。

相对于查找和修正这些错误来说，制造出这些错误要简单得多。

Best Practices

坚持只使用智能指针，就可以避免所有这些问题。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……

当我们 `delete` 一个指针后，指针值就变为无效了。虽然指针已经无效，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。在 `delete` 之后，指针就变成了人们所说的空悬指针（dangling pointer），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。◀ 463

未初始化指针（参见 2.3.2 节，第 49 页）的所有缺点空悬指针也都有。有一种方法可以避免空悬指针的问题：在指针即将要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之后，就没有机会继续使用指针了。如果我们需要保留指针，可以在 `delete` 之后将 `nullptr` 赋予指针，这样就清楚地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。例如：

```
int *p(new int(42)); // p 指向动态内存
auto q = p;          // p 和 q 指向相同的内存
delete p;            // p 和 q 均变为无效
p = nullptr;         // 指出 p 不再绑定到任何对象
```

本例中 `p` 和 `q` 指向相同的动态分配的对象。我们 `delete` 此内存，然后将 `p` 置为 `nullptr`，指出它不再指向任何对象。但是，重置 `p` 对 `q` 没有任何作用，在我们释放 `p` 所指向的（同时也是 `q` 所指向的！）内存时，`q` 也变为无效了。在实际系统中，查找指向相同内存的所有指针是异常困难的。

12.1.2 节练习

练习 12.6：编写函数，返回一个动态分配的 `int` 的 `vector`。将此 `vector` 传递给另一个函数，这个函数读取标准输入，将读入的值保存在 `vector` 元素中。再将 `vector` 传递给另一个函数，打印读入的值。记得在恰当的时刻 `delete vector`。

练习 12.7：重做上一题，这次使用 `shared_ptr` 而不是内置指针。

练习 12.8：下面的函数是否有错误？如果有，解释错误原因。

```
bool b() {
    int* p = new int;
    // ...
```

```

    return p;
}

```

练习 12.9：解释下面代码执行的结果：

```

int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;

```

464 12.1.3 shared_ptr 和 new 结合使用

如前所述，如果我们不初始化一个智能指针，它就会被初始化为一个空指针。如表 12.3 所示，我们还可以用 new 返回的指针来初始化智能指针：

```

shared_ptr<double> p1; // shared_ptr 可以指向一个 double
shared_ptr<int> p2(new int(42)); // p2 指向一个值为 42 的 int

```

接受指针参数的智能指针构造函数是 *explicit* 的（参见 7.5.4 节，第 265 页）。因此，我们不能将一个内置指针隐式转换为一个智能指针，必须使用直接初始化形式（参见 3.2.1 节，第 76 页）来初始化一个智能指针：

```

shared_ptr<int> p1 = new int(1024); // 错误：必须使用直接初始化形式
shared_ptr<int> p2(new int(1024)); // 正确：使用了直接初始化形式

```

p1 的初始化隐式地要求编译器用一个 new 返回的 int* 来创建一个 shared_ptr。由于我们不能进行内置指针到智能指针间的隐式转换，因此这条初始化语句是错误的。出于相同的原因，一个返回 shared_ptr 的函数不能在其返回语句中隐式转换一个普通指针：

```

shared_ptr<int> clone(int p) {
    return new int(p); // 错误：隐式转换为 shared_ptr<int>
}

```

我们必须将 shared_ptr 显式绑定到一个想要返回的指针上：

```

shared_ptr<int> clone(int p) {
    // 正确：显式地用 int* 创建 shared_ptr<int>
    return shared_ptr<int>(new int(p));
}

```

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，但是为了这样做，必须提供自己的操作来替代 delete。我们将在 12.1.4 节（第 415 页）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

<code>shared_ptr<T> p(q)</code>	<code>p</code> 管理内置指针 <code>q</code> 所指向的对象； <code>q</code> 必须指向 new 分配的内存，且能够转换为 <code>T*</code> 类型
<code>shared_ptr<T> p(u)</code>	<code>p</code> 从 <code>unique_ptr u</code> 那里接管了对象的所有权；将 <code>u</code> 置为空
<code>shared_ptr<T> p(q, d)</code>	<code>p</code> 接管了内置指针 <code>q</code> 所指向的对象的所有权。 <code>q</code> 必须能转换为 <code>T*</code> 类型（参见 4.11.2 节，第 143 页）。 <code>p</code> 将使用可调用对象 <code>d</code> （参见 10.3.2 节，第 346 页）来代替 <code>delete</code>

续表

<code>shared_ptr<T> p(p2, d)</code>	如表 12.2 所示, p 是 shared_ptr p2 的拷贝, 唯一的区别是 p 将用可调用对象 d 来代替 delete
<code>p.reset()</code>	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
<code>p.reset(q)</code>	
<code>p.reset(q, d)</code>	

不要混合使用普通指针和智能指针……



`shared_ptr` 可以协调对象的析构, 但这仅限于其自身的拷贝 (也是 `shared_ptr`) 之间。这也是为什么我们推荐使用 `make_shared` 而不是 `new` 的原因。这样, 我们就能在分配对象的同时就将 `shared_ptr` 与之绑定, 从而避免了无意中将同一块内存绑定到多个独立创建的 `shared_ptr` 上。

考虑下面对 `shared_ptr` 进行操作的函数:

```
// 在函数被调用时 ptr 被创建并初始化
void process(shared_ptr<int> ptr)
{
    // 使用 ptr
} // ptr 离开作用域, 被销毁
```

`process` 的参数是传值方式传递的, 因此实参会 ◀465 被拷贝到 `ptr` 中。拷贝一个 `shared_ptr` 会递增其引用计数, 因此, 在 `process` 运行过程中, 引用计数值至少为 2。当 `process` 结束时, `ptr` 的引用计数会递减, 但不会变为 0。因此, 当局部变量 `ptr` 被销毁时, `ptr` 指向的内存不会被释放。

使用此函数的正确方法是传递给它一个 `shared_ptr`:

```
shared_ptr<int> p(new int(42)); // 引用计数为 1
process(p); // 拷贝 p 会递增它的引用计数; 在 process 中引用计数值为 2
int i = *p; // 正确: 引用计数值为 1
```

虽然不能传递给 `process` 一个内置指针, 但可以传递给它一个 (临时的) `shared_ptr`, 这个 `shared_ptr` 是用一个内置指针显式构造的。但是, 这样做很可能会导致错误:

```
int *x(new int(1024));           // 危险: x 是一个普通指针, 不是一个智能指针
process(x); // 错误: 不能将 int* 转换为一个 shared_ptr<int>
process(shared_ptr<int>(x)); // 合法的, 但内存会被释放!
int j = *x; // 未定义的: x 是一个空悬指针!
```

在上面的调用中, 我们将一个临时 `shared_ptr` 传递给 `process`。当这个调用所在的表达式结束时, 这个临时对象就被销毁了。销毁这个临时变量会递减引用计数, 此时引用计数就变为 0 了。因此, 当临时对象被销毁时, 它所指向的内存会被释放。

但 `x` 继续指向 (已经释放的) 内存, 从而变成一个空悬指针。如果试图使用 `x` 的值, 其行为是未定义的。

当将一个 `shared_ptr` 绑定到一个普通指针时, 我们就将内存的管理责任交给了这个 `shared_ptr`。一旦这样做了, 我们就不应该再使用内置指针来访问 `shared_ptr` 所指向的内存了。

466



WARNING

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。

467

……也不要使用 get 初始化另一个智能指针或为智能指针赋值

智能指针类型定义了一个名为 `get` 的函数（参见表 12.1），它返回一个内置指针，指向智能指针管理的对象。此函数是为了这样一种情况而设计的：我们需要向不能使用智能指针的代码传递一个内置指针。使用 `get` 返回的指针的代码不能 `delete` 此指针。

