3

资源管理

Resource Management

所谓资源就是，一旦用了它，将来必须还给系统。如果不这样，糟糕的事情就会发生。C++ 程序中最常使用的资源就是动态分配内存（如果你分配内存却从来不曾归还它，会导致内存泄漏），但内存只是你必须管理的众多资源之一。其他常见的资源还包括文件描述器（file descriptors）、互斥锁（mutex locks）、图形界面中的字型和笔刷、数据库连接、以及网络 sockets。不论哪一种资源，重要的是，当你不再使用它时，必须将它还给系统。

尝试在任何运用情况下都确保以上所言，是件困难的事，但当你考虑到异常、函数内多重回传路径、程序维护员改动软件却没能充分理解随之而来的冲击，态势就很明显了：资源管理的特殊手段还不很充分够用。

本章一开始是一个直接而易懂且基于对象（object-based）的资源管理办法，建立在 C++ 对构造函数、析构函数、*copying* 函数的基础上。经验显示，经过训练后严守这些做法，可以几乎消除资源管理问题。然后本章的某些条款将专门用来对付内存管理。这些排列在后的专属条款弥补了先前一般化条款的不足，因为管理内存的那个对象必须知道如何适当而正确地工作。

条款 13：以对象管理资源

Use objects to manage resources.

假设我们使用一个用来塑模投资行为（例如股票、债券等等）的程序库，其中各式各样的投资类型继承自一个 root class `Investment`：

```
class Investment { ... };        // “投资类型” 继承体系中的 root class
```

进一步假设，这个程序库系通过一个工厂函数（factory function，见条款 7）供应我们某特定的 Investment 对象：

```
Investment* createInvestment();  //返回指针，指向 Investment 继承体系内
                                 //的动态分配对象。调用者有责任删除它。
                                 //这里为了简化，刻意不写参数。
```

一如以上注释所言，createInvestment 的调用端使用了函数返回的对象后，有责任删除之。现在考虑有个 f 函数履行了这个责任：

```
void f( )
{
    Investment* pInv = createInvestment( );        //调用 factory 函数
    ...
    delete pInv;                                   //释放 pInv 所指对象
}
```

这看起来妥当，但若干情况下 f 可能无法删除它得自 createInvestment 的投资对象——或许因为 "..." 区域内的一个过早的 return 语句。如果这样一个 return 被执行起来，控制流就绝不会触及 delete 语句。类似情况发生在对 createInvestment 的使用及 delete 动作位于某循环内，而该循环由于某个 continue 或 goto 语句过早退出。最后一种可能是 "..." 区域内的语句抛出异常，果真如此控制流将再次不会幸临 delete。无论 delete 如何被略过去，我们泄漏的不只是内含投资对象的那块内存，还包括那些投资对象所保存的任何资源。

当然啦，谨慎地编写程序可以防止这一类错误，但你必须想想，代码可能会在时间渐渐过去后被修改。一旦软件开始接受维护，可能会有某些人添加 return 语句或 continue 语句而未能全然领悟它对函数的资源管理策略造成的后果。更糟的是 f 的 "..." 区域有可能调用一个“过去从未抛出异常，却在被‘改善’之后开始那么做”的函数。因此单纯倚赖“f 总是会执行其 delete 语句”是行不通的。

为确保 createInvestment 返回的资源总是被释放，我们需要将资源放进对象内，当控制流离开 f，该对象的析构函数会自动释放那些资源。实际上这正是隐身于本条款背后的半边想法：把资源放进对象内，我们便可倚赖 C++ 的“析构函数自动调用机制”确保资源被释放。（稍后讨论另半边想法。）

许多资源被动态分配于 heap 内而后被用于单一区块或函数内。它们应该在控制流离开那个区块或函数时被释放。标准程序库提供的 auto_ptr 正是针对这种形势而设计的特制产品。auto_ptr 是个“类指针（pointer-like）对象”，也就是所谓“智能指针”，其析构函数自动对其所指对象调用 delete。下面示范如何使用 auto_ptr 以避免 f 函数潜在的资源泄漏可能性：

```
void f( )
{
   std::auto_ptr<Investment> pInv(createInvestment( ));
                                    //调用 factory 函数
   ...                              //一如以往地使用 pInv
}                                   //经由 auto_ptr 的析构函数自动删除 pInv
```

