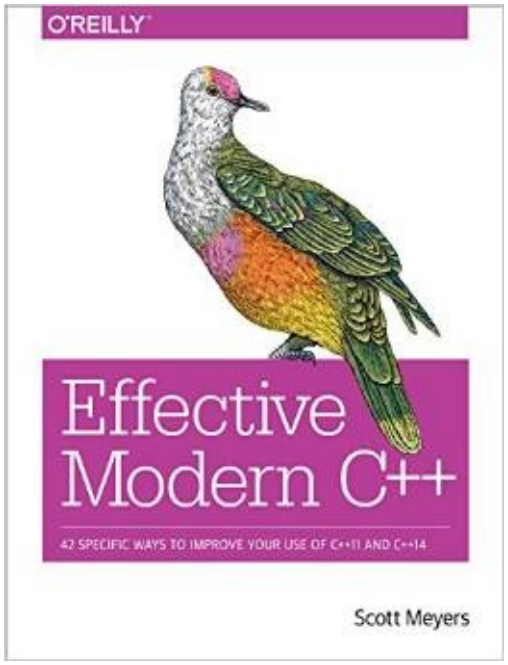
Table of Contents

Effective Modern C++

42 SPECIFIC WAYS TO IMPROVE YOUR USE OF C++11 AND C++14

Effective Modern C++ 中文翻译，欢迎大家提出翻译中的错误和用词不当的地方。



代码使用说明

使用gitbook作为静态编译输出，需要安装 `Node.js` ，然后从 `npm` 安装gitbook

```
npm install gitbook -g
```

然后git clone下来本书，然后输出静态网页，在浏览器上查看：

```
git clone git@github.com:XimingCheng/Effective-Modern-Cpp-Zh.git
cd Effective-Modern-Cpp-Zh
gitbook serve .
```

gitbook会默认在端口 `4000` 开启服务器，使用浏览器访问http://localhost:4000/就可以访问然后阅读本书的中文翻译。随后我会将本书编译生成的静态网页上传至github pages。

出版者的忠告

使用代码示例

这本书可以让你的工作得心应手。一般来说，如果在本书中提供了示例代码，你可以在你的程序和文档中。你不必为了代码的权限而联系我们，除非你要重新造一个伟大的轮子。举个例子，从书中代码编写一系列代码片段不需要授权，但是贩卖和分发O'Reilly的书籍代码的CD-ROM是需要授权的。引用本书和本书的例子代码来回答一些问题是不需要授权的，但是把很多的书中的代码整合进入你的产品文档里面是需要授权的。

我们会很高兴，但不是强迫引用归属。一个常见的写法包括标题，作者，出版社和ISBN号。举个例子："Effective Modern C++ by Scott Meyers (O'Reilly). Copyright 2015 Scott Meyers, 978-1-491-90399-5."

如果你觉得你对本书中的示例代码的使用超出了上述的权利要求范围，欢迎通过permissions@oreilly.com联系我们

Safari® Books Online

Safari Books Online是一个应需求的分发世界级领先的技术和商业作家的书籍和视频内容的电子库。

技术专家，软件开发者，web设计师和商务与创新型人士使用Safari Books Online作为他们的主力资源进行科研，解决问题，学习和认证训练。

Safari Books Online为企业用户，政府部门，教育用户和个人提供一些列的计划和打折

成员可以有权利访问上像O'Reilly Media，Prentice Hall Professional，Addison-Wesley Professional，Microsoft Press，Sams，Que，Peachpit Press，Focal Press，Cisco Press，John Wiley & Sons，Syngress，Morgan Kaufmann，IBM Redbooks，Packt，Adobe Press，FT Press，Apress，Manning，New Riders，McGraw-Hill，Jones & Bartlett，Course Technology和成百更多这样的可查询的数据库中的千本书籍，训练视频和重新出版的原稿。想获得更多关于Safari Books Online的信息，请访问我们的网站。

如何联系我们

关于本书的评论和问题可以联系出版社：

> O’Reilly Media, Inc. 1005 Gravenstein Highway North Sebastopol, CA 95472 800-998-9938 (in the United States or Canada) 707-829-0515 (international or local) 707-829-0104 (fax)

想问关于本书的技术性问题，可以发送邮件到 bookquestions@oreilly.com

想获得更多的关于本书的信息，课程，会议和新闻，参考我们的网站http://www.oreilly.com/

Facebook：http://facebook.com/oreilly

Twitter：http://twitter.com/oreillymedia

YouTube：http://www.youtube.com/oreillymedia

致谢

我在2009年开始着力于众所周知的C++0x（现在叫做C++11）。我给Usenet newsgroup `comp.std.c++` 投递了一些问题，我非常感谢社区的成员（特别是Daniel Krügler）的帖子。最近几年，当我有关于C++11和C++14的问题时候，我转战Stack Overflow，我同样受惠于这个社区带给我关于现代C++编程的一些理解。

2010年，我准备了一个关于C++0x的训练课程（最终以Overview of the New C++一书出版）。包括那些材料和知识都归功于和Stephan T. Lavavej, Bernhard Merkle, Stanley Friesen, Leor Zolman, Hendrik Schober, and Anthony Williams的审阅。没有他们的帮助，我可能永远都不会搞出这本Effective Modern C++。这个标题来源于我的一些读者在2014年2月18日发布的blog回帖"Help me name my book"，Andrei Alexandrescu（Modern C++ Design的作者）祝福书的标题不要抄袭他的术语。

我没法确定出这本书的所有原始信息，但是相关的资源有着直接的影响。条款4使用了一个由Stephan T. Lavavej和Matt P. Dziubinski建议在编译器中 `coax` 类型信息的未定义的模板，这也使得我对Boost.TypeIndex加以关注。在条款5中， `unsigned std::vector<int>::size_type` 的例子取自于2010年2月28日Andrey Karpov的文章"In what way can C++0x standard help you eliminate 64-bit errors"。 `std::pair<std::string, int>/std::pair<const std::string, int>` 的例子取自于Stephan T. Lavavej在Going Native 2012的talk"STL11: Magic && Secrets"。条款6有感于Herb Sutter在2013年8月12日的文章"GotW #94 Solution: AAA Style (Almost Always Auto)"。条款9的灵感来源于2012年5月27号Martinho Fernandes的博客"Handling dependent names"。条款12的检查reference qualifiers的重定义是基于2014年1月14号Casey在Stack Overflow的这个问题"What's a use case for overloading member functions on reference qualifiers"的回答。我的条款15的关于C++14对 `constexpr` 函数的扩展支持的信息来源于Rein Halbersma。条款16是基于Herb Sutter的C++ and Beyond 2012的演讲，"You don't know `const` and `mutable`"。条款18的建议工厂方法返回 `std::unique_ptr` 是基于Herb Sutter在2013年5月30号的文章，"GotW# 90 Solution: Factories"。在条款19中， `fastLoadWidget` 继承于Herb Sutter的Going Native 2013的演讲，"My Favorite C++ 10-Liner"。我在条款22中关于 `std::unique_ptr` 的不完备类型的论断来自于Herb Sutter在2011年11月27号的文章，"GotW #100: Compilation Firewalls"和Howard Hinnant在2011年5月22号在Stack Overflow的"Is std::unique_ptr required to know the full definition of T?"的回答。在条款25中的矩阵加法运算的例子是基于David Abrahams的作品。JoeArgonne在2012年12月8日的对2012年11月30日发布的帖子"Another alternative to lambda move capture"的评论是条款32的在C++11中基于 `std::bind` 的模拟init捕捉的来源。条款37的在 `std::thread` 的析构函数的隐式detach是取自于Hans-J. Boehm的2008年12月4日的paper，"N2802: A plea to reconsider detach-on-destruction for thread objects"。条款41最开始是来源于David Abrahams在2009年8月15日的博客帖子的讨论，"Want speed? Pass by value"。关于只移动types deserve special treatment归功于Matthew Fioravante，关于基于赋

值的copying stems来源于Howard Hinnant的评论。在条款42中，Stephan T. Lavavej和Howard Hinnant帮助我理解了emplacement和insertion函数的性能区别，Michael Winterberg带给我关于emplacement怎么导致内存泄露的问题，（Michael使用了Sean Parent的Going Native 2013的演讲，“C++ Seasoning”作为应用来源）。Michael并且指出了emplacement函数是怎么使用直接初始化，而insertion函数是使用拷贝初始化。

对一本技术书籍的校审是一件需要耗费时间精力并且绝对重要的任务，我非常荣幸有这么多的人愿意做这些事情。关于Effective Modern C++全部的或者部分的校审来自于Cassio Neri, Nate Kohl, Gerhard Kreuzer, Leor Zolman, Bart Vandewoestyne, Stephan T. Lavavej, Nevin ":-)" Liber, Rachel Cheng, Rob Stewart, Bob Steagall, Damien Watkins, Bradley E. Needham, Rainer Grimm, Fredrik Winkler, Jonathan Wakely, Herb Sutter, Andrei Alexandrescu, Eric Niebler, Thomas Becker, Roger Orr, Anthony Williams, Michael Winterberg, Benjamin Huchley, Tom Kirby-Green, Alexey A Nikitin, William Dealtry, Hubert Matthews, and Tomasz Kamiński。并且我收到了一些读者通过O'Reilly's Early Release EBooks和Safari Books Online's Rough Cuts和我的博客（The View from Aristeia）和电子邮件的反馈。我对这些人非常的感激。这本书有了他们的帮助而变的更好。我要特别感谢Stephan T. Lavavej和Rob Stewart，他们非常非常详细的标注让我怀疑他们在这本书上花的时间比我还多。对Leor Zolman也表示特殊的感谢，他不仅阅读了原稿，还对书中的所有示例代码做了double-check。

电子书的校审是Gerhard Kreuzer, Emyr Williams, and Bradley E. Needham做的.

我对书中的代码单行长度的显示限制在64个字母（这是和电子设备，设备方向切换和字体设置有关系的）是基于Michael Maher的提供。

Ashley Morgan Williams在Lake Oswego Pizzicato uniquely entertaining做好吃的晚餐（老子饿了，为了翻译没吃饭——译者注）。When it comes to man-sized Caesars, she's the go-to gal.（我实在不知道这句是啥意思——译者注）

More than 20 years after first living through my playing author, my wife, Nancy L.Urbano, once again tolerated many months of distracted conversations with a cocktail of resignation, exasperation, and timely splashes of understanding and support. During the same period, our dog, Darla, was largely content to doze away the hours I spent staring at computer screens, but she never let me forget that there's life beyond the keyboard. （这段描绘了非程序员的技术型作家的美好生活，和我们的反差太大我已无力翻译——译者注）

简介

如果你是一个像我一样有经验的C++程序猿，当初次体验C++11时，“啊，就是他，我明白了，这就是C++”。但是自从你学习了更多的内容，你会惊讶于他的变化。`auto`类型声明，基于区间的`for`循环，lambda表达式和右值引用改变了C++的样貌，还有新的并发API。除此之外，还包括一些合服语言习惯的改动。0和`typedef`都已经过时，`nullptr`和别名声明（alias declarations）强势登场。`enum`需要被作用域限制。现在更加建议使用在内部实现的智能指针。移动对象要比拷贝一个对象代价更小。

关于C++11我们要学习很多，还没有提C++14呢。

更重要的是，要想有效的利用好这些特性需要学习很多的东西。如果你拥有了关于“现代”C++特性，知识储备的基础，但是希望得到一个关于如何正确驾驭这些特性来创造运行正确，高效，可维护和可移植的软件的向导，搜寻的过程是非常具有挑战性的。这就是这本书的目的。他不是来介绍C++11和C++14的新特新的，而是来介绍如何使用他们的高效做法。

这本书的内容被封装成一系列叫做“条款”（item）的东东。想了解更多的关于类型推导的形式吗？或者想知道什么时候用（或者不用）`auto`声明吗？你对为什么`const`成员函数必须保证线程安全感兴趣吗？想知道怎么利用`std::unique_ptr`实现Pimpl Idiom，为什么不建议你在lambda表达式里面使用默认的捕捉模式，或者`std::atomic`和`volatile`有什么区别？答案都在这本书里面。更多的是平台独立，标准兼容的答案。这本书讲的是可移植的C++。

书中的条款是指导性建议，并不是法则，因为这些是有意外情况的。重要的不是每条条款带来的建议，而是这些建议背后的道理。一旦你理解了他们，你就可以在你的项目中扮演一个决策者的地位来判断他们是不是违背某个条款的指导性。这本书的目的不是告诉你什么要做，什么不要做，而是要你对C++11和C++14的基础之上进行深层理解。