虽然编译器不会给出错误信息，但将另一个智能指针也绑定到 `get` 返回的指针上是错误的：

```
shared_ptr<int> p(new int(42)); // 引用计数为 1
int *q = p.get(); // 正确：但使用 q 时要注意，不要让它管理的指针被释放
{ // 新程序块
    // 未定义：两个独立的 shared_ptr 指向相同的内存
    shared_ptr<int>(q);
} // 程序块结束，q 被销毁，它指向的内存被释放
int foo = *p; // 未定义：p 指向的内存已经被释放了
```

在本例中，`p` 和 `q` 指向相同的内存。由于它们是相互独立创建的，因此各自的引用计数都是 1。当 `q` 所在的程序块结束时，`q` 被销毁，这会导致 `q` 指向的内存被释放。从而 `p` 变成一个空悬指针，意味着当我们试图使用 `p` 时，将发生未定义的行为。而且，当 `p` 被销毁时，这块内存会被第二次 `delete`。



WARNING

`get` 用来将指针的访问权限传递给代码，你只有在确定代码不会 `delete` 指针的情况下，才能使用 `get`。特别是，永远不要用 `get` 初始化另一个智能指针或者为另一个智能指针赋值。

其他 `shared_ptr` 操作

`shared_ptr` 还定义了其他一些操作，参见表 12.2 和表 12.3 所示。我们可以用 `reset` 来将一个新的指针赋予一个 `shared_ptr`：

```
p = new int(1024);           // 错误：不能将一个指针赋予 shared_ptr
p.reset(new int(1024));     // 正确：p 指向一个新对象
```

与赋值类似，`reset` 会更新引用计数，如果需要的话，会释放 `p` 指向的对象。`reset` 成员经常与 `unique` 一起使用，来控制多个 `shared_ptr` 共享的对象。在改变底层对象之前，我们检查自己是否是当前对象仅有的用户。如果不是，在改变之前要制作一份新的拷贝：

```
if (!p.unique())
    p.reset(new string(*p)); // 我们不是唯一用户；分配新的拷贝
*p += newVal; // 现在我们知道我们是唯一的用户，可以改变对象的值
```

467

12.1.3 节练习

练习 12.10：下面的代码调用了第 413 页中定义的 `process` 函数，解释此调用是否正确。如果不正确，应如何修改？

```
shared_ptr<int> p(new int(42));
```

```
process(shared_ptr<int>(p));
```

练习 12.11: 如果我们像下面这样调用 process，会发生什么？

```
process(shared_ptr<int>(p.get()));
```

练习 12.12: p 和 q 的定义如下，对于接下来的对 process 的每个调用，如果合法，解释它做了什么，如果不合法，解释错误原因：

```
auto p = new int();
auto sp = make_shared<int>();
(a) process(sp);
(b) process(new int());
(c) process(p);
(d) process(shared_ptr<int>(p));
```

练习 12.13: 如果执行下面的代码，会发生什么？

```
auto sp = make_shared<int>();
auto p = sp.get();
delete p;
```

12.1.4 智能指针和异常



5.6.2 节（第 175 页）中介绍了使用异常处理的程序能在异常发生后令程序流程继续，我们注意到，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。

如果使用智能指针，即使程序块过早结束，智能指针类也能确保在内存不再需要时将其释放，：

```
void f()
{
    shared_ptr<int> sp(new int(42)); // 分配一个新对象
    // 这段代码抛出一个异常，且在 f 中未被捕获
} // 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。在上面的程序中，sp 是一个 shared_ptr，因此 sp 销毁时会检查引用计数。在此例中，sp 是指向这块内存的唯一指针，因此内存会被释放掉。

与之相对的，当发生异常时，我们直接管理的内存是不会自动释放的。如果使用内置指针管理内存，且在 new 之后在对应的 delete 之前发生了异常，则内存不会被释放：

```
void f()
{
    int *ip = new int(42); // 动态分配一个新对象
    // 这段代码抛出一个异常，且在 f 中未被捕获
    delete ip;           // 在退出之前释放内存
}
```

468

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



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（参见 12.1.1 节，第 402 页），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++ 两种语言设计的类，通常都要求用户显式地释放所使用的任何资源。

那些分配了资源，而又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——程序员非常容易忘记释放资源。类似的，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漏。

与管理动态内存类似，我们通常可以使用类似的技术来管理不具有良好定义的析构函数的类。例如，假定我们正在使用一个 C 和 C++ 都使用的网络库，使用这个库的代码可能是这样的：

```
struct destination;           // 表示我们正在连接什么
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
    // 如果我们在 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，已被证明是一种有效的方法。



使用我们自己的释放操作

469 默认情况下，`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`，我们必须首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。在本例中，我们的删除器必须接受单个类型为 `connection*` 的参数：

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

当我们创建一个 `shared_ptr` 时，可以传递一个（可选的）指向删除器函数的参数（参见 6.7 节，第 221 页）：

```
void f(destination &d /* 其他参数 */)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由异常而退出），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。接下来，end_connection 会调用 disconnect，从而确保连接被关闭。如果 f 正常退出，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 同样会被销毁，从而连接被关闭。

注意：智能指针陷阱

智能指针可以提供对动态分配的内存安全而又方便的管理，但这建立在正确使用的前提下。为了正确使用智能指针，我们必须坚持一些基本规范：

- 不使用相同的内置指针值初始化（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。
- 如果你使用 get() 返回的指针，记住当最后一个对应的智能指针销毁后，你的指针就变为无效了。
- 如果你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（参见 12.1.4 节，第 415 页和 12.1.5 节，第 419 页）。

12.1.4 节练习

练习 12.14：编写你自己版本的用 shared_ptr 管理 connection 的函数。

练习 12.15：重写第一题的程序，用 lambda（参见 10.3.2 节，第 346 页）代替 end_connection 函数。

12.1.5 unique_ptr

< 470

一个 unique_ptr “拥有” 它所指向的对象。与 shared_ptr 不同，某个时刻只能有一个 unique_ptr 指向一个给定对象。当 unique_ptr 被销毁时，它所指向的对象也被销毁。表 12.4 列出了 unique_ptr 特有的操作。与 shared_ptr 相同的操作列在表 12.1（第 401 页）中。

C++
11

与 shared_ptr 不同，没有类似 make_shared 的标准库函数返回一个 unique_ptr。当我们定义一个 unique_ptr 时，需要将其绑定到一个 new 返回的指针上。类似 shared_ptr，初始化 unique_ptr 必须采用直接初始化形式：

```
unique_ptr<double> p1; // 可以指向一个 double 的 unique_ptr  
unique_ptr<int> p2(new int(42)); // p2 指向一个值为 42 的 int
```

由于一个 unique_ptr 拥有它指向的对象，因此 unique_ptr 不支持普通的拷贝或赋值操作：

```
unique_ptr<string> p1(new string("Stegosaurus"));  
unique_ptr<string> p2(p1); // 错误：unique_ptr 不支持拷贝  
unique_ptr<string> p3;  
p3 = p2; // 错误：unique_ptr 不支持赋值
```

表 12.4: unique_ptr 操作 (另参见表 12.1, 第 401 页)	
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;
unique_ptr<T, D> u2	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> u(d)	空 unique_ptr, 指向类型为 T 的对象, 用类型为 D 的对象 d 替代 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()	释放 u 指向的对象
u.reset(q)	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
u.reset(nullptr)	

虽然我们不能拷贝或赋值 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:

```
// 将所有权从 p1 (指向 string Stegosaurus) 转移给 p2
unique_ptr<string> p2(p1.release()); // release 将 p1 置为空
unique_ptr<string> p3(new string("Trex"));
// 将所有权从 p3 转移给 p2
p2.reset(p3.release()); // reset 释放了 p2 原来指向的内存
```

release 成员返回 unique_ptr 当前保存的指针并将其置为空。因此, p2 被初始化为 p1 原来保存的指针, 而 p1 被置为空。

471 > reset 成员接受一个可选的指针参数, 令 unique_ptr 重新指向给定的指针。如果 unique_ptr 不为空, 它原来指向的对象被释放。因此, 对 p2 调用 reset 释放了用 "Stegosaurus" 初始化的 string 所使用的内存, 将 p3 对指针的所有权转移给 p2, 并将 p3 置为空。

调用 release 会切断 unique_ptr 和它原来管理的对象间的联系。release 返回的指针通常被用来初始化另一个智能指针或给另一个智能指针赋值。在本例中, 管理内存的责任简单地从一个智能指针转移给另一个。但是, 如果我们不用另一个智能指针来保存 release 返回的指针, 我们的程序就要负责资源的释放:

```
p2.release(); // 错误: p2 不会释放内存, 而且我们丢失了指针
auto p = p2.release(); // 正确, 但我们必须记得 delete(p)
```

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: 我们可以拷贝或赋值一个将要被销毁的 unique_ptr。最常见的例子是从函数返回一个 unique_ptr:

```
unique_ptr<int> clone(int p) {
    // 正确: 从 int* 创建一个 unique_ptr<int>
    return unique_ptr<int>(new int(p));
}
```

还可以返回一个局部对象的拷贝:

```
unique_ptr<int> clone(int p) {
    unique_ptr<int> ret(new int (p));
    // ...
    return ret;
}
```

对于两段代码，编译器都知道要返回的对象将要被销毁。在此情况下，编译器执行一种特殊的“拷贝”，我们将在 13.6.2 节（第 473 页）中介绍它。

向后兼容：auto_ptr

标准库的较早版本包含了一个名为 `auto_ptr` 的类，它具有 `unique_ptr` 的部分特性，但不是全部。特别是，我们不能在容器中保存 `auto_ptr`，也不能从函数中返回 `auto_ptr`。

虽然 `auto_ptr` 仍是标准库的一部分，但编写程序时应该使用 `unique_ptr`。

向 unique_ptr 传递删除器

类似 `shared_ptr`, `unique_ptr` 默认情况下用 `delete` 释放它指向的对象。与 `shared_ptr` 一样，我们可以重载一个 `unique_ptr` 中默认的删除器（参见 12.1.4 节，第 415 页）。但是，`unique_ptr` 管理删除器的方式与 `shared_ptr` 不同，其原因我们将在 16.1.6 节（第 599 页）中介绍。

< 472

重载一个 `unique_ptr` 中的删除器会影响到 `unique_ptr` 类型以及如何构造（或 `reset`）该类型的对象。与重载关联容器的比较操作（参见 11.2.2 节，第 378 页）类似，我们必须在尖括号中 `unique_ptr` 指向类型之后提供删除器类型。在创建或 `reset` 一个这种 `unique_ptr` 类型的对象时，必须提供一个指定类型的可调用对象（删除器）：

```
// p 指向一个类型为 objT 的对象，并使用一个类型为 delT 的对象释放 objT 对象
// 它会调用一个名为 fcn 的 delT 类型对象
unique_ptr<objT, delT> p (new objT, fcn);
```

作为一个更具体的例子，我们将重写连接程序，用 `unique_ptr` 来代替 `shared_ptr`，如下所示：

```
void f(destination &d /* 其他需要的参数 */)
{
    connection c = connect(&d); // 打开连接
    // 当 p 被销毁时，连接将会关闭
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
    // 使用连接
    // 当 f 退出时（即使是由于异常而退出），connection 会被正确关闭
}
```

在本例中我们使用了 `decltype`（参见 2.5.3 节，第 62 页）来指明函数指针类型。由于 `decltype(end_connection)` 返回一个函数类型，所以我们必须添加一个*来指出我们正在使用该类型的一个指针（参见 6.7 节，第 223 页）。

12.1.5 节练习

练习 12.16：如果你试图拷贝或赋值 `unique_ptr`，编译器并不总是能给出易于理解的错误信息。编写包含这种错误的程序，观察编译器如何诊断这种错误。

练习 12.17：下面的 `unique_ptr` 声明中，哪些是合法的，哪些可能导致后续的程序错误？解释每个错误的问题在哪里。

```

int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
(a) IntP p0(ix);           (b) IntP p1(pi);
(c) IntP p2(pi2);         (d) IntP p3(&ix);
(e) IntP p4(new int(2048)); (f) IntP p5(p2.get());

```

练习 12.18: `shared_ptr` 为什么没有 `release` 成员?



12.1.6 weak_ptr

473

C++
11

`weak_ptr` (见表 12.5) 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, 因此, `weak_ptr` 的名字抓住了这种智能指针“弱”共享对象的特点。

表 12.5: `weak_ptr`

<code>weak_ptr<T> w</code>	空 <code>weak_ptr</code> 可以指向类型为 <code>T</code> 的对象
<code>weak_ptr<T> w(sp)</code>	与 <code>shared_ptr</code> <code>sp</code> 指向相同对象的 <code>weak_ptr</code> 。 <code>T</code> 必须能转换为 <code>sp</code> 指向的类型
<code>w = p</code>	<code>p</code> 可以是一个 <code>shared_ptr</code> 或一个 <code>weak_ptr</code> 。赋值后 <code>w</code> 与 <code>p</code> 共享对象
<code>w.reset()</code>	将 <code>w</code> 置为空
<code>w.use_count()</code>	与 <code>w</code> 共享对象的 <code>shared_ptr</code> 的数量
<code>w.expired()</code>	若 <code>w.use_count()</code> 为 0, 返回 <code>true</code> , 否则返回 <code>false</code>
<code>w.lock()</code>	如果 <code>expired</code> 为 <code>true</code> , 返回一个空 <code>shared_ptr</code> ; 否则返回一个指向 <code>w</code> 的对象的 <code>shared_ptr</code>

当我们创建一个 `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

```

本例中 `wp` 和 `p` 指向相同的对象。由于是弱共享, 创建 `wp` 不会改变 `p` 的引用计数; `wp` 指向的对象可能被释放掉。

由于对象可能存在, 我们不能使用 `weak_ptr` 直接访问对象, 而必须调用 `lock`。此函数检查 `weak_ptr` 指向的对象是否仍存在。如果存在, `lock` 返回一个指向共享对象的 `shared_ptr`。与任何其他 `shared_ptr` 类似, 只要此 `shared_ptr` 存在, 它所指向的底层对象也就会一直存在。例如:

```

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
}

```

在这段代码中, 只有当 `lock` 调用返回 `true` 时我们才会进入 `if` 语句体。在 `if` 中, 使用 `np` 访问共享对象是安全的。

核查指针类

作为 `weak_ptr` 用途的一个展示, 我们将为 `StrBlob` 类定义一个伴随指针类。我们

的指针类将命名为 StrBlobPtr，会保存一个 `weak_ptr`，指向 `StrBlob` 的 `data` 成员，这是初始化时提供给它的。通过使用 `weak_ptr`，不会影响一个给定的 `StrBlob` 所指向的 `vector` 的生存期。但是，可以阻止用户访问一个不再存在的 `vector` 的企图。 ◀ 474