这个简单的例子示范“以对象管理资源”的两个关键想法：

- **获得资源后立刻放进管理对象**（managing object）**内**。以上代码中 createInvestment 返回的资源被当做其管理者 auto_ptr 的初值。实际上“以对象管理资源”的观念常被称为“资源取得时机便是初始化时机”（*Resource Acquisition Is Initialization;* RAII），因为我们几乎总是在获得一笔资源后于同一语句内以它初始化某个管理对象。有时候获得的资源被拿来赋值（而非初始化）某个管理对象，但不论哪一种做法，每一笔资源都在获得的同时立刻被放进管理对象中。
- **管理对象**（managing object）**运用析构函数确保资源被释放**。不论控制流如何离开区块，一旦对象被销毁（例如当对象离开作用域）其析构函数自然会被自动调用，于是资源被释放。如果资源释放动作可能导致抛出异常，事情变得有点棘手，但条款 8 已经能够解决这个问题，所以这里我们也就不多操心了。

由于 auto_ptr 被销毁时会自动删除它所指之物，所以一定要注意别让多个 auto_ptr 同时指向同一对象。如果真是那样，对象会被删除一次以上，而那会使你的程序搭上驶向“未定义行为”的快速列车上。为了预防这个问题，auto_ptrs 有一个不寻常的性质：若通过 *copy* 构造函数或 *copy assignment* 操作符复制它们，它们会变成 null，而复制所得的指针将取得资源的唯一拥有权！

```
std::auto_ptr<Investment>
  pInv1(createInvestment( ));                    //pInv1 指向
                                                 // createInvestment 返回物.
std::auto_ptr<Investment> pInv2(pInv1);          //现在 pInv2 指向对象,
                                                 //  pInv1 被设为 null.
pInv1 = pInv2;                                   //现在 pInv1 指向对象,
                                                 //  pInv2 被设为 null.
```

这一诡异的复制行为，复加上其底层条件：“受 auto_ptrs 管理的资源必须绝对没有一个以上的 auto_ptr 同时指向它”，意味 auto_ptrs 并非管理动态分配资源的神兵利器。举个例子，STL 容器要求其元素发挥“正常的”复制行为，因此这些容器容不得 auto_ptr。

auto_ptr 的替代方案是“引用计数型智慧指针”(*reference-counting smart pointer;* RCSP)。所谓 RCSP 也是个智能指针，持续追踪共有多少对象指向某笔资源，并在无人指向它时自动删除该资源。RCSPs 提供的行为类似垃圾回收（garbage collection），不同的是 RCSPs 无法打破环状引用（cycles of references，例如两个其实已经没被使用的对象彼此互指，因而好像还处在“被使用”状态）。

TR1 的 tr1::shared_ptr（见条款 54）就是个 RCSP，所以你可以这么写 f：

```
void f( )
{
   ...
   std::tr1::shared_ptr<Investment>
   pInv(createInvestment( ));                    //调用 factory 函数.
   ...                                           //使用 pInv 一如以往.
}                                  //经由 shared_ptr 析构函数自动删除 pInv
```

这段代码看起来几乎和使用 auto_ptr 的那个版本相同，但 shared_ptrs 的复制行为正常多了：

```
void f( )
{
   ...
   std::tr1::shared_ptr<Investment>
   pInv1(createInvestment( ));          //pInv1 指向
                                        //createInvestment 返回物.
   std::tr1::shared_ptr<Investment>
   pInv2(pInv1);                        //pInv1 和 pInv2 指向同一个对象.
   pInv1 = pInv2;                  //同上，无任何改变.
   ...
}                                       //pInv1 和 pInv2 被销毁,
                                        //它们所指的对象也就被自动销毁.
```