术语和约定

为了让我们之间互相理解，在一开始我们预定C++的一些术语是很重要的。到目前为止有四份关于C++的官方版本，每一份依据对应ISO标准草案定制的年份来命名：C++98，C++03，C++11和C++14。C++98和C++03只是在技术细节上略有区别，在这本书里面我把它们都称之为C++98。当我提到C++11的时候，指的是C++11和C++14，因为C++14就是一个C++11的超集。当我写到C++14的时候，是特指C++14。当我简单的提到C++的时候，应该指的是所有的语言版本。

我的表述	我所指的语言版本
C++	所有的版本
C++98	C++98和C++03
C++11	C++11和C++14
C++14	C++14

一般来说，我可能绝大时候说C++对运行效率比较重视（对所有的版本都是对的），但是C++98缺乏对并发的支持（这个对于C++03和C++98是对的），但是C++11支持lambda表达式（对C++11和C++14是对的），C++14提供了通用的函数返回值类型推导（对C++14是对的）。

C++11的最普遍的特性是移动语义（move semantics），移动语义的基石是从那些左值中区分出右值。这是因为右值标志着对象是可以在移动操作中使用的而左值通常不是。在概念上来说（在实际中不一定），右值代表着你可以引用的临时对象，不管是通过变量名还是通过一个指针或者左值引用。

一个有用的，有启发意义的判断一个表达式是左值的方法是取它的地址。如果可以取地址，它基本上就是一个左值。如果不行，通常来说是一个右值。这个启发式的特性可以很好的帮助我们记住一个表达式的类型，不管他是一个左值还是一个右值。也就是说，给定一个类型 `T`，你可以得到类型 `T` 的左值同时也可以得到它的右值。当处理一个有右值引用的参数时需要铭记于心，因为参数本身是个左值：

```
class Widget {
public:
    Widget(Widget&& rhs);       // rhs是一个左值，尽管他
    …                           // 有一个右值引用类型
};
```

这里，在 `Widget` 的移动构造函数里面完全可以取得 `rds` 的地址，所以 `rds` 是一个左值尽管他的类型是个右值引用。（因为类似的原因，所有的参数都是左值。）

这段代码片段阐述了我一般要遵守的几条原则：

- 类名是 `Widget`。我通常会使用 `Widget` 来代指一个任意的用户自定义类型。我使用 `Widget` 是不会声明他的，除非我要展示类的特殊细节。
- 我使用的参数名字叫做 `rhs`（“right-hand side”）。他是我在移动操作（移动构造函数和移动赋值运算符）中和拷贝操作（拷贝构造函数和复制赋值运算符）喜欢使用的名字。我还把他用在二元运算符的右边的参数：

  ```
  Matrix operator+(const Matrix& lhs, const Matrix& rhs);
  ```

不要惊讶，我希望 `lhs` 代表“left-hand side”。

- 我在代码和注释中使用这种格式向你表示你要注意这些东西。在 `Widget` 的移动构造函数中，我高亮了 `rhs` 和部分注释来表明 `rhs` 是一个左值。（很抱歉，译者使用的Markdown语法暂时无法控制代码里面的高亮——译者注。）高亮代码从根本上说不好也不坏。他只是一段你需要加以注意的特殊的代码。

- 我使用“...”来表示“在此处有其他的代码”。这种比较窄的省略号和用在C++11源代码里面的的变长模板的宽省略号（“...”）是不一样的。这听起来比较困惑。举个例子：

```
template<typename... Ts>               // 这里是C++
void processVals(const Ts&... params)  // 源代码里面的
{                                      // 省略号

  …                                   // 此处意味着
}                                      // “有些代码着这里省略了”
```

`processVals` 展示了我在模板中使用 `typename` 关键字，但是这只是一个个人习惯；关键字 `class` 也可以工作的正常（这里是不严谨的，nested dependent type name使用的时候， `typename` 是不能替换成 `class` 的——译者注）。当我要使用C++标准展示代码，我会使用 `class` 来做参数类型类型声明，因为标准就是这样做的。

当一个对象使用另外一个类型相同的对象来初始化的时候，新的对象称作一份初始化对象的拷贝，甚至这个拷贝是基于移动构造函数实现的也叫做对象的拷贝。遗憾的是，在C++中没有一个术语是用来区分拷贝构造个移动构造的拷贝。

```
void someFunc(Widget w);        // someFunc的参数w是以值传送

Widget wid;                     // wid是个Widget的对象

someFunc(wid);                  // 在这个someFunc调用里面，w是通过
                                // 拷贝构造函数生成wid的一个拷贝


someFunc(std::move(wid));       // 在这个someFunc调用里面，w是通过
                                // 移动构造函数生成wid的一个拷贝
```

在一个函数调用里面，在函数的调用方的表达式是函数的实参。这些表达式被用来初始化函数的形参。在上面的代码中的第一次调用 `someFunc` ，实参是 `wid` 。在第二次调用的地方，实参是 `std::move(wid)` 。两次调用的形参都是 `w` 。实参和形参的区别是很重要的，因为形参只能是左值，但是给他们初始化的实参即有可能是右值也有可能是左值。这和完美转发的过程是密切相关的，在完美转发中一个传递给一个函数的实参再传递给第二个函数，以此来保证原始的参数的右值特性或者左值特性被保留。（完美转发的细节在条款30中）。

良好设计的函数是异常安全的，也就意味着他们至少接受基本的异常保证（弱保证）。这样的函数确保调用者触发异常，程序任然保持正常（没有数据结构被损坏）没有资源泄露。函数保证强壮的异常安全（强保证）会确保程序发生异常的时候，程序的运行状态和之前调用这个函数的状态是一样的。

当我提到函数对象（仿函数也属于其中一种——译者注）的时候，我通常意味着这个类型支持 `operator()` 操作。也就是说，这个对象的行为像一个函数。有时候我会在一些更加通用的地方来使用这种说法（“ `functionName(arguments)` ”）。更加广义的定义不仅仅包含那些支持 `operator()` 的对象，也包括函数和C风格的函数指针。（狭义的定义来自于C++98，广义的定义来自于C++11）。添加成员函数指针被称之为可调用对象（callable objects）。通常你可以忽略他们的区别，仅仅认识到在C++中函数对象和可调用对象可以被用在一些函数调用的语法结构里面。

通过lambda表达式创造的函数对象通常称之为闭包（closure）。通常很少区分lambda表达式和它产生的闭包，我通常用lambdas来代指它们。类似的，我很少区分函数模板（生成函数的模板）和模板函数（利用函数模板生成的函数）。对于类模板和模板类也是如此。

在C++许多东西可以声明和定义。声明把类型和名字带入我们的视野但是细节啥都不给，例如是在哪儿放置的存储空间，问题是怎么实现的之类的：

```
extern int x;                   // 对象声明

class Widgets;                  // class声明

bool func(const Widget& w);     // 函数声明

enum class Color;               // 被作用域包裹的enum声明（参考条款10）
```

定义提供存储地址或者实现的细节：

```
int x;                          // 对象定义

class Widget {
  …                             // class定义
};

bool func(const Widget& w)
{ return w.size() < 10; }       // 函数定义

enum class Color
{ Yellow, Red, Blue };          // 被作用域包裹的enum定义
```

一个定义当然是需要对应一个声明，除非定义对某个东西非常重要，我通常指的是声明。

我指一个函数的签名是由函数的参数和返回值确定的。函数和参数的名字并不是函数签名的一部分。在上述代码中，`func`的签名是`bool(const Widget&)`。函数声明的组成部分除了他的参数和返回值（比如如果有`noexcept`或者`constexpr`）都被排除在外。（`noexcept`和`constexpr`在条款14和条款15中被讨论）。正式的“签名”的定义和我的略有出入。但对于这本书来说，我的定义会非常有用。（正式的定义会排除返回值类型）。

新的C++标准通常兼容于老的代码，但是有的时候标准化委员会会废弃一些特性。这些特性很有可能在未来的标准化进程中被移除。编译器可能对这些即将废弃的特性没有任何警告，但是你最好要避免使用它们。不仅仅是因为他们会给将来的代码带来头痛，而且他们通常是有好的实现来代替它们。举个例子，`std::auto_ptr`被C++11所废弃，因为有更好的相同功能的`std::unique_ptr`，而且能做的更好。（`std::auto_ptr`本来是设计用来防止内存泄露的智能指针，但是为了使用它你必须要注意一堆坑，一般旧的C++书籍也会说明不建议使用——译者注）。

有些时候标准说某个操作会导致未定义行为，这意味着运行时的行为无法预测，不用说，你是需要避开这种不确定性的。一个未确定性的例子是使用方括号（“[]”）去索引超出`std::vector`的长度，从一个未初始化的迭代器取值，或者是有趣的数据竞争（两个或者更多的线程，至少有一个是生产者，同时访问同一块内存区域）。

我把直接从new返回的原始指针叫做内建指针。一个原始指针的反义词就是智能指针。智能指针通常重载了指针取值运算符（`operator->`和`operator*`），在条款20里面会解释`std::weak_ptr`是个特殊情况。

在源码注释里面，我通常把“构造函数”简称为ctor，“析构函数”简称为dtor。

报告Bug和建议优化

我尽我的努力去让这本书能够带来清楚，准确，有用的信息，但是总是可以再度改善完美的。如果你发现书中的任何错误（技术的，解释的，语法的，印刷的，等等）或者你有一些关于让这本书更好的建议，可以给我发邮件 emc++@aristeia.com。关于修订Effective Modern C++可以交付于书新版，但是我不能确定出我不知道的问题。

查看这本书已经发现的问题，审核本书的勘误。http://www.aristeia.com/BookErrata/emc++-errata.html。

第一章 类型推导

C++98只有一种类型推导规则：函数模板。C++11修改了一点规则样本，并且添加额外的两条规则，一条是`auto`，另一个是`decltype`。C++14继续扩展了`auto`和`decltype`的使用情况。随着类型推导的广泛使用，会使得你从一些明显的或者是冗余的类型拼写中解放出来。它使得C++编写的软件更加具有适用性，因为改变代码中的一处地方的类型，编译器会在代码的其他地方自动的推导出类型定义。但是这使得代码扫描过程更加困难，因为类型推导对编译器来说并不是你想的那么简单。

不去理解类型推导是如何操作的，高效的使用现代C++进行编程是不可能的。在类型推导过程中有太多的上下文判断，在大多数情况，`auto`出现在调用函数模板时，在`decltype`表达式里面，和在C++14中，神秘的`decltype(auto)`构造。

本章提供每个C++开发者必需的关于类型推导的信息。解释了模板类型推导是怎么工作的，`auto`如何左右类型，`decltype`是怎样运行的。甚至解释了如何强制编译器限定显示类型推导的结果，这样会帮助你明确编译器推导的类型是不是你所需要的。

条款1：理解模板类型推导

Understand template type deduction.

当一个复杂系统的用户忽略这个系统是如何工作的，那就再好不过了，因为“如何”扯了一堆系统的设计细节。从这个方面来度量，C++的模板类型推导是个巨大的成功。成百上万的程序猿给模板函数传递完全类型匹配的参数，尽管有很多的程序猿会更加苛刻的给于这个函数推导的类型的严格描述。

如果上面的描述包括你，那我有好消息也有坏消息。好消息就是模板的类型推导是现代C++的最引人注目的特性：`auto`。如果你喜欢C++98模板的类型推导，那么你会喜欢上C++11的`auto`对应的模板类型推导。坏消息就是模板类型推导的法则是受限于`auto`的上下文的，有时候看起来应用到模板上不是那么直观。因为这个原因，真正的理解模板`auto`的类型推导是很重要的。这条条款囊括了你的所需。

如果你想大致看一段伪码，一段函数模板看起来会是这样：

```
template<typename T>
void f(ParamType param);
```

调用会是这样：

```
f(expr);                    // 用一些表达式来调用f
```

在编译的时候，编译器通过`expr`来进行推导出两个类型：一个是`T`的，另一个是`ParamType`。通常来说这些类型是不同的，因为`ParamType`通常包含一些类型的装饰，比如`const`或引用特性。举个例子，模板通常采用如下声明：

```
template<typename T>
void f(const T& param);        // ParamType 是 const T&
```

如果有这样的调用：

```
int x = 0;

f(x);                          // 使用int调用f
```

`T`被推导成`int`，`ParamType`被推导成`const int&`。

一般会很自然的期望 `T` 的类型和传递给他的参数的类型一致，也就是说 `T` 的类型就是 `expr` 的类型。在上面的例子中，`x` 是一个 `int`，`T` 也就被推导成 `int`。但是并不是所有的情况都是如此。`T` 的类型不仅和 `expr` 的类型独立，而且还和 `ParamType` 的形式独立。下面是三个例子：