`StrBlobPtr` 会有两个数据成员：`wptr`，或者为空，或者指向一个 `StrBlob` 中的 `vector`；`curr`，保存当前对象所表示的元素的下标。类似它的伴随类 `StrBlob`，我们的指针类也有一个 `check` 成员来检查解引用 `StrBlobPtr` 是否安全：

```
// 对于访问一个不存在元素的尝试，StrBlobPtr 抛出一个异常
class StrBlobPtr {
public:
    StrBlobPtr(): curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0):
        wptr(a.data), curr(sz) { }
    std::string& deref() const;
    StrBlobPtr& incr(); // 前缀递增
private:
    // 若检查成功，check 返回一个指向 vector 的 shared_ptr
    std::shared_ptr<std::vector<std::string>>
        check(std::size_t, const std::string&) const;
    // 保存一个 weak_ptr，意味着底层 vector 可能会被销毁
    std::weak_ptr<std::vector<std::string>> wptr;
    std::size_t curr; // 在数组中的当前位置
};
```

默认构造函数生成一个空的 `StrBlobPtr`。其构造函数初始化列表（参见 7.1.4 节，第 237 页）将 `curr` 显式初始化为 0，并将 `wptr` 隐式初始化为一个空 `weak_ptr`。第二个构造函数接受一个 `StrBlob` 引用和一个可选的索引值。此构造函数初始化 `wptr`，令其指向给定 `StrBlob` 对象的 `shared_ptr` 中的 `vector`，并将 `curr` 初始化为 `sz` 的值。我们使用了默认参数（参见 6.5.1 节，第 211 页），表示默认情况下将 `curr` 初始化为第一个元素的下标。我们将会看到，`StrBlob` 的 `end` 成员将会用到参数 `sz`。

值得注意的是，我们不能将 `StrBlobPtr` 绑定到一个 `const StrBlob` 对象。这个限制是由于构造函数接受一个非 `const StrBlob` 对象的引用而导致的。

`StrBlobPtr` 的 `check` 成员与 `StrBlob` 中的同名成员不同，它还要检查指针指向的 `vector` 是否还存在：

```
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i, const std::string &msg) const
{
    auto ret = wptr.lock(); // vector 还存在吗？
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // 否则，返回指向 vector 的 shared_ptr
}
```

由于一个 `weak_ptr` 不参与其对应的 `shared_ptr` 的引用计数，`StrBlobPtr` 指向的 `vector` 可能已经被释放了。如果 `vector` 已销毁，`lock` 将返回一个空指针。在本例中，任何 `vector` 的引用都会失败，于是抛出一个异常。否则，`check` 会检查给定索引，如果索引合法，`check` 返回从 `lock` 获得的 `shared_ptr`。 ◀ 475

指针操作

我们将在第 14 章学习如何定义自己的运算符。现在，我们将定义名为 deref 和 incr 的函数，分别用来解引用和递增 StrBlobPtr。

deref 成员调用 check，检查使用 vector 是否安全以及 curr 是否在合法范围内：

```
std::string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr]; // (*p) 是对象所指向的 vector
}
```

如果 check 成功，p 就是一个 shared_ptr，指向 StrBlobPtr 所指向的 vector。表达式 (*p)[curr] 解引用 shared_ptr 来获得 vector，然后使用下标运算符提取并返回 curr 位置上的元素。

incr 成员也调用 check：

```
// 前缀递增：返回递增后的对象的引用
StrBlobPtr& StrBlobPtr::incr()
{
    // 如果 curr 已经指向容器的尾后位置，就不能递增它
    check(curr, "increment past end of StrBlobPtr");
    ++curr; // 推进当前位置
    return *this;
}
```

当然，为了访问 data 成员，我们的指针类必须声明为 StrBlob 的 friend（参见 7.3.4 节，第 250 页）。我们还要为 StrBlob 类定义 begin 和 end 操作，返回一个指向它自身的 StrBlobPtr：

```
// 对于 StrBlob 中的友元声明来说，此前置声明是必要的
class StrBlobPtr;
class StrBlob {
    friend class StrBlobPtr;
    // 其他成员与 12.1.1 节（第 405 页）中声明相同
    // 返回指向首元素和尾后元素的 StrBlobPtr
    StrBlobPtr begin() { return StrBlobPtr(*this); }
    StrBlobPtr end()
    { auto ret = StrBlobPtr(*this, data->size());
      return ret; }
};
```

12.1.6 节练习

练习 12.19： 定义你自己版本的 StrBlobPtr，更新 StrBlob 类，加入恰当的 friend 声明及 begin 和 end 成员。

练习 12.20： 编写程序，逐行读入一个输入文件，将内容存入一个 StrBlob 中，用一个 StrBlobPtr 打印出 StrBlob 中的每个元素。

练习 12.21： 也可以这样编写 StrBlobPtr 的 deref 成员：

```
std::string& deref() const
```

```
{ return (*check(curr, "dereference past end))[curr]; }
```

你认为哪个版本更好？为什么？

练习 12.22: 为了能让 `StrBlobPtr` 使用 `const StrBlob`, 你觉得应该如何修改？定义一个名为 `ConstStrBlobPtr` 的类，使其能够指向 `const StrBlob`。



12.2 动态数组

`new` 和 `delete` 运算符一次分配/释放一个对象，但某些应用需要一次为很多对象分配内存的功能。例如，`vector` 和 `string` 都是在连续内存中保存它们的元素，因此，当容器需要重新分配内存时（参见 9.4 节，第 317 页），必须一次性为很多元素分配内存。

为了支持这种需求，C++语言和标准库提供了两种一次分配一个对象数组的方法。C++语言定义了另一种 `new` 表达式语法，可以分配并初始化一个对象数组。标准库中包含一个名为 `allocator` 的类，允许我们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，原因我们将在 12.2.2 节（第 427 页）中解释。

很多（可能是大多数）应用都没有直接访问动态数组的需求。当一个应用需要可变数量的对象时，我们在 `StrBlob` 中所采用的方法几乎总是更简单、更快速并且更安全的——即，使用 `vector`（或其他标准库容器）。如我们将在 13.6 节（第 470 页）中看到的，使用标准库容器的优势在新标准下更为显著。在支持新标准的标准库中，容器操作比之前的版本要快速得多。

Best Practices

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、更不容易出现内存管理错误并且可能有更好的性能。

如前所述，使用容器的类可以使用默认版本的拷贝、赋值和析构操作（参见 7.1.5 节，第 239 页）。分配动态数组的类则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。



直到学习完第 13 章，不要在类内的代码中分配动态内存。



12.2.1 new 和数组

477

为了让 `new` 分配一个对象数组，我们要在类型名之后跟一对方括号，在其中指明要分配的对象的数目。在下例中，`new` 分配要求数量的对象并（假定分配成功后）返回指向第一个对象的指针：

```
// 调用 get_size 确定分配多少个 int
int *pia = new int[get_size()]; // pia 指向第一个 int
```

方括号中的大小必须是整型，但不必是常量。

也可以用一个表示数组类型的类型别名（参见 2.5.1 节，第 60 页）来分配一个数组，这样，`new` 表达式中就不需要方括号了：

```
typedef int arrT[42];      // arrT 表示 42 个 int 的数组类型
int *p = new arrT;         // 分配一个 42 个 int 的数组；p 指向第一个 int
```

在本例中，`new` 分配一个 `int` 数组，并返回指向第一个 `int` 的指针。即使这段代码中没

有方括号，编译器执行这个表达式时还是会用 `new []`。即，编译器执行如下形式：

```
int *p = new int[42];
```

分配一个数组会得到一个元素类型的指针

虽然我们通常称 `new T[]` 分配的内存为“动态数组”，但这种叫法某种程度上有些误导。当用 `new` 分配一个数组时，我们并未得到一个数组类型的对象，而是得到一个数组元素类型的指针。即使我们使用类型别名定义了一个数组类型，`new` 也不会分配一个数组类型的对象。在上例中，我们正在分配一个数组的事实甚至都是不可见的——连 `[num]` 都没有。`new` 返回的是一个元素类型的指针。