由于 tr1::shared_ptrs 的复制行为“一如预期”，它们可被用于 STL 容器以及其他“auto_ptr 之非正统复制行为并不适用”的语境上。

尽管如此，可别误会了，本条款并不专门针对 auto_ptr, tr1::shared_ptr 或任何其他智能指针，而只是强调“以对象管理资源”的重要性，auto_ptr 和 tr1::shared_ptr 只不过是实际例子。如果想知道 tr1:shared_ptr 的更多信息，请看条款 14, 18 和 54。

auto_ptr 和 tr1::shared_ptr 两者都在其析构函数内做 delete 而不是 delete[]动作（条款 16 对两者的不同有些描述）。那意味在动态分配而得的 array 身上使用 auto_ptr 或 tr1::shared_ptr 是个馊主意。尽管如此，可叹的是，那么做仍能通过编译：

```
std::auto_ptr<std::string>                          //馊主意！会用上错误的
  aps(new std::string[10]);                         //  delete 形式。
std::tr1::shared_ptr<int> spi(new int[1024]);       //相同问题。
```

你或许会惊讶地发现，并没有特别针对“C++ 动态分配数组”而设计的类似 auto_ptr 或 tr1::shared_ptr 那样的东西，甚至 TR1 中也没有。那是因为 vector 和 string 几乎总是可以取代动态分配而得的数组。如果你还是认为拥有针对数组而设计、类似 auto_ptr 和 tr1::shared_ptr 那样的 classes 较好，看看 Boost 吧（见条款 55）。在那儿你会很高兴地发现 boost::scoped_array 和 boost::shared_array classes，它们都提供你要的行为。

本条款也建议，如果你打算手工释放资源（例如使用 delete 而非使用一个资源管理类; resource-managing class），容易发生某些错误。罐装式的资源管理类如 auto_ptr 和 tr1::shared_ptr 往往比较能够轻松遵循本条款忠告，但有时候你所使用的资源是目前这些预制式 classes 无法妥善管理的。既然如此就需要精巧制作你自己的资源管理类。那并不是非常困难，但的确涉及若干你需要考虑的细节。那些考虑形成了条款 14 和条款 15 的标题。

作为最后批注，我必须指出，createInvestment返回的“未加工指针”（raw pointer）简直是对资源泄漏的一个死亡邀约，因为调用者极易在这个指针身上忘记调用delete。（即使他们使用auto_ptr或tr1::shared_ptr来执行delete，他们首先必须记得将createInvestment的返回值存储于智能指针对象内。）为与此问题搏斗，首先需要对createInvestment进行接口修改，那是条款18面对的事。

请记住

- 为防止资源泄漏，请使用RAII对象，它们在构造函数中获得资源并在析构函数中释放资源。
- 两个常被使用的RAII classes分别是tr1::shared_ptr和auto_ptr。前者通常是较佳选择，因为其*copy*行为比较直观。若选择auto_ptr，复制动作会使它（被复制物）指向null。

条款14：在资源管理类中小心 *coping* 行为

Think carefully about copying behavior in resource-managing classes.

条款13导入这样的观念：“资源取得时机便是初始化时机”（*Resource Acquisition Is Initialization;* RAII），并以此作为“资源管理类”的脊柱，也描述了auto_ptr和tr1::shared_ptr如何将这个观念表现在heap-based资源上。然而并非所有资源都是heap-based，对那种资源而言，像auto_ptr和tr1::shared_ptr这样的智能指针往往不适合作为资源掌管者（resource handlers）。既然如此，有可能偶而你会发现，你需要建立自己的资源管理类。

例如，假设我们使用C API函数处理类型为Mutex的互斥器对象（mutex objects），共有lock和unlock两函数可用：

```
void lock(Mutex* pm);        //锁定pm所指的互斥器.
void unlock(Mutex* pm);      //将互斥器解除锁定.
```

为确保绝不会忘记将一个被锁住的Mutex解锁，你可能会希望建立一个class用来管理机锁。这样的class的基本结构由RAII守则支配，也就是“资源在构造期间获得，在析构期间释放”：

```
class Lock {
public:
   explicit Lock(Mutex* pm)
     : mutexPtr(pm)
   { lock(mutexPtr); }                    //获得资源
```

```
    ~Lock() { unlock(mutexPtr); }        //释放资源
private:
    Mutex *mutexPtr;
};
```