- `ParamType` 是一个指针或者是一个引用类型，但并不是一个通用的引用类型（通用的引用类型的内容在条款24。此时，你要知道例外情况会出现的，他们的类型并不和左值应用或者右值引用）。
- `ParamType` 是一个通用的引用
- `ParamType` 既不是指针也不是引用

这样的话，我们就有了三种类型需要检查的类型推导场景。每一种都是基于我们队模板的通用的调用封装：

```
template<typename T>
void f(ParamType param);

f(expr);                    // 从expr推导出T和ParamType的类型
```

第一种情况：`ParamType` 是个非通用的引用或者是一个指针

最简单的情况是当 `ParamType` 是一个引用类型或者是一个指针，但并非是通用的引用。在这种情况下，类型推导的过程如下：

1. 如果 `expr` 的类型是个引用，忽略引用的部分。
2. 然后利用 `expr` 的类型和 `ParamType` 对比去判断 `T` 的类型。

举一个例子，如果这个是我们的模板，

```
template<typename T>
void f(T& param);           // param是一个引用类型
```

我们有这样的代码变量声明：

```
int x = 27;                 // x是一个int
const int cx = x;           // cx是一个const int
const int& rx = x;          // rx是const int的引用
```

`param` 和 `T` 在不同的调用下面的类型推导如下：

```
f(x);                       // T是int，param的类型时int&

f(cx);                      // T是const int，
                            // param的类型是const int&
f(rx);                      // T是const int
                            // param的类型时const int&
```

在第二和第三部分的调用，注意 `cx` 和 `rx` 由于被指定为 `const` 类型变量，`T` 被推导成 `const int`，这也就导致了参数的类型被推导为 `const int&`。这对调用者非常重要。当传递一个 `const` 对象给一个引用参数，他们期望对象会保留常量特性，也就是说，参数变成了 `const` 的引用。这也就是为什么给一个以 `T&` 为参数的模板传递一个 `const` 对象是安全的：对象的 `const` 特性是 `T` 类型推导的一部分。

在第三个例子中，注意尽管 `rx` 的类型是一个引用，`T` 仍然被推导成了一个非引用的。这是因为 `rx` 的引用特性会被类型推导所忽略。

这些例子展示了左值引用参数的处理方式，但是类型推导在右值引用上也是如此。当然，右值参数只可能传递给右值引用参数，但是这个限制和类型推导没有关系。

如果我们把 `f` 的参数类型从 `T&` 变成 `const T&`，情况就会发生变化，但是并不会令人惊讶。由于 `param` 的声明是 `const` 引用的，`cx` 和 `rx` 的 `const` 特性会被保留，这样的话 `T` 的 `const` 特性就没有必要了。

```
template<typename T>
void f(const T& param);     // param现在是const的引用

int x = 27;                 // 和之前一样
const int cx = x;           // 和之前一样
const int& rx = x;          // 和之前一样

f(x);                       // T是int，param的类型是const int&

f(cx);                      // T是int，param的类型是const int&

f(rx);                      // T是int，param的类型是const int&
```

和之前一样，`rx` 的引用特性在类型推导的过程中会被忽略。

如果 `param` 是一个指针（或者指向 `const` 的指针）而不是引用，情况也是类似：

```
template<typename T>
void f(T* param);           // param是一个指针

int x = 27;                 // 和之前一样
const int *px = &x;         // px是一个指向const int x的指针

f(&x);                      // T是int，param的类型是int*

f(px);                      // T是const int
                            // param的类型时const int*
```

到目前为止，你或许瞌睡了，因为C++在引用和指针上的类型推导法则是如此的自然，我写出来读者看显得很没意思。所有的事情都这么明显！这就是读者所期望的的类型推导系统吧。

第二种情况：`ParamType`是个通用的引用（Universal Reference）

对于通用的引用参数，情况就变得不是那么明显了。这些参数被声明成右值引用（也就是函数模板使用一个类型参数`T`，一个通用的引用参数的申明类型是`T&&`），但是当传递进去右值参数情况变得不一样。完整的讨论请参考条款24，这里是先行版本。

- 如果`expr`是一个左值，`T`和`ParamType`都会被推导成左值引用。这有些不同寻常。第一，这是模板类型`T`被推导成一个引用的唯一情况。第二，尽管`ParamType`利用右值引用的语法来进行推导，但是他最终推导出来的类型是左值引用。
- 如果`expr`是一个右值，那么就执行“普通”的法则（第一种情况）

举个例子：

```
template<typename T>
void f(T&& param);            // param现在是一个通用的引用

int x = 27;                 // 和之前一样
const int cx = x;           // 和之前一样
const int& rx = x;          // 和之前一样

f(x);                        // x是左值，所以T是int&
                            // param的类型也是int&

f(cx);                       // cx是左值，所以T是const int&
                            // param的类型也是const int&

f(rx);                       // rx是左值，所以T是const int&
                            // param的类型也是const int&

f(27);                       // 27是右值，所以T是int
                            // 所以param的类型是int&&
```

条款23解释了这个例子推导的原因。关键的地方在于通用引用的类型推导法则和左值引用或者右值引用的法则大不相同。特殊的情况下，当使用了通用的引用，左值参数和右值参数的类型推导大不相同。这在非通用的类型推到上面绝对不会发生。

第三种情况：`ParamType`既不是指针也不是引用

当`ParamType`既不是指针也不是引用，我们把它处理成pass-by-value：

```
template<typename T>
void f(T param);            // param现在是pass-by-value
```

这就意味着`param`就是完全传给他的参数的一份拷贝——一个完全新的对象。基于这个事实可以从`expr`给出推导的法则：

1. 和之前一样，如果`expr`的类型是个引用，将会忽略引用的部分。
2. 如果在忽略`expr`的引用特性，`expr`是个`const`的，也要忽略掉`const`。如果是`volatile`，照样也要忽略掉（`volatile`对象并不常见。它们常常被用在实现设备驱动上面。查看更多的细节，请参考条款40。）

这样的话：

```
int x = 27;                 // 和之前一样
const int cx = x;           // 和之前一样
const int& rx = x;          // 和之前一样

f(x);                       // T和param的类型都是int

f(cx);                      // T和param的类型也都是int

f(rx);                      // T和param的类型还都是int
```

注意尽管`cx`和`rx`都是`const`类型，`param`却不是`const`的。这是有道理的。`param`是一个和`cx`和`rx`独立的对象——一个`cx`和`rx`的拷贝。`cx`和`rx`不能被修改和`param`能不能被修改是没有关系的。这就是为什么`expr`的常量特性（或者是易变性）（在很多的C++书籍上面`const`特性和`volatile`特性被称之为CV特性——译者注）在推导`param`的类型的时候被忽略掉了：`expr`不能被修改并不意味着它的一份拷贝不能被修改。

认识到`const`（和`volatile`）在按值传递参数的时候会被忽略掉。正如我们所见，引用的`const`或者是指针指向`const`，`expr`的`const`特性在类型推导的过程中会被保留。但是考虑到`expr`是一个`const`的指针指向一个`const`对象，而且`expr`被通过按值传递传递给`param`：

```
template<typename T>
void f(T param);              // param仍然是按值传递的（pass by value）

const char* const ptr =       // ptr是一个const指针，指向一个const对象
  "Fun with pointers";

f(ptr);                       // 给参数传递的是一个const char * const类型
```

这里，位于星号右边的 `const` 是表明指针是常量 `const` 的：`ptr` 不能被修改指向另外一个不同的地址，并且也不能置成 `null`。（星号左边的 `const` 表明 `ptr` 指向的——字符串——是 `const` 的，也就是说字符串不能被修改。）当这个 `ptr` 传递给 `f`，组成这个指针的内存bit被拷贝给 `param`。这样的话，指针自己（`ptr`）本身是被按值传递的。按照按值传递的类型推导法则，`ptr` 的 `const` 特性会被忽略，这样 `param` 的推导出来的类型就是 `const char*`，也就是一个可以被修改的指针，指向一个 `const` 的字符串。`ptr` 指向的东西的 `const` 特性被加以保留，但是 `ptr` 自己本身的 `const` 特性会被忽略，因为它要被重新复制一份而创建了一个新的指针 `param`。

数组参数

这主要出现在mainstream的模板类型推导里面，但是有一种情况需要特别加以注意。就是数组类型和指针类型是不一样的，尽管它们通常看起来是可以替换的。一个最基本的幻觉就是在很多的情况下，一个数组会被退化成一个指向其第一个元素的指针。这个退化的代码常常如此：

```
const char name[] = "J. P. Briggs";     // name的类型是const char[13]

const char * ptrToName = name;          // 数组被退化成指针
```

在这里，`const char*` 指针 `ptrToName` 使用 `name` 初始化，实际的 `name` 的类型是 `const char[13]`。这些类型（`const char*` 和 `const char[13]`）是不一样的，但是因为数组到指针的退化规则，代码会被正常编译。

但是如果一个数组传递给一个安置传递的模板参数里面情况会如何？会发生什么呢？

```
template<typename T>
void f(T param);              // 模板拥有一个按值传递的参数

f(name);                      // T和param的类型会被推到成什么呢？
```

我们从一个没有模板参数的函数开始。是的，是的，语法是合法的，

```
void myFunc(int param[]);       // 和上面的函数相同
```

但是以数组声明，但是还是把它当成一个指针声明，也就是说 `myFunc` 可以和下面的声明等价：

```
void myFunc(int* param);        // 和上面的函数是一样的
```

这样的数组和指针等价的声明经常会在以C语言为基础的C++里面出现，这也就导致了数组和指针是等价的错觉。

因为数组参数声明会被当做指针参数，传递给模板函数的按值传递的数组参数会被退化成指针类型。这就意味着在模板 `f` 的调用中，模板参数 `T` 被推导成 `const char*` ：

```
f(name);                    // name是个数组，但是T被推导成const char*
```

但是来一个特例。尽管函数不能被真正的定义成参数为数组，但是可以声明参数是数组的引用！所以如果我们修改模板 `f` 的参数成引用，

```
template<typename T>
void f(T& param);           // 引用参数的模板
```

然后传一个数组给他

```
f(name);                    // 传递数组给f
```

`T` 最后推导出来的实际的类型就是数组！类型推导包括了数组的长度，所以在这个例子里面，`T` 被推导成了 `const char [13]` ，函数 `f` 的参数（数组的引用）被推导成了 `const char (&)[13]` 。是的，语法看起来怪怪的，但是理解了这些可以升华你的精神（原文knowing it will score you mondo points with those few souls who care涉及到了几个宗教词汇——译者注）。

有趣的是，声明数组的引用可以使的创造出一个推导出一个数组包含的元素长度的模板：

```
// 在编译的时候返回数组的长度（数组参数没有名字，
// 因为只关心数组包含的元素的个数）
template<typename T, std::size_t N>
constexpr std::size_t arraySize(T (&)[N]) noexcept
{
    return N;                   // constexpr和noexcept在随后的条款中介绍
}
```

（`constexpr`是一种比`const`更加严格的常量定义，`noexcept`是说明函数永远都不会抛出异常——译者注）

正如条款15所述，定义为`constexpr`说明函数可以在编译的时候得到其返回值。这就使得创建一个和一个数组长度相同的一个数组，其长度可以从括号初始化：

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };      // keyVals有七个元素

int mappedVals[arraySize(keyVals)];              // mappedVals长度也是七
```

当然，作为一个现代的C++开发者，应该优先选择内建的`std::array`：

```
std::array<int, arraySize(keyVals)> mappedVals; // mappedVals长度是七
```

由于`arraySize`被声明称`noexcept`，这会帮助编译器生成更加优化的代码。可以从条款14查看更多详情。

函数参数

数组并不是C++唯一可以退化成指针的东西。函数类型可以被退化成函数指针，和我们之前讨论的数组的推导类似，函数可以被推华城函数指针：