由于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`（参见 3.5.3 节，第 106 页）。这些函数使用数组维度（回忆一下，维度是数组类型的一部分）来返回指向首元素和尾后元素的指针。出于相同的原因，也不能用范围 `for` 语句来处理（所谓的）动态数组中的元素。



WARNING

要记住我们所说的动态数组并不是数组类型，这是很重要的。

初始化动态分配对象的数组

默认情况下，`new` 分配的对象，不管是单个分配的还是数组中的，都是默认初始化的。可以对数组中的元素进行值初始化（参见 3.3.1 节，第 88 页），方法是在大小之后跟一对空括号。

```
int *pia = new int[10];           // 10 个未初始化的 int
int *pia2 = new int[10]();        // 10 个值初始化为 0 的 int
string *psa = new string[10];     // 10 个空 string
string *psa2 = new string[10]();   // 10 个空 string
```

478 在新标准中，我们还可以提供一个元素初始化器的花括号列表：

```
// 10 个 int 分别用列表中对应的初始化器初始化
int *pia3 = new int[10]{0,1,2,3,4,5,6,7,8,9};
// 10 个 string，前 4 个用给定的初始化器初始化，剩余的进行值初始化
string *psa3 = new string[10]{"a", "an", "the", string(3,'x')};
```

与内置数组对象的列表初始化（参见 3.5.1 节，第 102 页）一样，初始化器会用来初始化动态数组中开始部分的元素。如果初始化器数目小于元素数目，剩余元素将进行值初始化。如果初始化器数目大于元素数目，则 `new` 表达式失败，不会分配任何内存。在本例中，`new` 会抛出一个类型为 `bad_array_new_length` 的异常。类似 `bad_alloc`，此类型定义在头文件 `new` 中。

C++ 11 虽然我们用空括号对数组中元素进行值初始化，但不能在括号中给出初始化器，这意味着不能用 `auto` 分配数组（参见 12.1.2 节，第 407 页）。

动态分配一个空数组是合法的

可以用任意表达式来确定要分配的对象的数目：

```
size_t n = get_size();    // get_size 返回需要的元素的数目
int* p = new int[n];      // 分配数组保存元素
for (int* q = p; q != p + n; ++q)
/* 处理数组 */;
```

这产生了一个有意思的问题：如果 `get_size` 返回 0，会发生什么？答案是代码仍能正常工作。虽然我们不能创建一个大小为 0 的静态数组对象，但当 `n` 等于 0 时，调用 `new[n]` 是合法的：

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

当我们用 `new` 分配一个大小为 0 的数组时，`new` 返回一个合法的非空指针。此指针保证与 `new` 返回的其他任何指针都不相同。对于零长度的数组来说，此指针就像尾后指针一样（参见 3.5.3 节，第 106 页），我们可以像使用尾后迭代器一样使用这个指针。可以用此指针进行比较操作，就像上面循环代码中那样。可以向此指针加上（或从此指针减去）0，也可以从此指针减去自身从而得到 0。但此指针不能解引用——毕竟它不指向任何元素。

在我们假想的循环中，若 `get_size` 返回 0，则 `n` 也是 0，`new` 会分配 0 个对象。`for` 循环中的条件会失败（`p` 等于 `q+n`，因为 `n` 为 0）。因此，循环体不会被执行。

释放动态数组

为了释放动态数组，我们使用一种特殊形式的 `delete`——在指针前加上一个空方括号对：

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

< 479

第二条语句销毁 `pa` 指向的数组中的元素，并释放对应的内存。数组中的元素按逆序销毁，即，最后一个元素首先被销毁，然后是倒数第二个，依此类推。

当我们释放一个指向数组的指针时，空方括号对是必需的：它指示编译器此指针指向一个对象数组的第一个元素。如果我们在 `delete` 一个指向数组的指针时忽略了方括号（或者在 `delete` 一个指向单一对象的指针时使用了方括号），其行为是未定义的。

回忆一下，当我们使用一个类型别名来定义一个数组类型时，在 `new` 表达式中不使用 `[]`。即使这样，在释放一个数组指针时也必须使用方括号：

```
typedef int arrT[42];    // arrT 是 42 个 int 的数组的类型别名
int *p = new arrT;       // 分配一个 42 个 int 的数组；p 指向第一个元素
delete [] p;             // 方括号是必需的，因为我们当初分配的是一个数组
```

不管外表如何，`p` 指向一个对象数组的首元素，而不是一个类型为 `arrT` 的单一对象。因此，在释放 `p` 时我们必须使用 `[]`。



如果我们在 `delete` 一个数组指针时忘记了方括号，或者在 `delete` 一个单一对象的指针时使用了方括号，编译器很可能不会给出警告。我们的程序可能在执行过程中在没有任何警告的情况下行为异常。

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 `unique_ptr` 版本。为了用一个 `unique_ptr` 管理动态数组，我们必须在对象类型后面跟一对空方括号：

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

类型说明符中的方括号 (`<int[]>`) 指出 `up` 指向一个 `int` 数组而不是一个 `int`。由于 `up` 指向一个数组，当 `up` 销毁它管理的指针时，会自动使用 `delete[]`。

指向数组的 `unique_ptr` 提供的操作与我们在 12.1.5 节（第 417 页）中使用的那些操作有一些不同，我们在表 12.6 中描述了这些操作。当一个 `unique_ptr` 指向一个数组时，我们不能使用点和箭头成员运算符。毕竟 `unique_ptr` 指向的是一个数组而不是单个对象，因此这些运算符是无意义的。另一方面，当一个 `unique_ptr` 指向一个数组时，我们可以使用下标运算符来访问数组中的元素：

```
for (size_t i = 0; i != 10; ++i)
    up[i] = i; // 为每个元素赋予一个新值
```

480 >

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*`（参见 4.11.2 节，第 143 页）

`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

与 `unique_ptr` 不同，`shared_ptr` 不直接支持管理动态数组。如果希望使用 `shared_ptr` 管理一个动态数组，必须提供自己定义的删除器：

```
// 为了使用 shared_ptr，必须提供一个删除器
shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });
sp.reset(); // 使用我们提供的 lambda 释放数组，它使用 delete[]
```

本例中我们传递给 `shared_ptr` 一个 `lambda`（参见 10.3.2 节，第 346 页）作为删除器，它使用 `delete[]` 释放数组。

如果未提供删除器，这段代码将是未定义的。默认情况下，`shared_ptr` 使用 `delete` 销毁它指向的对象。如果此对象是一个动态数组，对其使用 `delete` 所产生的问题与释放一个动态数组指针时忘记 `[]` 产生的问题一样（参见 12.2.1 节，第 425 页）。

`shared_ptr` 不直接支持动态数组管理这一特性会影响我们如何访问数组中的元素：

```
// shared_ptr 未定义下标运算符，并且不支持指针的算术运算
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i; // 使用 get 获取一个内置指针
```

`shared_ptr` 未定义下标运算符，而且智能指针类型不支持指针算术运算。因此，为了访问数组中的元素，必须用 `get` 获取一个内置指针，然后用它来访问数组元素。

12.2.1 节练习

练习 12.23： 编写一个程序，连接两个字符串字面常量，将结果保存在一个动态分配的 `char` 数组中。重写这个程序，连接两个标准库 `string` 对象。

练习 12.24： 编写一个程序，从标准输入读取一个字符串，存入一个动态分配的字符数组中。描述你的程序如何处理变长输入。测试你的程序，输入一个超出你分配的数组长度的字符串。