客户对 Lock 的用法符合 RAII 方式：

```
Mutex m;        //定义你需要的互斥器
...
{               //建立一个区块用来定义 critical section.
Lock ml(&m); //锁定互斥器.
    ...         //执行 critical section 内的操作.
}               //在区块最末尾，自动解除互斥器锁定.
```

这很好，但如果 Lock 对象被复制，会发生什么事？

```
Lock ml1(&m);       //锁定 m
Lock ml2(ml1);      //将 ml1 复制到 ml2 身上。这会发生什么事?
```

这是某个一般化问题的特定例子。那个一般化问题是每一位 RAII class 作者一定需要面对的："当一个 RAII 对象被复制，会发生什么事？"大多数时候你会选择以下两种可能：

- **禁止复制**。许多时候允许 RAII 对象被复制并不合理。对一个像 Lock 这样的 class 这是有可能的，因为很少能够合理拥有"同步化基础器物"（synchronization primitives）的复件（副本）。如果复制动作对 RAII class 并不合理，你便应该禁止之。条款 6 告诉你怎么做：将 *copying* 操作声明为 private。对 Lock 而言看起来是这样：

  ```
  class Lock: private Uncopyable {        //禁止复制。见条款 6。
  public:
      ...                                 //如前
  };
  ```

- **对底层资源祭出"引用计数法"**（reference-count）。有时候我们希望保有资源，直到它的最后一个使用者（某对象）被销毁。这种情况下复制 RAII 对象时，应该将资源的"被引用数"递增。tr1::shared_ptr 便是如此。

  通常只要内含一个 tr1::shared_ptr 成员变量，RAII classes 便可实现出 reference-counting *copying* 行为。如果前述的 Lock 打算使用 reference counting，它可以改变 mutexPtr 的类型，将它从 Mutex* 改为 tr1::shared_ptr<Mutex>。然而很不幸 tr1::shared_ptr 的缺省行为是"当引用次数为 0 时删除其所指物"，那不是我们所要的行为。当我们用上一个 Mutex，我们想要做的释放动作是解

除锁定而非删除。

幸运的是 tr1::shared_ptr 允许指定所谓的“删除器”（deleter），那是一个函数或函数对象（function object），当引用次数为 0 时便被调用（此机能并不存在于 auto_ptr——它总是将其指针删除）。删除器对 tr1::shared_ptr 构造函数而言是可有可无的第二参数，所以代码看起来像这样：

```
class Lock {
public:
     explicit Lock(Mutex* pm)        //以某个 Mutex 初始化 shared_ptr
       : mutexPtr(pm, unlock)        //  并以 unlock 函数为删除器.
     {
          lock(mutexPtr.get());     //条款 15 谈到 "get"
     }
private:
     std::tr1::shared_ptr<Mutex> mutexPtr;   //使用 shared_ptr
};                                           //替换 raw pointer
```

请注意，本例的 Lock class 不再声明析构函数。因为没有必要。条款 5 说过，class 析构函数（无论是编译器生成的，或用户自定的）会自动调用其 non-static 成员变量（本例为 mutexPtr）的析构函数。而 mutexPtr 的析构函数会在互斥器的引用次数为 0 时自动调用 tr1::shared_ptr 的删除器（本例为 unlock）。（当你阅读这个 class 的原始码，或许会感谢其中有一条注释指出：你并没有忘记析构，你只是倚赖了编译器生成的缺省行为。）

- **复制底部资源。**有时候，只要你喜欢，可以针对一份资源拥有其任意数量的复件（副本）。而你需要“资源管理类”的唯一理由是，当你不再需要某个复件时确保它被释放。在此情况下复制资源管理对象，应该同时也复制其所包覆的资源。也就是说，复制资源管理对象时，进行的是“深度拷贝”。

  某些标准字符串类型是由“指向 heap 内存”之指针构成（那内存被用来存放字符串的组成字符）。这种字符串对象内含一个指针指向一块 heap 内存。当这样一个字符串对象被复制，不论指针或其所指内存都会被制作出一个复件。这样的字符串展现深度复制（deep copying）行为。