```
void someFunc(int, double);     // someFunc是一个函数
                                // 类型是void(int, double)

template<typename T>
void f1(T param);               // 在f1中 参数直接按值传递

template<typename T>
void f2(T& param);              // 在f2中 参数是按照引用传递

f1(someFunc);                   // param被推导成函数指针
                                // 类型是void(*)(int, double)

f2(someFunc);                   // param被推导成函数指针
                                // 类型时void(&)(int, double)
```

这在实践中极少有不同，如果你知道数组到指针的退化，或许你也就会就知道函数到函数指针的退化。

所以你现在知道如下：`auto`相关的模板推导法则。我把最重要的部分单独在下面列出来。在通用引用中对待左值的处理有一点混乱，但是数组退化成指针和函数退化成函数指针的做法更加混乱呢。有时候你要对你的编译器和需求大吼一声，“告诉我到底类型推导成啥了啊！”当这种情况发生的时候，去参考条款4，因为它致力于让编译器告诉你是如何处理的。

要记住的东西
在模板类型推导的时候，有引用特性的参数的引用特性会被忽略
在推导通用引用参数的时候，左值会被特殊处理
在推导按值传递的参数时候，`const`和/或`volatile`参数会被视为非`const`和非`volatile`
在模板类型推导的时候，参数如果是数组或者函数名称，他们会被退化成指针，除非是用在初始化引用类型

条款二：理解 `auto` 类型推导

如果你已经阅读了条款1关于模板相关的类型推导，你就已经知道了机会所有关于 `auto` 的类型推导，因为除了一个例外，`auto` 类型推导就是模板类型推导。但是它怎么就会是模板类型推导呢？模板类型推导涉及模板和函数以及参数，但是 `auto` 和上面的这些没有任何的关系。

这是对的，但是没有关系。模板类型推导和 `auto` 类型推导是有一个直接的映射。有一个书面上的从一种情况转换成另外一种情况的算法。

在条款1，模板类型推导是使用下面的通用模板函数来解释的：

```
template<typename T>
void f(ParamType param);
```

在这里通常调用：

```
f(expr);                    // 使用一些表达式来当做调用f的参数
```

在调用 `f` 的地方，编译器使用 `expr` 来推导 `T` 和 `ParamType` 的类型。

当一个变量被声明为 `auto`，`auto` 相当于模板中的 `T`，而对变量做的相关的类型限定就像 `ParamType`。这用代码说明比直接解释更加容易理解，所以看下面的这个例子：

```
auto x = 27;
```

这里，对 `x` 的类型定义就仅仅是 `auto` 本身。从另一方面，在这个声明中：

```
const auto cx = x;
```

类型被声明成 `const auto`，在这儿：

```
const auto& rx = x;
```

类型被声明称 `const auto&`。在这些例子中推导 `x`，`cx`，`rx` 的类型的时候，编译器处理每个声明的时候就和处理对应的表达式初始化的模板：

```
template<typename T>                    // 推导x的类型的
void func_for_x(T param);               // 概念上的模板

func_for_x(27);                         // 概念上的调用：
                                        // param的类型就是x的类型

template<typename T>
void func_for_cx(const T param);        // 推导cx的概念上的模板

func_for_cx(x);                         // 概念调用：param的推导类型就是cx的类型

template<typename T>
void func_for_rx(const T& param);       // 推导rx概念上的模板

func_for_rx(x);                         // 概念调用：param的推导类型就是rx的类型
```

正如我所说，对`auto`的类型推导只存在一种情况的例外（这个后面就会讨论），其他的就和模板类型推导完全一样了。

条款1把模板类型推导划分成三部分，基于在通用的函数模板的`ParamType`的特性和`param`的类型声明。在一个用`auto`声明的变量上，类型声明代替了`ParamType`的作用，所以也有三种情况：

- 情况1：类型声明是一个指针或者是一个引用，但不是一个通用的引用
- 情况2：类型声明是一个通用引用
- 情况3：类型声明既不是一个指针也不是一个引用

我们已经看了情况1和情况3的例子：

```
auto x = 27;                            // 情况3（x既不是指针也不是引用）

const auto cx = x;                      // 情况3（cx二者都不是）

const auto& rx = x;                     // 情况1（rx是一个非通用的引用）
```

情况2正如你期待的那样：

```
auto&& uref1 = x;                       // x是int并且是左值
                                        // 所以uref1的类型是int&

auto&& uref2 = cx;                      // cx是int并且是左值
                                        // 所以uref2的类型是const int&

auto&& uref3 = 27;                      // 27是int并且是右值
                                        // 所以uref3的类型是int&&
```

条款1讲解了在非引用类型声明里，数组和函数名称如何退化成指针。这在`auto`类型推导上面也是一样：

```
const char name[] =                 // name的类型是const char[13]
    "R. N. Briggs";

auto arr1 = name;                   // arr1的类型是const char*

auto& arr2 = name;                  // arr2的类型是const char (&)[13]

void someFunc(int, double);         // someFunc是一个函数，类型是
                                    // void (*)(int, double)

auto& func2 = someFunc;             // func1的类型是
                                    // void (&)(int, double)
```

正如你所见，`auto`类型推导和模板类型推导工作很类似。它们就像一枚硬币的两面。

除了有一种情况是不一样的。我们从如果你想声明一个用27初始化的`int`，C++98你有两种语法选择：

```
int x1 = 27;
int x2(27);
```

C++11，通过标准支持的统一初始化（使用花括号初始化——译者注），可以添加下面的代码：

```
int x3 = { 27 };
int x4{ 27 };
```

综上四种语法，都会生成一种结果：一个拥有27数值的`int`。

但是正如条款5所解释的，使用`auto`来声明变量比使用固定的类型更好，所以在上述的声明中把`int`换成`auto`更好。最直白的写法就如下面的代码：

```
auto x1 = 27;
auto x2(27);
auto x3 = {27};
auto x4{ 27 };
```

上面的所有声明都可以编译，但是他们和被替换的相对应的语句的意义并不一样。头两个的确是一样的，声明一个初始化值为27的`int`。然而后面两个，声明了一个类型为`std::intializer_list<int>`的变量，这个变量包含了一个单一的元素27！

```
auto x1 = 27;                       // 类型时int，值是27

auto x2(27);                        // 同上

auto x3 = { 27 };                   // 类型是std::intializer_list<int>
                                    // 值是{ 27 }

auto x4{ 27 };                      // 同上
```

这和 `auto` 的一种特殊类型推导有关系。当使用一对花括号来初始化一个 `auto` 类型的变量的时候，推导的类型是 `std::intializer_list` 。如果这种类型无法被推导（比如在花括号中的变量拥有不同的类型），代码会编译错误。

```
auto x5 = { 1, 2, 3.0 };            // 错误！ 不能讲T推导成
                                    // std::intializer_list<T>
```

正如注释中所说的，在这种情况，类型推导会失败，但是认识到这里实际上是有两种类型推导是非常重要的。一种是 `auto: x5` 的类型被推导。因为 `x5` 的初始化是在花括号里面， `x5` 必须被推导成 `std::intializer_list` 。但是 `std::intializer_list` 是一个模板。实例是对一些 `T` 实例化成 `std::intializer_list<T>` ，这就意味着 `T` 的类型必须被推导出来。类型推导就在第二种的推导的范围上失败了。在这个例子中，类型推导失败是因为在花括号里面的数值并不是单一类型的。

对待花括号初始化的行为是 `auto` 唯一和模板类型推导不一样的地方。当 `auto` 声明变量被使用一对花括号初始化，推导的类型是 `std::intializer_list` 的一个实例。但是如果相同的初始化递给相同的模板，类型推导会失败，代码不能编译。

```
auto x = { 11, 23, 9 };             // x的类型是
                                    // std::initializer_list<int>

template<typename T>                // 和x的声明等价的
void f(T param);                    // 模板

f({ 11, 23, 9 });                   // 错误的！没办法推导T的类型
```

但是，如果你明确模板的 `param` 的类型是一个不知道 `T` 类型的 `std::initializer_list<T>` ：

```
template<typename T>
void f(std::initializer_list<T> initList);

f({ 11, 23, 9 });                   // T被推导成int，initList的
                                    // 类型是std::initializer_list<int>
```

所以 auto 和模板类型推导的本质区别就是 auto 假设花括号初始化代表的是std::initializer_list，但是模板类型推导却不是。

你可能对为什么 auto 类型推导有一个对花括号初始化有一个特殊的规则而模板的类型推导却没有感兴趣。我自己也非常奇怪。可是我一直没有能够找到一个有力的解释。但是法则就是法则，这就意味着你必须记住如果使用 auto 声明一个变量并且使用花括号来初始化它，类型推导的就是 std::initializer_list 。你必须习惯这种花括号的初始化哲学——使用花括号里面的数值来初始化是理所当然的。在C++11编程里面的一个经典的错误就是误被声明成 std::initializer_list ，而其实你是想声明另外的一种类型。这个陷阱使得一些开发者仅仅在必要的时候才会在初始化数值周围加上花括号。（什么时候是必要的会在条款7里面讨论。）

对于C++11，这是一个完整的故事，但是对于C++14来说，故事还要继续。C++14允许 auto 表示推导的函数返回值（参看条款3），而且C++14的lambda可能会在参数声明里面使用 auto 。但是，这里面的使用是复用了模板的类型推导，而不是 auto 的类型推导。所以一个使用 auto 声明的返回值的函数，返回一个花括号初始化就无法编译。

```
auto createInitList()
{
    return { 1, 2, 3 };              // 编译错误：不能推导出{ 1, 2, 3 }的类型
}
```

在C++14的lambda里面，当 auto 用在参数类型声明的时候也是如此：

```
std::vector<int> v;
…

auto resetV =
    [&v](const auto& newValue) { v = newValue; }    // C++14

…
resetV({ 1, 2, 3 });                 // 编译错误，不能推导出{ 1, 2, 3 }的类型
```

要记住的东西
auto 类型推导通常和模板类型推导类似，但是 auto 类型推导假定花括号初始化代表的类型是 std::initializer_list ，但是模板类型推导却不是这样
auto 在函数返回值或者lambda参数里面执行模板的类型推导，而不是通常意义的 auto 类型推导

条款三：理解 `decltype`

`decltype` 是一个怪异的发明。给定一个变量名或者表达式，`decltype` 会告诉你这个变量名或表达式的类型。`decltype` 的返回的类型往往也是你期望的。然而有时候，它提供的结果会使开发者极度抓狂而不得参考其他文献或者在线的Q&A网站。

我们从在典型的情况开始讨论，这种情况下 `decltype` 不会有令人惊讶的行为。
与 `templates` 和 `auto` 在类型推导中行为相比（请见条款一和条款二），`decltype` 一般只是复述一遍你所给他的变量名或者表达式的类型，如下：

```
const int i = 0;            // decltype(i) is const int

bool f(const Widget& w);    // decltype(w) is const Widget&
                            // decltype(f) is bool(const Widget&)
struct Point{
  int x, y;                 // decltype(Point::x) is int
};

Widget w;                   // decltype(w) is Widget

if (f(w)) ...               // decltype(f(w)) is bool

template<typename T>        // simplified version of std::vector
class vector {
public:
  ...
  T& operator[](std::size_t index);
  ...
};

vector<int> v;              // decltype(v) is vector<int>
...
if(v[0] == 0)               // decltype(v[0]) is int&
```

看到没有？毫无令人惊讶的地方。

在C++11中，`decltype` 最主要的用处可能就是用来声明一个函数模板，在这个函数模板中返回值的类型取决于参数的类型。举个例子，假设我们想写一个函数，这个函数中接受一个支持方括号索引（也就是"[]"）的容器作为参数，验证用户的合法性后返回索引结果。这个函数的返回值类型应该和索引操作的返回值类型是一样的。

操作子 `[]` 作用在一个对象类型为 `T` 的容器上得到的返回值类型为 `T&`。对 `std::deque` 一般是成立的，例如，对 `std::vector`，这个几乎是处处成立的。然而，
对 `std::vector<bool>`，`[]` 操作子不是返回 `bool&`，而是返回一个全新的对象。发生这种情况的原理将在条款六中讨论，对于此处重要的是容器的 `[]` 操作返回的类型是取决于容器的。

`decltype`使得这种情况很容易来表达。下面是一个模板程序的部分，展示了如何使用`decltype`来求返回值类型。这个模板需要改进一下，但是我们先推迟一下：

```
template<typename Container, typename Index>    // works, but
auto authAndAccess(Container& c, Index i)       // requires
  -> decltype(c[i])                             // refinements
{
  authenticateUser();
  return c[i];
}
```

将`auto`用在函数名之前和类型推导是没有关系的。更精确地讲，此处使用了`C++11`的尾随返回类型技术，即函数的返回值类型在函数参数之后声明(“->”后边)。尾随返回类型的一个优势是在定义返回值类型的时候使用函数参数。例如在函数`authAndAccess`中，我们使用了`c`和`i`定义返回值类型。在传统的方式下，我们在函数名前面声明返回值类型，`c`和`i`是得不到的，因为此时`c`和`i`还没被声明。

使用这种类型的声明，`authAndAccess`的返回值就是`[]`操作子的返回值，这正是我们所期望的。

`C++11`允许单语句的`lambda`表达式的返回类型被推导，在`C++14`中之中行为被拓展到包括多语句的所有的`lambda`表达式和函数。在上面`authAndAccess`中，意味着在`C++14`中我们可以忽略尾随返回类型，仅仅保留开头的`auto`。使用这种形式的声明，意味着将会使用类型推导。特别注意的是，编译器将从函数的实现来推导这个函数的返回类型：