练习 12.25：给定下面的 new 表达式，你应该如何释放 pa？

```
int *pa = new int[10];
```



12.2.2 allocator 类

<481

new 有一些灵活性上的局限，其中一方面表现在它将内存分配和对象构造组合在了一起。类似的，delete 将对象析构和内存释放组合在了一起。我们分配单个对象时，通常希望将内存分配和对象初始化组合在一起。因为在这种情况下，我们几乎肯定知道对象应有什么值。

当分配一大块内存时，我们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）。

一般情况下，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。例如：

```
string *const p = new string[n]; // 构造 n 个空 string
string s;
string *q = p; // q 指向第一个 string
while (cin >> s && q != p + n)
    *q++ = s; // 赋予*q 一个新值
const size_t size = q - p; // 记住我们读取了多少个 string
// 使用数组
delete[] p; // p 指向一个数组；记得用 delete[] 来释放
```

new 表达式分配并初始化了 n 个 string。但是，我们可能不需要 n 个 string，少量 string 可能就足够了。这样，我们就可能创建了一些永远也用不到的对象。而且，对于那些确实要使用的对象，我们也在初始化之后立即赋予了它们新值。每个使用到的元素都被赋值了两次：第一次是在默认初始化时，随后是在赋值时。

更重要的是，那些没有默认构造函数的类就不能动态分配数组了。

allocator 类

标准库 **allocator** 类定义在头文件 `memory` 中，它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。表 12.7 概述了 allocator 支持的操作。在本节中，我们将介绍这些 allocator 操作。在 13.5 节（第 464 页），我们将看到如何使用这个类的典型例子。

类似 `vector`，`allocator` 是一个模板（参见 3.3 节，第 86 页）。为了定义一个 `allocator` 对象，我们必须指明这个 `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
```

这个 `allocate` 调用为 n 个 string 分配了内存。

482

表 12.7：标准库 allocator 类及其算法

allocator<T> a	定义了一个名为 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，这块内存保存了 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。在调用 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（参见 12.1.1 节，第 402 页）

allocator 分配未构造的内存

allocator 分配的内存是未构造的（unconstructed）。我们按需要在此内存中构造对象。在新标准库中，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。类似 make_shared 的参数（参见 12.1.1 节，第 401 页），这些额外参数必须是与构造的对象的类型相匹配的合法的初始化器：

```
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"); // *q 为 hi!
```

在早期版本的标准库中，construct 只接受两个参数：指向创建对象位置的指针和一个元素类型的值。因此，我们只能将一个元素拷贝到未构造空间中，而不能用元素类型的任何其他构造函数来构造一个元素。

还未构造对象的情况下就使用原始内存是错误的：

```
cout << *p << endl; // 正确：使用 string 的输出运算符
cout << *q << endl; // 灾难：q 指向未构造的内存！
```



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

当我们用完对象后，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（参见 12.1.1 节，第 402 页）：

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

在循环开始处，q 指向最后构造的元素之后的位置。我们在调用 destroy 之前对 q 进行了递减操作。因此，第一次调用 destroy 时，q 指向最后一个构造的元素。最后一步循环中我们 destroy 了第一个构造的元素，随后 q 将与 p 相等，循环结束。



我们只能对真正构造了的元素进行 destroy 操作。

一旦元素被销毁后，就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);
```

我们传递给 `deallocate` 的指针不能为空，它必须指向由 `allocate` 分配的内存。而且，传递给 `deallocate` 的大小参数必须与调用 `allocate` 分配内存时提供的大小参数具有一样的值。

拷贝和填充未初始化内存的算法

标准库还为 `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 描述了这些函数，它们都定义在头文件 `memory` 中。

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。

<code>uninitialized_copy(b, e, b2)</code>	从迭代器 <code>b</code> 和 <code>e</code> 指出的输入范围内拷贝元素到迭代器 <code>b2</code> 指定的未构造的原始内存中。 <code>b2</code> 指向的内存必须足够大，能容纳输入序列中元素的拷贝
<code>uninitialized_copy_n(b, n, b2)</code>	从迭代器 <code>b</code> 指向的元素开始，拷贝 <code>n</code> 个元素到 <code>b2</code> 开始的内存中
<code>uninitialized_fill(b, e, t)</code>	在迭代器 <code>b</code> 和 <code>e</code> 指定的原始内存范围内创建对象，对象的值均为 <code>t</code> 的拷贝
<code>uninitialized_fill_n(b, n, t)</code>	从迭代器 <code>b</code> 指向的内存地址开始创建 <code>n</code> 个对象。 <code>b</code> 必须指向足够大的未构造的原始内存，能够容纳给定数量的对象

作为一个例子，假定有一个 `int` 的 `vector`，希望将其内容拷贝到动态内存中。我们将分配一块比 `vector` 中元素所占用空间大一倍的动态内存，然后将原 `vector` 中的元素拷贝到前一半空间，对后一半空间用一个给定值进行填充：

```
// 分配比 vi 中元素所占用空间大一倍的动态内存
auto p = alloc.allocate(vi.size() * 2);
// 通过拷贝 vi 中的元素来构造从 p 开始的元素
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
// 将剩余元素初始化为 42
uninitialized_fill_n(q, vi.size(), 42);
```

484

类似拷贝算法（参见 10.2.2 节，第 341 页），`uninitialized_copy` 接受三个迭代器参数。前两个表示输入序列，第三个表示这些元素将要拷贝到的目的空间。传递给 `uninitialized_copy` 的目的位置迭代器必须指向未构造的内存。与 `copy` 不同，`uninitialized_copy` 在给定目的位置构造元素。

类似 `copy`，`uninitialized_copy` 返回（递增后的）目的位置迭代器。因此，一次 `uninitialized_copy` 调用会返回一个指针，指向最后一个构造的元素之后的位置。在本例中，我们将此指针保存在 `q` 中，然后将 `q` 传递给 `uninitialized_fill_n`。此函数类似 `fill_n`（参见 10.2.2 节，第 340 页），接受一个指向目的位置的指针、一个计数和一个值。它会在目的位置指针指向的内存中创建给定数目个对象，用给定值对它们进行初始化。

12.2.2 节练习

练习 12.26: 用 allocator 重写第 427 页中的程序。



12.3 使用标准库: 文本查询程序

我们将实现一个简单的文本查询程序, 作为标准库相关内容学习的总结。我们的程序允许用户在一个给定文件中查询单词。查询结果是单词在文件中出现的次数及其所在行的列表。如果一个单词在一行中出现多次, 此行只列出一次。行会按照升序输出——即, 第 7 行会在第 9 行之前显示, 依此类推。

例如, 我们可能读入一个包含本章内容 (指英文版中的文本) 的文件, 在其中寻找单词 element。输出结果的前几行应该是这样的:

```
element occurs 112 times
  (line 36) A set element contains only a key;
  (line 158) operator creates a new element
  (line 160) Regardless of whether the element
  (line 168) When we fetch an element from a map, we
  (line 214) If the element is not found, find returns
```

接下来还有大约 100 行, 都是单词 element 出现的位置。



12.3.1 文本查询程序设计

485

开始一个程序的设计的一种好方法是列出程序的操作。了解需要哪些操作会帮助我们分析出需要什么样的数据结构。从需求入手, 我们的文本查询程序需要完成如下任务:

- 当程序读取输入文件时, 它必须记住单词出现的每一行。因此, 程序需要逐行读取输入文件, 并将每一行分解为独立的单词
- 当程序生成输出时,
 - 它必须能提取每个单词所关联的行号
 - 行号必须按升序出现且无重复
 - 它必须能打印给定行号中的文本。

利用多种标准库设施, 我们可以很漂亮地实现这些要求:

- 我们将使用一个 `vector<string>` 来保存整个输入文件的一份拷贝。输入文件中的每行保存为 `vector` 中的一个元素。当需要打印一行时, 可以用行号作为下标来提取行文本。
- 我们使用一个 `istringstream` (参见 8.3 节, 第 287 页) 来将每行分解为单词。
- 我们使用一个 `set` 来保存每个单词在输入文本中出现的行号。这保证了每行只出现一次且行号按升序保存。
- 我们使用一个 `map` 来将每个单词与它出现的行号 `set` 关联起来。这样我们就可以方便地提取任意单词的 `set`。

我们的解决方案还使用了 `shared_ptr`, 原因稍后进行解释。

数据结构

虽然我们可以用 `vector`、`set` 和 `map` 来直接编写文本查询程序, 但如果定义一个更

为抽象的解决方案，会更为有效。我们将从定义一个保存输入文件的类开始，这会令文件查询更为容易。我们将这个类命名为 `TextQuery`，它包含一个 `vector` 和一个 `map`。`vector` 用来保存输入文件的文本，`map` 用来关联每个单词和它出现的行号的 `set`。这个类将会有个用来读取给定输入文件的构造函数和一个执行查询的操作。

查询操作要完成的任务非常简单：查找 `map` 成员，检查给定单词是否出现。设计这个函数的难点是确定应该返回什么内容。一旦找到了一个单词，我们需要知道它出现了多少次、它出现的行号以及每行的文本。

返回所有这些内容的最简单的方法是定义另一个类，可以命名为 `QueryResult`，来保存查询结果。这个类会有一个 `print` 函数，完成结果打印工作。

在类之间共享数据

486

我们的 `QueryResult` 类要表达查询的结果。这些结果包括与给定单词关联的行号的 `set` 和这些行对应的文本。这些数据都保存在 `TextQuery` 类型的对象中。

由于 `QueryResult` 所需要的数据都保存在一个 `TextQuery` 对象中，我们就必须确定如何访问它们。我们可以拷贝行号的 `set`，但这样做可能很耗时。而且，我们当然不希望拷贝 `vector`，因为这可能会引起整个文件的拷贝，而目标只不过是为了打印文件的一小部分而已（通常会是这样）。

通过返回指向 `TextQuery` 对象内部的迭代器（或指针），我们可以避免拷贝操作。但是，这种方法开启了一个陷阱：如果 `TextQuery` 对象在对应的 `QueryResult` 对象之前被销毁，会发生什么？在此情况下，`QueryResult` 就将引用一个不再存在的对象中的数据。

对于 `QueryResult` 对象和对应的 `TextQuery` 对象的生存期应该同步这一观察结果，其实已经暗示了问题的解决方案。考虑到这两个类概念上“共享”了数据，可以使用 `shared_ptr`（参见 12.1.1 节，第 400 页）来反映数据结构中的这种共享关系。

使用 `TextQuery` 类

当我们设计一个类时，在真正实现成员之前先编写程序使用这个类，是一种非常有用的方法。通过这种方法，可以看到类是否具有我们所需要的操作。例如，下面的程序使用了 `TextQuery` 和 `QueryResult` 类。这个函数接受一个指向要处理的文件的 `ifstream`，并与用户交互，打印给定单词的查询结果

```
void runQueries(ifstream &infile)
{
    // infile 是一个 ifstream，指向我们要处理的文件
    TextQuery tq(infile); // 保存文件并建立查询 map
    // 与用户交互：提示用户输入要查询的单词，完成查询并打印结果
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // 若遇到文件尾或用户输入了'q'时循环终止
        if (!(cin >> s) || s == "q") break;
        // 指向查询并打印结果
        print(cout, tq.query(s)) << endl;
    }
}
```

我们首先用给定的 `ifstream` 初始化一个名为 `tq` 的 `TextQuery` 对象。`TextQuery` 的构造函数读取输入文件，保存在 `vector` 中，并建立单词到所在行号的 `map`。

487 while (无限) 循环提示用户输入一个要查询的单词，并打印出查询结果，如此往复。循环条件检测字面常量 `true` (参见 2.1.3 节，第 37 页)，因此永远成功。循环的退出是通过 `if` 语句中的 `break` (参见 5.5.1 节，第 170 页) 实现的。此 `if` 语句检查输入是否成功。如果成功，它再检查用户是否输入了 `q`。输入失败或用户输入了 `q` 都会使循环终止。一旦用户输入了要查询的单词，我们要求 `tq` 查找这个单词，然后调用 `print` 打印搜索结果。

12.3.1 节练习

练习 12.27: `TextQuery` 和 `QueryResult` 类只使用了我们已经介绍过的语言和标准库特性。不要提前看后续章节内容，只用已经学到的知识对这两个类编写你自己的版本。

练习 12.28: 编写程序实现文本查询，不要定义类来管理数据。你的程序应该接受一个文件，并与用户交互来查询单词。使用 `vector`、`map` 和 `set` 容器来保存来自文件的数据并生成查询结果。

练习 12.29: 我们曾经用 `do while` 循环来编写管理用户交互的循环 (参见 5.4.4 节，第 169 页)。用 `do while` 重写本节程序，解释你倾向于哪个版本，为什么。



12.3.2 文本查询程序类的定义

我们以 `TextQuery` 类的定义开始。用户创建此类的对象时会提供一个 `istream`，用来读取输入文件。这个类还提供一个 `query` 操作，接受一个 `string`，返回一个 `QueryResult` 表示 `string` 出现的那些行。

设计类的数据成员时，需要考虑与 `QueryResult` 对象共享数据的需求。`QueryResult` 类需要共享保存输入文件的 `vector` 和保存单词关联的行号的 `set`。因此，这个类应该有两个数据成员：一个指向动态分配的 `vector` (保存输入文件) 的 `shared_ptr` 和一个 `string` 到 `shared_ptr<set>` 的 `map`。`map` 将文件中每个单词关联到一个动态分配的 `set` 上，而此 `set` 保存了该单词所出现的行号。

为了使代码更易读，我们还会定义一个类型成员 (参见 7.3.1 节，第 243 页) 来引用行号，即 `string` 的 `vector` 中的下标：

```
class QueryResult; // 为了定义函数 query 的返回类型，这个定义是必需的
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // 输入文件
    // 每个单词到它所在的行号的集合的映射
    std::map<std::string,
            std::shared_ptr<std::set<line_no>>> wm;
};
```

488 这个类定义最困难的部分是解开类名。与往常一样，对于可能置于头文件中的代码，在使用标准库名字时要加上 `std::` (参见 3.1 节，第 74 页)。在本例中，我们反复使用了 `std::`，

使得代码开始可能有些难读。例如，

```
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

如果写成下面的形式可能就更好理解一些

```
map<string, shared_ptr<set<line_no>>> wm;
```

TextQuery 构造函数

TextQuery 的构造函数接受一个 ifstream，逐行读取输入文件：