- **转移底部资源的拥有权。**某些罕见场合下你可能希望确保永远只有一个 RAII 对象指向一个未加工资源（raw resource），即使 RAII 对象被复制依然如此。此时资源的拥有权会从被复制物转移到目标物。一如条款 13 所述，这是 auto_ptr 奉行的复制意义。

Coping 函数（包括 *copy* 构造函数和 *copy assignment* 操作符）有可能被编译器自动创建出来，因此除非编译器所生版本做了你想要做的事（条款 5 提过其缺省行为），否则你得自己编写它们。某些情况下你或许也想支持这些函数的一般版本，这样的版本描述于条款 45。

请记住

- 复制 RAII 对象必须一并复制它所管理的资源，所以资源的 *copying* 行为决定 RAII 对象的 *copying* 行为。
- 普遍而常见的 RAII class *copying* 行为是：抑制 *copying*、施行引用计数法（reference counting）。不过其他行为也都可能被实现。

条款 15：在资源管理类中提供对原始资源的访问

Provide access to raw resources in resource-managing classes.

资源管理类（resource-managing classes）很棒。它们是你对抗资源泄漏的堡垒。排除此等泄漏是良好设计系统的根本性质。在一个完美世界中你将倚赖这样的 classes 来处理和资源之间的所有互动，而不是玷污双手直接处理原始资源（raw resources）。但这个世界并不完美。许多 APIs 直接指涉资源，所以除非你发誓（这其实是一种少有实际价值的举动）永不录用这样的 APIs，否则只得绕过资源管理对象（resource-managing objects）直接访问原始资源（raw resources）。

举个例子，条款 13 导入一个观念：使用智能指针如 `auto_ptr` 或 `tr1::shared_ptr` 保存 factory 函数如 `createInvestment` 的调用结果：

```
std::tr1::shared_ptr<Investment> pInv(createInvestment()); //见条款 13
```

假设你希望以某个函数处理 `Investment` 对象，像这样：

```
int daysHeld(const Investment* pi);                    //返回投资天数
```

你想要这么调用它：

```
int days = daysHeld(pInv);                    //错误!
```

却通不过编译，因为 daysHeld 需要的是 Investment* 指针，你传给它的却是个类型为 tr1::shared_ptr<Investment> 的对象。

这时候你需要一个函数可将 RAII class 对象（本例为 tr1::shared_ptr）转换为其所内含之原始资源（本例为底部之 Investment*）。有两个做法可以达成目标：显式转换和隐式转换。

tr1::shared_ptr 和 auto_ptr 都提供一个 get 成员函数，用来执行显式转换，也就是它会返回智能指针内部的原始指针（的复件）：

```
int days = daysHeld(pInv.get());              //很好，将 pInv 内的原始指针
                                              //传给 daysHeld
```

就像（几乎）所有智能指针一样，tr1::shared_ptr 和 auto_ptr 也重载了指针取值（pointer dereferencing）操作符（operator-> 和 operator*），它们允许隐式转换至底部原始指针：

```
class Investment {                            //investment 继承体系的根类
public:
   bool isTaxFree() const;
   ...
};

Investment* createInvestment();               //factory 函数

std::tr1::shared_ptr<Investment>              //令 tr1::shared_ptr
 pi1(createInvestment( ));                    //  管理一笔资源。

bool taxable1 = !(pi1->isTaxFree());          //经由 operator-> 访问资源。
...
std::auto_ptr<Investment> pi2(createInvestment());   //令 auto_ptr
                                                     //管理一笔资源。
bool taxable2 = !((*pi2).isTaxFree());        //经由 operator* 访问资源。
...
```

由于有时候还是必须取得 RAII 对象内的原始资源，某些 RAII class 设计者于是联想到“将油脂涂在滑轨上”，做法是提供一个隐式转换函数。考虑下面这个用于字体的 RAII class（对 C API 而言字体是一种原生数据结构）：

```
FontHandle getFont();                         //这是个C API。为求简化暂略参数。
```

```
void releaseFont(FontHandle fh);        //来自同一组 C API

class Font {                            //RAII class
public:
   explicit Font(FontHandle fh)         //获得资源;
    : f(fh)                             //采用 pass-by-value,
   { }                                  //  因为 C API 这样做。
   ~Font( ) { releaseFont(f ); }        //释放资源
private:
   FontHandle f;                        //原始（raw）字体资源
};
```