```
template<typename Container, typename Index>         // C++14;
auto authAndAccess(Container &c, Index i)            // not quite
{                                                    // correct
  authenticateUser();
  return c[i];
}                                  // return type deduced from c[i]
```

条款二解释说，对使用`auto`来表明函数返回类型的情况，编译器使用模板类型推导。但是这样是回产生问题的。正如我们所讨论的，对绝大部分对象类型为`T`的容器，`[]`操作子返回的类型是`&T`，然而条款一提到，在模板类型推导的过程中,初始表达式的引用会被忽略。思考这对下面代码意味着什么：

```
std::deque<int> d;
...
authAndAccess(d, 5) = 10;       // authenticate user, return d[5],
                                // then assign 10 to it;
                                // this won't compile!
```

此处，`d[5]`返回的是`int&`，但是`authAndAccess`的`auto`返回类型声明将会剥离这个引用，从而得到的返回类型是`int`。`int`作为一个右值成为真正的函数返回类型。上面的代码尝试给一个右值`int`赋值为10。这种行为是在`C++`中被禁止的，所以代码无法编译通过。

为了让`authAndAccess`按照我们的预期工作，我们需要为它的返回值使用`decltype`类型推导，即指定`authAndAccess`要返回的类型正是表达式`c[i]`的返回类型。`C++`的拥护者们预期到在某种情况下有使用`decltype`类型推导规则的需求，并将这个功能在`C++14`中通过`decltype(auto)`实现。这使这对原本的冤家（`decltype`和`auto`）在一起完美地发挥作用：`auto`指定需要推导的类型，`decltype`表明在推导的过程中使用`decltype`推导规则。因此，我们可以重写`authAndAccess`如下：

```
template<typename Container, typename Index>  // C++14; works,
decltype(auto)                                // but still
authAndAccess(Container &c, Index i)          // requires
{                                             // refinement
  authenticateUser();
  return c[i];
}
```

现在`authAndAccess`的返回类型就是`c[i]`的返回类型。在一般情况下，`c[i]`返回`T&`，`authAndAccess`就返回`T&`，在不常见的情况下，`c[i]`返回一个对象，`authAndAccess`也返回一个对象。

`decltype(auto)`并不仅限使用在函数返回值类型上。当时想对一个表达式使用`decltype`的推导规则时，它也可以很方便的来声明一个变量：

```
Widget w;
const Widget& cw = w;
auto myWidget1 = cw;                    // auto type deduction
                                        // myWidget1's type is Widget
decltype(auto) myWidget2 = cw           // decltype type deduction:
                                        // myWidget2's type is
                                        //  const Widget&
```

我知道，到目前为止会有两个问题困扰着你。一个是我们前面提到的，对`authAndAccess`的改进。我们在这里讨论。

再次看一下`C++14`版本的`authAndAccess`的声明：

```
template<typename Container, typename Index>
decltype(auto) anthAndAccess(Container &c, Index i);
```

这个容器是通过非`const`左值引用传入的，因为通过返回一个容器元素的引用是来修改容器是被允许的。但是这也意味着不可能将右值传入这个函数。右值不能和一个左值引用绑定（除非是`const`的左值引用，这不是这里的情况）。

诚然，传递一个右值容器给`authAndAccess`是一种极端情况。一个右值容器作为一个临时对象，在`anthAndAccess`所在语句的最后被销毁，意味着对容器中一个元素的引用（这个引用通常是`authAndAccess`返回的）在创建它的语句结束的地方将被悬空。然而，这对于传给`authAndAccess`一个临时对象是有意义的。一个用户可能仅仅想拷贝一个临时容器中的一个元素，例如：

```
std::deque<std::string> makeStringDeque(); // factory function
// make copy of 5th element of deque returned
// from makeStringDeque
auto s = authAndAccess(makeStringDeque(), 5);
```

支持这样的应用意味着我们需要修改`authAndAccess`的声明来可以接受左值和右值。重载可以解决这个问题（一个重载负责左值引用参数，另外一个负责右值引用参数），但是我们将有两个函数需要维护。避免这种情况的一个方法是使`authAndAccess`有一个既可以绑定左值又可以绑定右值的引用参数，条款24将说明这正是统一引用（`universal reference`）所做的。因此`authAndAccess`可以像如下声明：

```
template<typename Container, typename Index>  // c is now a
decltype(auto) authAndAccess(Container&& c,   // universal
                             Index i);        // reference
```

在这个模板中，我们不知道我们在操作什么类型的容器，这也意味着我们等同地忽略了它用到的索引对象的类型。对于一个不清楚其类型的对象使用传值传递通常会冒一些风险，比如因为不必要的复制而造成的性能降低，对象切片的行为问题，被同事嘲笑，但是对容器索引的情况，正如一些标准库的索引（`std::string, std::vector, std::deque`的`[]`操作）按值传递看上去是合理的，因此对它们我们仍坚持按值传递。

然而，我们需要更新这个模板的实现，将`std::forward`应用给统一引用，使得它和条款25中的建议是一致的。

```
template<typename Container, typename Index>     // final
decltype(auto)                                   // C++14
authAndAccess(Container&& c, Index i)            // version
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

这个实现可以做我们期望的任何事情，但是它要求使用支持 C++14 的编译器。如果你没有一个这样的编译器，你可以使用这个模板的 C++11 版本。它出了要你自己必须指定返回类型以外，和对应的 C++14 版本是完全一样的，

```
template<typename Container, typename Index>    // final
auto                                            // C++11
authAndAccess(Container&& c, Index i)           // version
-> decltype(std::forward<Container>(c)[i])
{
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

另外一个容易被你挑刺的地方是我在本条款开头的那句话：decltype 几乎所有时候都会输出你所期望的类型，但是有时候它的输出也会令你吃惊。诚实的讲，你不太可能遇到这种以外，除非你是一个重型库的实现人员。

为了彻底的理解 decltype 的行为，你必须使你自己对一些特殊情况比较熟悉。这些特殊情况太晦涩难懂，以至于很少有书会像本书一样讨论，但是同时也可以增加我们对 decltype 的认识。

对一个变量名使用 decltype 得到这个变量名的声明类型。变量名属于左值表达式，但这并不影响 decltype 的行为。然而，对于一个比变量名更复杂的左值表达式，decltype 保证返回的类型是左值引用。因此说，如果一个非变量名的类型为 T 的左值表达式，decltype 报告的类型是 T& 。这很少产生什么影响，因为绝大部分左值表达式的类型有内在的左值引用修饰符。例如，需要返回左值的函数返回的总是左值引用。

这种行为的意义是值得我们注意的。但是在下面这个语句中

```
int x = 0;
```

x 是一个变量名，因此 decltyper(x) 是 int 。但是如果给 x 加上括号"(x)"就得到一个比变量名复杂的表达式。作为变量名，x 是一个左值，同时 C++ 定义表达式 (x) 也是左值。因此 decltype((x)) 是 int& 。给一个变量名加上括号会改变 decltype 返回的类型。

在 C++11 中，这仅仅是个好奇的探索，但是和 C==14 中对 decltype(auto) 支持相结合，函数中返回语句的一个细小改变会影响对这个函数的推导类型。

```
decltype(auto) f1()
{
  int x = 0;
  ...
  return x;        // decltype(x) is int, so f1 returns int
}

decltype(auto) f2()
{
  int x = 0;
  return (x);     // decltype((x)) is int&, so f2 return int&
}
```

`f2` 不仅返回值类型与 `f1` 不同，它返回的是对一个局部变量的引用。这种类型的代码将把你带上一个为定义行为的快速列车-你完全不想登上的列车。

最主要的经验教训就是当使用 `decltype(auto)` 时要多留心一些。被推导的表达式中看上去无关紧要的细节都可能影响 `decltype` 返回的类型。为了保证推导出的类型是你所期望的，请使用条款4中的技术。

同时不能更大视角上的认识。当然， `decltype` （无论只有 `decltype` 或者还是和 `auto` 联合使用）有可能偶尔会产生类型推导的惊奇行为，但是这不是常见的情况。一般情况下， `decltype` 会产生你期望的类型。将 `decltype` 应用于变量名无非是正确的，因为在这种情况下， `decltype` 做的就是报告这个变量名的声明类型。

要记住的东西
`decltype` 几乎总是得到一个变量或表达式的类型而不需要任何修改
对于非变量名的类型为 `T` 的左值表达式， `decltype` 总是返回 `T&`
`C++14` 支持 `decltype(auto)` ，它的行为就像 `auto` ,从初始化操作来推导类型，但是它推导类型时使用 `decltype` 的规则

条款4：知道如何查看类型推导

对类型推导结果的查看的工具的选择和你在软件开发过程中的相关信息有关系。我们要探讨三种可能：在你编写代码的时候，在编译的时候和在运行的时候得到类型推导的信息。

IDE编辑器

在IDE里面的代码编辑器里面当你使用光标悬停在实体之上，常常可以显示出程序实体（例如变量，参数，函数等等）的类型。举一个例子，下面的代码：

```
const int theAnswer = 42;
auto x = theAnswer;
auto y = &theAnswer;
```

一个IDE的编辑器很可能会展示出 `x` 的推导的类型是 `int` ，`y` 的类型是 `const int*` 。

对于这样的情况，你的代码必须处在一个差不多可以编译的状态，因为这样可以使得IDE接受这种在IDE内部运行这的一个C++编译器（或者至少是一个前端）的信息。如果那个编译器无法能够有足够的能力去感知你的代码并且parse你的代码然后去执行类型推导，他就无法展示对应推导的类型了。

对于简单的类型例如 `int` ，IDE里面的信息是正常的。但是我们随后会发现，涉及到更加复杂的类型的时候，从IDE里面得到的信息并不一定是有帮助性的。

编译器诊断

一个有效的让编译器展示类型的办法就是故意制造编译问题。编译的错误输出会报告会和捕捉到的类型相关错误。

假设，举个例子，我们希望看在上面例子中的 `x` 和 `y` 被推导的类型。我们首先声明一个类模板，但是并不定义这个模板。就像下面优雅的做法：

```
template<typename T>                    // 声明TD
class TD;                               // TD == "Type Displayer"
```

尝试实例化这个模板会导致错误信息，因为没有模板的定义实现。想看 `x` 和 `y` 被推导的类型，只要尝试去使用这些类型去实例化 `TD` ：

```
TD<decltype(x)> xType;                  // 引起的错误
TD<decltype(y)> yType;                  // 包含了x和y的类型
```

我使用的变量名字的形式 `variableNameType` 是因为这样有利于输出的错误信息可以帮助我定位我要寻找的信息。对上面的代码，我的一个编译器输出了诊断信息，其中的一部分如下：（我把我们关注的类型信息高亮了（原文中高亮了模板中的 `int` 和 `const int*` ，但是Markdown在代码block中操作粗体比较麻烦，译文中没有加粗——译者注））：

```
error: aggregate 'TD<int> xType' has incomplete type and cannot be defined
error: aggregate 'TD<const int *> yType' has incomplete type and cannot be defined
```

另一个编译器提供相同的信息，但是格式不太一样：

```
error: 'xType' uses undefined class 'TD<int>'
error: 'yType' uses undefined class 'TD<const int *>'
```

排除格式的区别，我测试了所有的编译器都会在这种代码的技术中输出有用的错误信息。

运行时输出

`printf` 到运行的时候可以用来显示类型信息（这并不是我推荐你使用 `printf` 的原因），但是它提供了对输出格式的完全掌控。挑战就在于你要创造一个你关心的对象的输出的格式控制展示的textual。“这还不容易，”你会这样想，“就是用 `typeid` 和 `std::type_info::name` 来救场啊。”在后续的对 `x` 和 `y` 的类型推导中，你可以发现你可以这样写：

```
std::cout << typeid(x).name() << '\n'; // display types for
std::cout << typeid(y).name() << '\n'; // x and y
```

这是基于对类似于 `x` 或者 `y` 运算 `typeid` 可以得到一个 `std::type_info` 对象，`std::type_info` 有一个成员函数，`name` 可以提供一个C-style的字符串（也就是 `const char*` ）代表了类型的名字。

调用 `std::type_info::name` 并不会确定返回有意义的东西，但是实现上是有帮助性质的。帮助是多种多样的。举一个例子，GNU和Clang编译器返回 `x` 的类型是“ `i` ”，`y` 的类型是“ `PKi` ”。这些编译器的输出结果你一旦学会就可以理解他们，“ `i` ”意味着“ `int` ”，“ `PK` ”意味着“pointer to ~~konst~~ const”（所有的编译器都支持一个工具，`C++filt` ，它可以解析这样的“乱七八糟”的类型。）微软的编译器提供更加直白的输出：“ `int` ”对 `x` ，“ `int const*` ”对 `y` 。

因为这些结果对 `x` 和 `y` 而言都是正确的，你可能认为类型输出的问题就此解决了，但是这并不能轻率。考虑一个更加复杂的例子：