```
// 读取输入文件并建立单词到行号的映射
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {           // 对文件中每一行
        file->push_back(text);          // 保存此行文本
        int n = file->size() - 1;        // 当前行号
        istringstream line(text);         // 将行文本分解为单词
        string word;
        while (line >> word) {          // 对行中每个单词
            // 如果单词不在 wm 中，以之为下标在 wm 中添加一项
            auto &lines = wm[word];      // lines 是一个 shared_ptr
            if (!lines) // 在我们第一次遇到这个单词时，此指针为空
                lines.reset(new set<line_no>); // 分配一个新的 set
            lines->insert(n);           // 将此行号插入 set 中
        }
    }
}
```

构造函数的初始化器分配一个新的 vector 来保存输入文件中的文本。我们用 getline 逐行读取输入文件，并存入 vector 中。由于 file 是一个 shared_ptr，我们用-> 运算符解引用 file 来提取 file 指向的 vector 对象的 push_back 成员。

接下来我们用一个 istringstream (参见 8.3 节，第 287 页) 来处理刚刚读入的一行中的每个单词。内层 while 循环用 istringstream 的输入运算符来从当前行读取每个单词，存入 word 中。在 while 循环内，我们用 map 下标运算符提取与 word 相关联的 shared_ptr<set>，并将 lines 绑定到此指针。注意，lines 是一个引用，因此改变 lines 也会改变 wm 中的元素。

若 word 不在 map 中，下标运算符会将 word 添加到 wm 中 (参见 11.3.4 节，第 387 页)，与 word 关联的值进行值初始化。这意味着，如果下标运算符将 word 添加到 wm 中，lines 将是一个空指针。如果 lines 为空，我们分配一个新的 set，并调用 reset 更新 lines 引用的 shared_ptr，使其指向这个新分配的 set。

不管是否创建了一个新的 set，我们都调用 insert 将当前行号添加到 set 中。由于 lines 是一个引用，对 insert 的调用会将新元素添加到 wm 中的 set 中。如果一个 [489] 给定单词在同一行中出现多次，对 insert 的调用什么都不会做。

QueryResult 类

QueryResult 类有三个数据成员：一个 string，保存查询单词；一个 shared_ptr，指向保存输入文件的 vector；一个 shared_ptr，指向保存单词出现行号的 set。它唯

一的一个成员函数是一个构造函数，初始化这三个数据成员：

```
class QueryResult {
    friend std::ostream& print(std::ostream&, const QueryResult&);
public:
    QueryResult(std::string s,
                std::shared_ptr<std::set<line_no>> p,
                std::shared_ptr<std::vector<std::string>> f):
        sought(s), lines(p), file(f) { }
private:
    std::string sought; // 查询单词
    std::shared_ptr<std::set<line_no>> lines;           // 出现的行号
    std::shared_ptr<std::vector<std::string>> file;       // 输入文件
};
```

构造函数的唯一工作是将参数保存在对应的数据成员中，这是在其初始化器列表中完成的（参见 7.1.4 节，第 237 页）。

query 函数

query 函数接受一个 string 参数，即查询单词，query 用它来在 map 中定位对应的行号 set。如果找到了这个 string，query 函数构造一个 QueryResult，保存给定 string、TextQuery 的 file 成员以及从 wm 中提取的 set。

唯一的问题是：如果给定 string 未找到，我们应该返回什么？在这种情况下，没有可返回的 set。为了解决此问题，我们定义了一个局部 static 对象，它是一个指向空的行号 set 的 shared_ptr。当未找到给定单词时，我们返回此对象的一个拷贝：

```
QueryResult
TextQuery::query(const string &sought) const
{
    // 如果未找到 sought，我们将返回一个指向此 set 的指针
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // 使用 find 而不是下标运算符来查找单词，避免将单词添加到 wm 中！
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // 未找到
    else
        return QueryResult(sought, loc->second, file);
}
```

490 打印结果

print 函数在给定的流上打印出给定的 QueryResult 对象：

```
ostream &print(ostream &os, const QueryResult &qr)
{
    // 如果找到了单词，打印出现次数和所有出现的位置
    os << qr.sought << " occurs " << qr.lines->size() << " "
        << make_plural(qr.lines->size(), "time", "s") << endl;
    // 打印单词出现的每一行
    for (auto num : *qr.lines) // 对 set 中每个单词
        // 避免行号从 0 开始给用户带来的困惑
        os << "\t(line " << num + 1 << ") "
```

```
    << * (qr.file->begin() + num) << endl;
return os;
}
```

我们调用 qr.lines 指向的 set 的 size 成员来报告单词出现了多少次。由于 set 是一个 shared_ptr，必须解引用 lines。调用 make_plural（参见 6.3.2 节，第 201 页）来根据大小是否等于 1 打印 time 或 times。

在 for 循环中，我们遍历 lines 所指向的 set。for 循环体打印行号，并按人们习惯的方式调整计数值。set 中的数值就是 vector 中元素的下标，从 0 开始编号。但大多数用户认为第一行的行号应该是 1，因此我们对每个行号都加上 1，转换为人们更习惯的形式。

我们用行号从 file 指向的 vector 中提取一行文本。回忆一下，当给一个迭代器加上一个数时，会得到 vector 中相应偏移之后位置的元素（参见 3.4.2 节，第 99 页）。因此，file->begin() + num 即为 file 指向的 vector 中第 num 个位置的元素。

注意此函数能正确处理未找到单词的情况。在此情况下，set 为空。第一条输出语句会注意到单词出现了 0 次。由于 *res.lines 为空，for 循环一次也不会执行。

12.3.2 节练习

练习 12.30: 定义你自己版本的 TextQuery 和 QueryResult 类，并执行 12.3.1 节（第 431 页）中的 runQueries 函数。

练习 12.31: 如果用 vector 代替 set 保存行号，会有什么差别？哪种方法更好？为什么？

练习 12.32: 重写 TextQuery 和 QueryResult 类，用 StrBlob 代替 vector<string> 保存输入文件。

练习 12.33: 在第 15 章中我们将扩展查询系统，在 QueryResult 类中将会需要一些额外的成员。添加名为 begin 和 end 的成员，返回一个迭代器，指向一个给定查询返回的行号的 set 中的位置。再添加一个名为 get_file 的成员，返回一个 shared_ptr，指向 QueryResult 对象中的文件。

491

小结

在 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。现代 C++ 程序应尽可能使用智能指针。

术语表

allocator 标准库类，用来分配未构造的内存。

空悬指针（dangling pointer） 一个指针，指向曾经保存一个对象但现在已释放的内存。众所周知，空悬指针引起的程序错误非常难以调试。

delete 释放 `new` 分配的内存。`delete p` 释放对象，`delete []p` 释放 `p` 指向的数组。`p` 可以为空，或者指向 `new` 分配的内存。

释放器（deleter） 传递给智能指针的函数，用来代替 `delete` 释放指针绑定的对象。

析构函数（destructor） 特殊的成员函数，负责在对象离开作用域或被释放时完成清理工作。

动态分配的（dynamically allocated） 在自由空间中分配的对象。在自由空间中分配的对象直到被显式释放或程序结束才会销毁。

自由空间（free store） 程序可用的内存池，保存动态分配的对象。

堆（heap） 自由空间的同义词。

new 从自由空间分配内存。`new T` 分配并构造一个类型为 `T` 的对象，并返回一个指向该对象的指针。如果 `T` 是一个数组类型，`new` 返回一个指向数组首元素的指针。类似的，`new [n] T` 分配 `n` 个类型为 `T` 的对象，并返回指向数组首元素的指针。

默认情况下，分配的对象进行默认初始化。我们也可以提供可选的初始化器。

定位 new（placement new） 一种 `new` 表达式形式，接受一些额外的参数，在 `new` 关键字后面的括号中给出。例如，`new (nothrow) int` 告诉 `new` 不要抛出异常。

引用计数（reference count） 一个计数器，记录有多少用户共享一个对象。智能指针用它来判断什么时候释放所指向的对象是安全的。

shared_ptr 提供所有权共享的智能指针：对共享对象来说，当最后一个指向它的 `shared_ptr` 被销毁时会被释放。

智能指针（smart pointer） 标准库类型，行为类似指针，但可以检查什么时候使用指针是安全的。智能指针类型负责在恰当的时候释放内存。

unique_ptr 提供独享所有权的智能指针：当 `unique_ptr` 被销毁时，它指向的对象被释放。`unique_ptr` 不能直接拷贝或赋值。

weak_ptr 一种智能指针，指向由 `shared_ptr` 管理的对象。在确定是否应释放对象时，`shared_ptr` 并不把 `weak_ptr` 统计在内。