假设有大量与字体相关的 C API，它们处理的是 FontHandles，那么“将 Font 对象转换为 FontHandle”会是一种很频繁的需求。Font class 可为此提供一个显式转换函数，像 get 那样：

```
class Font {
public:
   ...
   FontHandle get() const { return f; }    //显式转换函数
   ...
};
```

不幸的是这使得客户每当想要使用 API 时就必须调用 get：

```
void changeFontSize(FontHandle f, int newSize);      //C API
Font f(getFont());
int newFontSize;
...
changeFontSize(f.get(),newFontSize);  //明白地将 Font 转换为 FontHandle
```

某些程序员可能会认为，如此这般地到处要求显式转换，足以使人们倒尽胃口，不再愿意使用这个 class，从而增加了泄漏字体的可能性，而 Font class 的主要设计目的就是为了防止资源（字体）泄漏。

另一个办法是令 Font 提供隐式转换函数，转型为 FontHandle：

```
class Font {
public:
   ...
   operator FontHandle() const          //隐式转换函数
   { return f; }
   ...
};
```

这使得客户调用 C API 时比较轻松且自然：

```
Font f(getFont());
int newFontSize;
...
changeFontSize(f, newFontSize);          //将 Font 隐式转换为 FontHandle
```

但是这个隐式转换会增加错误发生机会。例如客户可能会在需要 Font 时意外创建一个 FontHandle：

```
Font f1(getFont( ));
...
FontHandle f2 = f1;          //喔欧！原意是要拷贝一个 Font 对象，
                             //却反而将 f1 隐式转换为其底部的 FontHandle
                             //然后才复制它。
```

以上程序有个 FontHandle 由 Font 对象 f1 管理，但那个 FontHandle 也可通过直接使用 f2 取得。那几乎不会有好下场。例如当 f1 被销毁，字体被释放，而 f2 因此成为“虚吊的”（**dangle**）。

是否该提供一个显式转换函数（例如 get 成员函数）将 RAII class 转换为其底部资源，或是应该提供隐式转换，答案主要取决于 RAII class 被设计执行的特定工作，以及它被使用的情况。最佳设计很可能是坚持条款 18 的忠告：“让接口容易被正确使用，不易被误用”。通常显式转换函数如 get 是比较受欢迎的路子，因为它将“非故意之类型转换”的可能性最小化了。然而有时候，隐式类型转换所带来的“自然用法”也会引发天秤倾斜。

你的内心也可能认为，RAII class 内的那个返回原始资源的函数，与“封装”发生矛盾。那是真的，但一般而言它谈不上是什么设计灾难。RAII classes 并不是为了封装某物而存在；它们的存在是为了确保一个特殊行为——资源释放——会发生。如果一定要，当然也可以在这基本功能之上再加一层资源封装，但那并非必要。此外也有某些 RAII classes 结合十分松散的底层资源封装，藉以获得真正的封装实现。例如 tr1::shared_ptr 将它的所有引用计数机构封装了起来，但还是让外界很容易访问其所内含的原始指针。就像多数设计良好的 classes 一样，它隐藏了客户不需要看的部分，但备妥客户需要的所有东西。

请记住

- APIs 往往要求访问原始资源（raw resources），所以每一个 RAII class 应该提供一个“取得其所管理之资源”的办法。
- 对原始资源的访问可能经由显式转换或隐式转换。一般而言显式转换比较安全，但隐式转换对客户比较方便。

条款 16：成对使用 new 和 delete 时要采取相同形式

Use the same form in corresponding uses of new and delete.