```
template<typename T>                // template function to
void f(const T& param);             // be called

std::vector<Widget> createVec();    // 工厂方法

const auto vw = createVec();        // init vw w/factory return

if (!vw.empty()) {
    f(&vw[0]);                      // 调用f
    …
}
```

在代码中，涉及了一个用户定义的类型（`Widget`），一个STL容器（`std::vector`），一个`auto`变量（`vw`），这对你的编译器的类型推导的可视化是非常具有表现性的。举个例子，想看到模板类型参数`T`和`f`的函数模板参数`param`。

在问题中没有`typeid`是很直接的。在`f`中添加一些代码去展示你想要的类型：

```
template<typename T>
void f(const T& param)
{
    using std::cout;
    cout << "T = " << typeid(T).name() << '\n';          // 展示T
    cout << "param = " << typeid(param).name() << '\n'; // 展示param的类型
    …
}
```

使用GNU和Clang编译器编译会输出如下结果：

```
T = PK6Widget
param = PK6Widget
```

我们已经知道对于这些编译器，`PK`意味着“pointer to `const`”，所以比较奇怪的就是数字6，这是在后面跟着的类的名字(`Widget`)的字母字符的长度。所以这些编译器就告我们`T`和`param`的类型都是`const Widget*`。

微软的编译器输出：

```
T = class Widget const *
param = class Widget const *
```

三种不同的编译器都产出了相同的建议性信息，这表明信息是准确的。但是更加仔细的分析，在模板`f`中，`param`的类型是`const T&`。`T`和`param`的类型是一样的难道不会感到奇怪吗？举个例子，如果`T`是`int`，`param`的类型应该是`const int&`——根本不是相同的类型。

悲剧的是，`std::type_info::name`的结果并不可靠。在这种情况下，举个例子，所有的三种编译器报告的`param`的类型都是不正确的。更深入的话，它们本来就是不正确的，因为`std::type_info::name`的特化指定了类型会被当做它们被传给模板函数的时候的按值传递的参数。正如条款1所述，这就意味着如果类型是一个引用，他的引用特性会被忽略，如果在忽略引用之后存在`const`（或者`volatile`），它的`const`特性（或者`volatile`特性）会被忽略。这就是为什么`param`的类型——`const Widget * const &`——被报告成了`const Widget*`。首先类型的引用特性被去掉了，然后结果参数指针的`const`特性也被消除了。

同样的悲剧，由IDE编辑器显示的类型信息也并不准确——或者说至少并不可信。对之前的相同的例子，一个我知道的IDE的编辑器报告出`T`的类型（我不打算说）：

```
const
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget> >::_Alloc>::value_type>::value_type *
```

还是这个相同的IDE编辑器，`param`的类型是：

```
const std::_Simple_types<...>::value_type *const &
```

这个没有`T`的类型那么吓人，但是中间的“...”会让你感到困惑，直到你发现这是IDE编辑器的一种说辞“我们省略所有`T`类型的部分”。带上一点运气，你的开发环境也许会对这样的代码有着更好的表现。

如果你更加倾向于库而不是运气，你就应该知道`std::type_info::name`可能在IDE中会显示类型失败，但是Boost TypeIndex库（经常写做Boost.TypeIndex）是被设计成可以成功显示的。这个库并不是C++标准的一部分，也不是IDE和模板的一部分。更深层的是，事实上Boost库（在boost.com）是一个跨平台的，开源的，并且基于一个偏执的团队都比较喜欢的协议。这就意味着基于标准库之上使用Boost库的代码接近于一个跨平台的体验。

这里展示了一段我们使用Boost.TypeIndex的函数`f`精准的输出类型信息：

```
#include <boost/type_index.hpp>
template<typename T>
void f(const T& param)
{
    using std::cout;
    using boost::typeindex::type_id_with_cvr;

    // show T
    cout << "T = "
         << type_id_with_cvr<T>().pretty_name()
         << '\n';

    // show param's type
    cout << "param = "
         << type_id_with_cvr<decltype(param)>().pretty_name()
         << '\n';
    …
}
```

这个模板函数 `boost::typeindex::type_id_with_cvr` 接受一个类型参数（我们想知道的类型信息）来正常工作，它不会去除 `const` ， `volatile` 或者引用特性（这也就是模板中的“ `cvr` ”的意思）。返回的结果是个 `boost::typeindex::type_index` 对象，其中的 `pretty_name` 成员函数产出一个 `std::string` 包含一个对人比较友好的类型展示的字符串。

通过这个 `f` 的实现，再次考虑之前使用 `typeid` 导致推导出现错误的 `param` 类型信息：

```
std::vector<Widget> createVec();    // 工厂方法

const auto vw = createVec();        // init vw w/factory return

if (!vw.empty()) {
    f(&vw[0]);                      // 调用f
    …
}
```

在GNU和Clang的编译器下面，Boost.TypeIndex输出（准确）的结果：

```
T = Widget const*
param = Widget const* const&
```

微软的编译器实际上输出的结果是一样的：

```
T = class Widget const *
param = class Widget const * const &
```

这种接近相同的结果很漂亮，但是需要注意IDE编辑器，编译器错误信息，和类似于Boost.TypeIndex的库仅仅是一个对你编译类型推导的一种工具而已。所有的都是有帮助意义的，但是到目前为止，没有什么关于类型推导法则1-3的替代品。

要记住的东西
类型推导的结果常常可以通过IDE的编辑器，编译器错误输出信息和Boost TypeIndex库的结果中得到
一些工具的结果不一定有帮助性也不一定准确，所以对C++标准的类型推导法则加以理解是很有必要的

第一章 `auto` 关键字

概念上，`auto`是竟可能的简单，但是它比看上去要精细的多。用它来节省敲击键盘，当然没有问题，但是它也防止了困扰手动类型声明的正确性和性能问题。因此，一些`auto`类型的推导结果虽然完成符合规定的算法，但是从程序员的角度来看是错误的。在这种情况下，知道如何引导`auto`得到正确的结果是很重要的，因为回到手动声明类型虽然是一个变通方案，但是应该尽量避免。

这简短的一章涵盖了所有`auto`的输入和输出。

条款五：优先使用 `auto` 而非显式类型声明

使用下面语句是简单快乐的

```
int x;
```

等等。见鬼，我忘记初始化 `x` 了，因此它的值是无法确定的。也许，它会被初始化为0。但是这根据上下文语境决定。这真令人叹息。

不要介意。我们来看看一个要通过迭代器解引用初始化的局部变量声明的简单与快乐。

```
template<typename It>
void dwim(It b, It e)
{
  while(b != e){
  typename std::iterator_traits<It>::value_type
    currValue = *b;
  ...
  }
}
```

额。`typename std::iterator_traits<It>::value_type` 来表示被迭代器指向的值的类型？真的是这样吗？我必须努力不去想这是多么有趣的一件事。见鬼。等等，难道我已经说出来了。

好吧，有三个令人愉悦的地方：声明一个封装好的局部变量的类型带来的快乐。是的，这是没有问题的。一个封装体的类型只有编译器知道，因此不能被显示的写出来。哎，见鬼。

见鬼，见鬼，见鬼！使用 `C++` 编程并不是它本该有的愉悦体验。

是的，过去的确不是。但是由于 `C++11` ，得益于 `auto` ，这些问题都消失了。 `auto` 变量从他们的初始化推导出其类型，所以它们必须被初始化。这就意味着你可以在现代的 `C++` 高速公路上对没有初始化的变量的问题说再见了。

```
int x1;                 // potentially uninitialized
auto x2;                // error! initializer required
auto x3 = 0;            // fine, x's value is well-defined
```

如上所述，高速公路上不再有由于解引用迭代器的声明局部变量而引起的坑坑洼洼。

```
template<typename It>
void dwim(It b, It e)
{
  while(b != e){
    auto currValue = *b;
  ...
  }
}
```

由于 `auto` 使用类型推导（参见条款2），它可以表示那些仅仅被编译器知晓的类型：

```
auto dereUPLess =                          // comparison func.
  [](const std::unique_ptr<Widget>& p1,    // for Widgets
     const std::unique_ptr<Widget>& p2)    // pointed to by
  { return *p1 < *p2};                      // std::unique_ptrs
```

非常酷。在 `C++14` 中，模板（原文为temperature）被进一步丢弃，因为使用 `lambda` 表达式的参数可以包含 `auto` ：

```
auto derefLess =                           // C++14 comparison
[](const auto& p1,                         // function for
   const auto& p2)                         // values pointed
{ return *p1 < *p2; };
```

尽管非常酷，也许你在想，我们不需要使用 `auto` 去声明一个持有封装体的变量，因为我们可以使用一个 `std::function` 对象。这是千真万确的，我们可以这样干，但是也许那不是你正在思考的东西。也许你在思考“ `std::function` 是什么东东？”。因此让我们解释清楚。

`std::function` 是 `C++11` 标准库的一个模板，它可以使函数指针普通化。鉴于函数指针只能指向一个函数，然而， `std::function` 对象可以应用任何可以被调用的对象，就像函数。就像你声明一个函数指针的时候，必须指明这个函数指针指向的函数的类型，你产生一个 `std::function` 对象时，你也指明它要引用的函数的类型。你可以通过 `std::function` 的模板参数来完成这个工作。例如，有声明一个名为 `func` 的 `std::function` 对象，它可以引用有如下特点的可调用对象：

```
bool(const std::unique_ptr<Widget> &,     // C++11 signature for
     const std::unique_ptr<Widget> &)     // std::unique_ptr<Widget>
                                          // comparison funtion
```

你可以这么写：

```
std::function<bool(const std::unique_ptr<Widget> &,
                   const std::unique_ptr<Widget> &)> func;
```

因为 lambda 表达式得到一个可调用对象，封装体可以存储在 std::function 对象里面。这意味着，我们可以声明不适用 auto 的 C++11 版本的 dereUPLess 如下：

```
std::function<bool(const std::unique_ptr<Widget>&,
                   const std::unique_ptr<Widget>&)>
  derefUPLess = [](const std::unique_ptr<Widget>& p1,
                   const std::unique_ptr<Widget>& p2)
                  {return *p1 < *p2; };
```

意识到需要重复参数的类型这种冗余的语法是重要的，使用 std::function 和使用 auto 并不一样。一个使用 auto 声明持有一个封装的变量和封装体有同样的类型，也仅使用和封装体同样大小的内存。持有一个封装体的被 std::function 声明的变量的类型是 std::function 模板的一个实例，并且对任何类型只有一个固定的大小。这个内存大小可能不能满足封装体的需求。出现这种情况时， std::function 将会开辟堆空间来存储这个封装体。导致的结果就是 std::function 对象一般会比 auto 声明的对象使用更多的内存。由于实现细节中，约束内嵌的使用和提供间接函数的调用，通过 std::function 对象来调用一个封装体比通过 auto 对象要慢。换言之， std::function 方法通常体积比 auto 大，并且慢，还有可能导致内存不足的异常。就像你在上面一个例子中看到的，使用 auto 的工作量明显小于使用 std::function 。持有一个封装体时， auto 和 std::function 之间的竞争，对 auto 简直就是游戏。（一个相似的论点也成立对于持有 std::blind 调用结果的 auto 和 std::function ，但是在条款34中，我将竭尽所能的说服你尽可能使用 lambda 表达式，而不是 std::blind ）。

auto 的优点除了可以避免未初始化的变量，变量声明引起的歧义，直接持有封装体的能力。还有一个就是可以避免“类型截断”问题。下面有个例子，你可能见过或者写过：

```
std::vector<int> v;
...
unsigned sz = v.size();
```

v.size() 定义的返回类型是 std::vector<int>::size_type ，但是很少有开发者对此十分清楚。 std::vector<int>::size_type 被指定为一个非符号的整数类型，因此很多程序员认为 unsigned 类型是足够的，然后写出了上面的代码。这将导致一些有趣的后果。比如说在32位 Windows 系统上， unsigned 和 std::vector<int>::size_type 有同样的大小，但是在64位的 Windows 上， unsigned 是32bit的，而 std::vector<int>::size_type 是64bit的。这意味着上面的代码在32位 Windows 系统上工作良好，但是在64位 Windows 系统上时有可能不正确，当应用程序从32位移植到64位上时，谁又想在这种问题上浪费时间呢？ 使用 auto 可以保证你不必被上面的东西所困扰：

```
auto sz = v.size()    // sz's type is std::vector<int>::size_type
```

仍然不太确定使用 auto 的高明之处？看看下面的代码：

```
std::unordered_map<std::string, int> m;
...

for (const std::pair<std::string, int>& p : m)
{
...                 // do something with p
}
```

这看上去完美合理。但是有一个问题，你看出来了吗？意识到 `std::unorder_map` 的 `key` 部分是 `const` 类型的，在哈希表中的 `std::pair` 的类型不是 `std::pair<std::string, int>` ，而是 `std::pair<const std::sting, int>` 。但是这不是循环体外变量 `p` 的声明类型。后果就是，编译器竭尽全力去找到一种方式，把 `std::pair<const std::string, int>` 对象（正是哈希表中的内容）转化为 `std::pair<std::string, int>` 对象（ `p` 的声明类型）。这个过程将通过复制 `m` 的一个元素到一个临时对象，然后将这个临时对象和 `p` 绑定完成。在每个循环结束的时候这个临时对象将被销毁。如果是你写了这个循环，你将会感觉代码的行为令人吃惊，因为你本来想简单地将引用 `p` 和 `m` 的每个元素绑定的。 这种无意的类型不匹配可以通过 `auto` 解决

```
for (const auto& p : m)
{
  ...                   // as before
}
```

这不仅仅更高效，也更容易敲击代码。更近一步，这个代码还有一些吸引人的特性，比如如果你要取 `p` 的地址，你的确得到一个指向 `m` 的元素的指针。如果不使用 `auto` ，你将得到一个指向临时对象的指针——这个临时对象在每次循环结束时将被销毁。

上面两个例子中——在应该使用 `std::vector<int>::size_type` 的时候使用 `unsigned` 和在该使用 `std::pair<const std::sting, int>` 的地方使用 `std::pair<std::string, int>` ——说明显式指定的类型是如何导致你万万没想到的隐式的转换的。如果你使用 `auto` 作为目标变量的类型，你不必为你声明类型和用来初始化它的表达式类型之间的不匹配而担心。

有好几个使用 `auto` 而不是显式类型声明的原因。然而， `auto` 不是完美的。 `auto` 变量的类型都是从初始化它的表达式推导出来的，一些初始化表达式并不是我们期望的类型。发生这种情况时，你可以参考条款2和条款6来决定怎么办，我不在此处展开了。相反，我将我的精力集中在你将传统的类型声明替代为 `auto` 时带来的代码可读性问题。

首先，深呼吸放松一下。 `auto` 是一个可选项，不是必须项。如果根据你的专业判断，使用显式的类型声明比使用 `auto` 会使你的代码更加清晰或者更好维护，或者在其他方面更有优势，你可以继续使用显式的类型声明。牢记一点， `C++` 并没有在这个方面有什么大的突破，这种技术在其他语言中被熟知，叫做类型推断(`type inference`)。其他的静态类型过程式语言（像 `C#` ， `D` ， `Scala` ， `Visual Basic` ）也有或多或少等价的特点，对静态类型的函数编程语言（像 `ML` ， `Haskell` ， `OCaml` ， `F#` 等）另当别论。一定程度上说，这是受到动态类型语

言的成功所启发，比如 `Perl` ， `Python` ， `Ruby` ，在这些语言中很少显式指定变量的类型。软件开发社区对于类型推断有很丰富的经验，这些经验表明这些技术和创建及维护巨大的工业级代码库没有矛盾。

一些开发者被这样的事实困扰，使用 `auto` 会消除看一眼源代码就能确定对象的类型的能力。然而，IDE提示对象类型的功能经常能缓解这个问题（甚至考虑到在条款4中提到的IDE的类型显示问题），在很多情况下，一个对象类型的摘要视图和显示完全的类型一样有用。比如，摘要视图足以让开发者知道这个对象是容器还是计数器或者一个智能指针，而不需要知道这个容器，计数器或者智能指针的确切特性。假设比较好的选择变量名字，这样的摘要类型信息几乎总是唾手可得的。

事实是显式地写出类型可能会引入一些难以察觉的错误，导致正确性或者效率问题，或者两者兼而有之。除此之外， `auto` 类型会自动的改变如果初始化它的表达式改变后，这意味着通过使用 `auto` 可以使代码重构变得更简单。举个例子，如果一个函数被声明为返回 `int` ，但是你稍后决定返回 `long` 可能更好一些，如果你把这个函数的返回结果存储在一个 `auto` 变量中，在下次编译的时候，调用代码将会自动的更新。结果如果存储在一个显式声明为 `int` 的变量中，你需要找到所有调用这个函数的地方然后改写他们。

要记住的东西
`auto` 变量一定要被初始化，并且对由于类型不匹配引起的兼容和效率问题有免疫力，可以简单化代码重构，一般会比显式的声明类型敲击更少的键盘
`auto` 类型的变量也受限于条款2和条款6中描述的陷阱

条款6：当auto推导出非预期类型时应当使用显式的类型初始化

条款5解释了使用 `auto` 关键字去声明变量，这样就比直接显示声明类型提供了一系列的技术优势，但是有时候 `auto` 的类型推导会和你想的南辕北辙。举一个例子，假设我有一个函数接受一个 `Widget` 返回一个 `std::vector<bool>` ，其中每个 `bool` 表征 `Widget` 是否接受一个特定的特性：

```
std::vector<bool> features(const Widget& w);
```

进一步的，假设第五个bit表示 `Widget` 是否有高优先级。我们可以这样写代码：

```
Widget w;
…
bool highPriority = features(w)[5];          // w是不是个高优先级的？
…
processWidget(w, highPriority);              // 配合优先级处理w
```

这份代码没有任何问题。它工作正常。但是如果我们做一个看起来无伤大雅的修改，把 `highPriority` 的显式的类型换成 `auto` ：

```
auto highPriority = features(w)[5];          // w是不是个高优先级的？
```

情况变了。所有的代码还是可以编译，但是他的行为变得不可预测：

```
processWidget(w, highPriority);              // 未定义行为
```

正如注释中所提到的，调用 `processWidget` 现在会导致未定义的行为。但是为什么呢？答案是非常的令人惊讶的。在使用 `auto` 的代码中， `highPriority` 的类型已经不是 `bool` 了。尽管 `std::vector<bool>` 从概念上说是 `bool` 的容器，对 `std::vector<bool>` 的 `operator[]` 运算符并不一定是返回容器中的元素的引用（ `std::vector::operator[]` 对所有的类型都返回引用，就是除了 `bool` ）。事实上，他返回的是一个 `std::vector<bool>::reference` 对象（是一个在 `std::vector<bool>` 中内嵌的class）。

`std::vector<bool>::reference` 存在是因为 `std::vector<bool>` 是对 `bool` 数据封装的模板特化，一个bit对应一个 `bool` 。这就给 `std::vector::operator[]` 带来了问题，因为 `std::vector<T>` 的 `operator[]` 应该返回一个 `T&` ，但是C++禁止bits的引用。没办法返回一个 `bool&` ， `std::vector<T>` 的 `operator[]` 于是就返回了一个行为上和 `bool&` 相似的对象。想

要这种行为成功，`std::vector<bool>::reference` 对象必须能在 `bool&` 的能处的语境中使用。在 `std::vector<bool>::reference` 对象的特性中，是他隐式的转换成 `bool` 才使得这种操作得以成功。（不是转换成 `bool&` ，而是 `bool` 。去解释详细的 `std::vector<bool>::reference` 对象如何模拟一个 `bool&` 的行为有有些偏离主题，所以我们就只是简单的提一下这种隐式转换只是这种技术中的一部。）

在大脑中带上这种信息，再次阅读原先的代码：

```
bool highPriority = features(w)[5];          // 直接显示highPriority的类型
```

这里，`features` 返回了一个 `std::vector<bool>` 对象，在这里 `operator[]` 被调用。`operator[]` 返回一个 `std::vector<bool>::reference` 对象，这个然后隐式的转换成 `highPriority` 需要用来初始化的 `bool` 类型。于是就以 `features` 返回的 `std::vector<bool>` 的第五个bit的数值来结束 `highPriority` 的数值，这也是我们所预期的。

和使用 `auto` 的 `highPriority` 声明进行对比：

```
auto highPriority = features(w)[5];          // 推导highPriority的类型
```

这次，`features` 返回一个 `std::vector<bool>` 对象，而且，`operator[]` 再次被调用。`operator[]` 继续返回一个 `std::vector<bool>::reference` 对象，但是现在有一个变化，因为 `auto` 推导 `highPriority` 的类型。`highPriority` 根本并没有 `features` 返回的 `std::vector<bool>` 的第五个bit的数值。

数值和 `std::vector<bool>::reference` 是如何实现的是有关系的。一种实现是这样的对象包含一个指向包含bit引用的机器word的指针，在word上面加上偏移。考虑这个对 `highPriority` 的初始化的意义，假设 `std::vector<bool>::reference` 的实现是恰当的。

调用 `features` 会返回一个临时的 `std::vector<bool>` 对象。这个对象是没有名字的，但是对于这个讨论的目的，我会把它叫做 `temp` ，`operator[]` 是在 `temp` 上调用的，`std::vector<bool>::reference` 返回一个由 `temp` 管理的包含一个指向一个包含bits的数据结构的指针，在word上面加上偏移定位到第五个bit。`highPriority` 也是一个 `std::vector<bool>::reference` 对象的一份拷贝，所以 `highPriority` 也在 `temp` 中包含一个指向word的指针，加上偏移定位到第五个bit。在这个声明的结尾，`temp` 被销毁，因为它是个临时对象。因此，`highPriority` 包含一个野指针，这也就是调用 `processWidget` 会造成未定义的行为的原因：

```
processWidget(w, highPriority);          // 未定义的行为，highPriority包含野指针
```

`std::vector<bool>::reference` 是代理类的一个例子：一个类的存在是为了模拟和对外行为和另外一个类保持一致。代理类在各种各样的目的上被使用。`std::vector<bool>::reference` 的存在是为了提供一个对 `std::vector<bool>` 的 `operator[]` 的错觉，让它返回一个对bit的引

用，而且标准库的智能指针类型（参考第4章）也是一些对托管的资源的代理类，使得他们的资源管理类似于原始指针。代理类的功能是良好确定的。事实上，“代理”模式是软件设计模式中的最坚挺的成员之一。

一些代理类被设计用来隔离用户。这就是 `std::shared_ptr` 和 `std::unique_ptr` 的情况。另外一些代理类是为了一些或多或少的不可见性。 `std::vector<bool>::reference` 就是这样一个“不可见”的代理，和他类似的是 `std::bitset` ，对应的是 `std::bitset::reference` 。

同时在一些C++库里面的类存在一种被称作表达式模板的技术。这些库最开始是为了提高数值运算的效率。提供一个 `Matrix` 类和 `Matrix` 对象 `m1, m2, m3 and m4` ，举一个例子，下面的表达式：

```
Matrix sum = m1 + m2 + m3 + m4;
```

可以计算的更快如果 `Matrix` 的 `operator+` 返回一个结果的代理而不是结果本身。这是因为，对于两个 `Matrix` ， `operator+` 可能返回一个类似于 `Sum<Matrix, Matrix>` 的代理类而不是一个 `Matrix` 对象。和 `std::vector<bool>::reference` 一样，这里会有一个隐式的从代理类到 `Matrix` 的转换，这个可能允许 `sum` 从由 `=` 右边的表达式产生的代理对象进行初始化。（其中的对象可能会编码整个初始化表达式，也就是，变成一种类似于 `Sum<Sum<Sum<Matrix, Matrix>, Matrix>, Matrix>` 的类型。这是一个客户端需要屏蔽的类型。）

作为一个通用的法则，“不可见”的代理类不能和 `auto` 愉快的玩耍。这种类常常它的生命周期不会被设计成超过一个单个的语句，所以创造这样的类型的变量是会违反库的设计假定。这就是 `std::vector<bool>::reference` 的情况，而且我们可以看到这种违背约定的做法会导致未定义的行为。

因此你要避免使用下面的代码的形式：

```
auto someVar = expression of "invisible" proxy class type;
```

但是你怎么能知道代理类被使用呢？软件使用它们的时候并不可能会告知它们的存在。它们是不可见的，至少在概念上！一旦你发现了他们，难道你就必须放弃使用 `auto` 加之条款5所声明的 `auto` 的各种好处吗？

我们先看看怎么解决如何发现它们的问题。尽管“不可见”的代理类被设计用来fly beneath programmer radar in day-to-day use，库使用它们的时候常常会撰写关于它们的文档来解释为什么这样做。你对你所使用的库的基础设计理念越熟悉，你就越不可能在这些库中被代理的使用搞得狼狈不堪。

当文档不够用的时候，头文件可以弥补空缺。很少有源码封装一个完全的代理类。它们常常从一些客户调用者期望调用的函数返回，所有函数签名常常可以表征它们的存在。这里是 `std::vector<bool>::operator[]` 的例子：