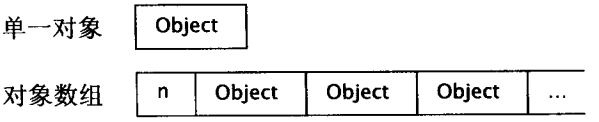
以下动作有什么错？

```
std::string* stringArray = new std::string[100];
...
delete stringArray;
```

每件事看起来都井然有序。使用了 new，也搭配了对应的 delete。但还是有某样东西完全错误：你的程序行为不明确（未有定义）。最低限度，stringArray 所含的 100 个 string 对象中的 99 个不太可能被适当删除，因为它们的析构函数很可能没被调用。

当你使用 new（也就是通过 new 动态生成一个对象），有两件事发生。第一，内存被分配出来（通过名为 operator new 的函数，见条款 49 和条款 51）。第二，针对此内存会有一个（或更多）构造函数被调用。当你使用 delete，也有两件事发生：针对此内存会有一个（或更多）析构函数被调用，然后内存才被释放（通过名为 operator delete 的函数，见条款 51）。delete 的最大问题在于：即将被删除的内存之内究竟存有多少对象？这个问题的答案决定了有多少个析构函数必须被调用起来。

实际上这个问题可以更简单些：即将被删除的那个指针，所指的是单一对象或对象数组？这是个必不可缺的问题，因为单一对象的内存布局一般而言不同于数组的内存布局。更明确地说，数组所用的内存通常还包括“数组大小”的记录，以便 delete 知道需要调用多少次析构函数。单一对象的内存则没有这笔记录。你可以把两种不同的内存布局想象如下，其中 n 是数组大小：



当然啦，这只是个例子。编译器不需非得这么实现不可，虽然很多编译器的确是这样做的。

当你对着一个指针使用 delete，唯一能够让 delete 知道内存中是否存在一个“数组大小记录”的办法就是：由你来告诉它。如果你使用 delete 时加上中括号（方括号），delete 便认定指针指向一个数组，否则它便认定指针指向单一对象：

```
std::string* stringPtr1 = new std::string;
std::string* stringPtr2 = new std::string[100];
...
delete stringPtr1;                  //删除一个对象
delete [ ] stringPtr2;              //删除一个由对象组成的数组
```

如果你对 stringPtr1 使用 "delete []" 形式，会发生什么事？结果未有定义，但不太可能让人愉快。假设内存布局如上，delete 会读取若干内存并将它解释为“数组大小”，然后开始多次调用析构函数，浑然不知它所处理的那块内存不但不是个数组，也或许并未持有它正忙着销毁的那种类型的对象。

如果你没有对 stringPtr2 使用 "delete []" 形式，又会发生什么事呢？唔，其结果亦未有定义，但你可以猜想可能导致太少的析构函数被调用。犹有进者，这对内置类型如 int 者亦未有定义（甚至有害），即使这类类型并没有析构函数。

游戏规则很简单：如果你调用 new 时使用[]，你必须在对应调用 delete 时也使用[]。如果你调用 new 时没有使用[]，那么也不该在对应调用 delete 时使用[]。

当你撰写的 class 含有一个指针指向动态分配内存，并提供多个构造函数时，上述规则尤其重要，因为这种情况下你必须小心地在所有构造函数中使用相同形式的 new 将指针成员初始化。如果没这样做，又如何知道该在析构函数中使用什么形式的 delete 呢？

这个规则对于喜欢使用 typedef 的人也很重要，因为它意味 typedef 的作者必须说清楚，当程序员以 new 创建该种 typedef 类型对象时，该以哪一种 delete 形式删除之。考虑下面这个 typedef：

```
typedef std::string AddressLines[4];        //每个人的地址有 4 行，
                                            //每行是一个 string
```

由于 AddressLines 是个数组，如果这样使用 new：

```
std::string* pal = new AddressLines;  //注意，"new AddressLines" 返回
                                      //一个 string*，就像
                                      //"new string[4]" 一样。
```

那就必须匹配“数组形式”的 delete：

```
delete pal;             //行为未有定义！
delete [ ] pal;         //很好。
```

为避免诸如此类的错误，最好尽量不要对数组形式做 typedefs 动作。这很容易达成，因为 C++ 标准程序库（条款 54）含有 string, vector 等 templates，可将数组的需求降至几乎为零。例如你可以将本例的 AddressLines 定义为“由 strings 组成的一个 vector”，也就是其类型为 vector<string>。

请记住

- 如果你在 new 表达式中使用 []，必须在相应的 delete 表达式中也使用 []。如果你在 new 表达式中不使用 []，一定不要在相应的 delete 表达式中使用 []。

条款 17：以独立语句将 newed 对象置入智能指针

Store newed objects in smart pointers in standalone statements.

假设我们有个函数用来揭示处理程序的优先权，另一个函数用来在某动态分配所得的 Widget 上进行某些带有优先权的处理：

```
int priority();
void processWidget(std::tr1::shared_ptr<Widget> pw, int priority);
```

由于谨记“以对象管理资源”（条款 13）的智慧铭言，processWidget 决定对其动态分配得来的 Widget 运用智能指针（这里采用 tr1::shared_ptr）。

现在考虑调用 processWidget：

```
processWidget(new Widget, priority());
```

等等，不要考虑这个调用形式。它不能通过编译。tr1::shared_ptr构造函数需要一个原始指针（raw pointer），但该构造函数是个explicit构造函数，无法进行隐式转换，将得自"newWidget"的原始指针转换为processWidget所要求的tr1::shared_ptr。如果写成这样就可以通过编译：

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority());
```

令人惊讶的是，虽然我们在此使用“对象管理式资源”（object-managing resources），上述调用却可能泄漏资源。稍后我再详加解释。

编译器产出一个processWidget调用码之前，必须首先核算即将被传递的各个实参。上述第二实参只是一个单纯的对 priority 函数的调用，但第一实参std::tr1:: shared_ptr<Widget>(new Widget)由两部分组成：

- 执行 "new Widget" 表达式
- 调用tr1::shared_ptr构造函数

于是在调用processWidget之前，编译器必须创建代码，做以下三件事：

- 调用priority
- 执行 "new Widget"
- 调用tr1::shared_ptr构造函数

C++ 编译器以什么样的次序完成这些事情呢？弹性很大。这和其他语言如 Java 和 C# 不同，那两种语言总是以特定次序完成函数参数的核算。可以确定的是 "new Widget" 一定执行于tr1::shared_ptr构造函数被调用之前，因为这个表达式的结果还要被传递作为tr1::shared_ptr构造函数的一个实参，但对priority的调用则可以排在第一或第二或第三执行。如果编译器选择以第二顺位执行它（说不定可因此生成更高效的代码，谁知道！），最终获得这样的操作序列：

1. 执行 "new Widget"
2. 调用 priority
3. 调用tr1::shared_ptr构造函数

现在请你想想，万一对 priority 的调用导致异常，会发生什么事？在此情况下 "new Widget" 返回的指针将会遗失，因为它尚未被置入tr1::shared_ptr内，后者是我们期盼用来防卫资源泄漏的武器。是的，在对processWidget的调用过程中可能引发资源泄漏，因为在“资源被创建（经由 "new Widget"）”和“资源被

转换为资源管理对象”两个时间点之间有可能发生异常干扰。

避免这类问题的办法很简单：使用分离语句，分别写出 (1) 创建 Widge，(2) 将它置入一个智能指针内，然后再把那个智能指针传给 processWidget：

```
std::tr1::shared_ptr<Widget> pw(new Widget);     //在单独语句内以
                                                 //  智能指针存储
                                                 //  newed 所得对象。
processWidget(pw, priority());          //这个调用动作绝不至于造成泄漏。
```

以上之所以行得通，因为编译器对于“跨越语句的各项操作”没有重新排列的自由（只有在语句内它才拥有那个自由度）。在上述修订后的代码内，"newWidget" 表达式以及“对 tr1::shared_ptr 构造函数的调用”这两个动作，和“对 priority 的调用”是分隔开来的，位于不同语句内，所以编译器不得在它们之间任意选择执行次序。

请记住

- 以独立语句将 newed 对象存储于（置入）智能指针内。如果不这样做，一旦异常被抛出，有可能导致难以察觉的资源泄漏。