```
namespace std {                         // from C++ Standards
    template <class Allocator>
    class vector<bool, Allocator> {
        public:
        …
        class reference { … };
        reference operator[](size_type n);
        …
    };
}
```

假设你知道对 `std::vector<T>` 的 `operator[]` 常常返回一个 `T&` ，在这个例子中的这种非常规的 `operator[]` 的返回类型一般就表征了代理类的使用。在你正在使用的这些接口之上加以关注常常可以发现代理类的存在。

在实践上，很多的开发者只会在尝试修复一些奇怪的编译问题或者是调试一些错误的单元测试结果中发现代理类的使用。不管你是如何发现它们，一旦 `auto` 被决定作为推导代理类的类型而不是它被代理的类型，它就不需要涉及到关于 `auto` ， `auto` 自己本身没有问题。问题在于 `auto` 推导的类型不是所想让它推导出来的类型。解决方案就是强制一个不同的类型推导。我把这种方法叫做显式的类型初始化原则。

显式的类型初始化原则涉及到使用 `auto` 声明一个变量，但是转换初始化表达式到 `auto` 想要的类型。下面就是一个强制 `highPriority` 类型是 `bool` 的例子：

```
auto highPriority = static_cast<bool>(features(w)[5]);
```

这里， `features(w)[5]` 还是返回一个 `std::vector<bool>::reference` 的对象，就和它经常的表现一样，但是强制类型转换改变了表达式的类型成为 `bool` ，然后 `auto` 才推导其作为 `highPriority` 的类型。在运行的时候，从 `std::vector<bool>::operator[]` 返回的 `std::vector<bool>::reference` 对象支持执行转换到 `bool` 的行为，作为转换的一部分，从 `features` 返回的任然存活的指向 `std::vector<bool>` 的指针被间接引用。这样就在运行的开始避免了未定义行为。索引5然后放置在bits指针的偏移上，然后暴露的 `bool` 就作为 `highPriority` 的初始化数值。

针对于 `Matrix` 的例子，显示的类型初始化原则可能会看起来是这样的：

```
auto sum = static_cast<Matrix>(m1 + m2 + m3 + m4);
```

关于这个原则下面的程序并不禁止初始化但是要排除代理类类型。强调你要谨慎地创建一个类型的变量，它和从初始化表达式生成的类型是不同的也是有帮助意义的。举一个例子，假设你有一个函数去计算一些方差：

```
double calcEpsilon();               // 返回方差
```

`calcEpsilon`明确的返回一个`double`，但是假设你知道你的程序，`float`的精度就够了的时候，而且你要关注`double`和`float`的长度的区别。你可以声明一个`float`变量去存储`calcEpsilon`的结果：

```
float ep = calcEpsilon();           // 隐式转换double到float
```

但是这个会很难表明“我故意减小函数返回值的精度”，一个使用显式的类型初始化原则是这样做的：

```
auto ep = static_cast<float>(calcEpsilon());
```

[illegible]7:[illegible]()[illegible]{}[illegible]

[illegible]C++11[illegible]

```
int x(0);     //[illegible]()[illegible]
int y = 0;     //[illegible]=[illegible]
int z{0};     //[illegible]{}[illegible]
```

[illegible]

```
int z = {0};     //[illegible]{}[illegible]=[illegible]
```

[illegible]C++[illegible]C++[illegible]int[illegible]

```
Widget w1;     //[illegible]
Widget w2 = w1;     //[illegible]
w1 = w2;     //[illegible]=[illegible]
```

[illegible]C++98[illegible]

[illegible]STL[illegible](1,3,5)[illegible]

[illegible]C++11[illegible]:

```
std::vector<int> v{1,3,5};//v鍖j鍞'鍞1,3,5
```

鍞鍞鍞鍞Χ鍞鍞鍞鍞鍞鍞Ï鍞Ó鍞鍞鍞鍞ĺ鍞Ц鍞鍞鍞鍞'鍞鍞Ĭ鍞鍞¨鍞C++11鍞e鍞ʜ鍞鍞鍞鍞鍞□鍞 ♀ 鍞 ·鍞鍞鍞鍞鍞鍞"="ʜ鍞鍞鍞'鍞鍞鍞鍞鍞鍞鍞鍞鍞Ц鍞У鍞

```
class Widget{
   ...
   private:
    int x{0};    //鍞ȷ鍞xĭ鍞Ï0
    int y = 0;    //鍞ȷ
    int z(0);    //鍞鍞鍞
}
```

鍞ʜ鍞鍞鍞鍞鍞ó鍞鍞鍞鍞鍞(std:atomics-
鍞o鍞鍞40鍞)鍞鍞鍞'鍞ô鍞鍞鍞ź鍞鍞鍞ц鍞'鍞鍞鍞鍞鍞鍞"="鍞鍞鍞鍞:

```
std::atomic<int> ai1{0};    //鍞ȷ
std::atomic<int> ai2(0);    //鍞ȷ
std::atomic<int> ai3 = 0;    //鍞鍞鍞
```

鍞鍞鍞 ˙ 鍞鍞鍞鍞鍞鍞鍞鍞鍞鍞ц鍞'鍞鍞鍞鍞Ï鍞鍞鍞鍞鍞鍞鍞鍞ÏC++鍞鍞'
鍞鍞鍞鍞鍞鍞鍞鍞鍞鍞

Widget w2(); //�dz������壬
������ʰ������Widget������w2����

```
������������'ô�����ŵ��ᵉ����ó������'ô�����Ĭ©C��컰������������::
```

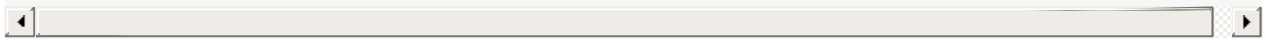
Widget w3{}; //����Widget□ı��컰��

```
'ô�����Ž��г'�����к��h���ĵ¸ó������▫�������ь燚��������▫ý㆒㆒㆒
```

��′�����ʒ��ƶ��Ĺ���Χ□��ï′��std::initializer_list���ǻ��:

```
class Widget{
public:
    Widget(int i,bool b);                //′��
    Widget(int i,double d);                //′��
    Widget(std::initializer_list<long double> il);    //′��

    operator float() const;                //ת����float��
    ...
};
Widget w5(w4);                          //'�����ţ����ÿ�������ǻ��
Widget w6(w4);                          //'ô����ţ�����std::initializer_list������ ��ǻ��
,w4��ת����float��Ȣ����ת����long double
Widget w7(std::move(w4));                //'�����ţ�����move���ǻ��
Widget w8{std::move(w4)};                //'ô����ţ�����std::initializer_list���ǻ�����
�w6 ��h��
```



������������std::initializer_lists�����Ĺ��ǻ
����Ըø҂���ʰ�����ĵ����Dz����ġ�����

```
class Widget{
public:
   Widget(int i,bool b);            //����
   Widget(int i,double d);            //����

   Widget(std::initializer_list<bool> il);    // ��������bool

   ...
};
Widget w{10,5.0};                //������ç��������ի��ת��
```

������| ������������ǰ��������ǻ
��(ȷ̣�������Dz������ő̀ƥ����)��������ɕ����std::initializer_list
������Ĺ��ǻ������������������ǻ
����Ç��int(10)��double(5.0)ת����bool𐐀������ת�������������
�������իu(bool𐐀Ͳ□□□int��double����)����իu������
□
ڑ������ڑ���ת𐐀′���DZ���·ġ̀ƥ̀ƥ̀ƥ̀ƥ̀ƥ̀ƥ̀ƥ÷Ƿ��ġ̀
��������벧□����
..̣e��□𐐀′ô�����ų𐐀′���IJ���ת����std::initializer_list��ʰ𐐀 ����
��Ż̀ƥ̀ƥ̀ƥ̀ƥ̀ƥ̀ƥ̀ƥ

```
class Widget{
public:
    Widget(int i,bool b);            //��
    Widget(int i,double d);            //��

    //std::initalizer_list ��������std::string
    Widget(std::initializer_list<std::string> il);//û����'ת��
    ...
    };

    Widget w1(10,true);                //'�����ų�'��������õ�h�����ǻ��
    Widget w2{10,true};                //'ô����ų�'��������õ�h�����ǻ��
    Widget w3(10,5.0);                //'�����ų�'��������õĵ�����ǻ��
    Widget w4{10,5.0};                //'ô����ų�'��������õĵ�����ǻ��
```

����������Ω̃ω̃ϰ̃ϰ̃ϰ

```
std::vector<int> v1(10,20)            //'初始化std::initializer_list列表，而不是10个20
std::vector<int> v2{10,20}            //'初始化std::initializer_list列表，而不是2个10和20
```

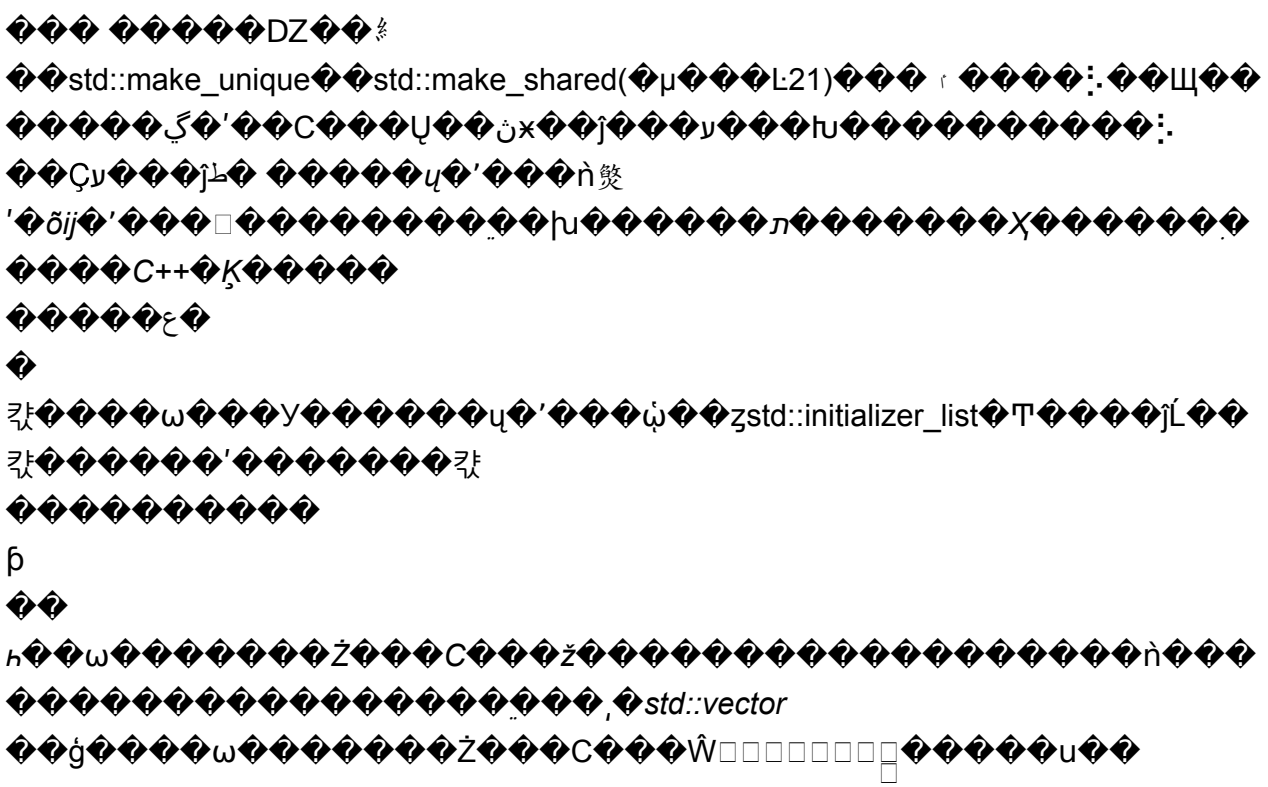
```
template<typename T,                 //
    typename... Ts>                 //'
void doSomeWork(TS&&... params)
{
   create local T objcet from params...
   ...
}
```



```
   T localObject(std::forward<Ts>(params)...);        //'
   T localObject{std::forward<Ts>(params)...};        //'
```



```
  std::vector<int> v;
  ...
  doSomeWork<std::vector<int>>(10,20);
```



条款五：优先使用 nullptr 而不是 0 或者 NULL

0 字面上是一个 int 类型，而不是指针，这是显而易见的。 C++ 扫描到一个 0 ，但是发现在上下文中仅有一个指针用到了它，编译器将勉强将 0 解释为空指针，但是这仅仅是一个应变之策。 C++ 最初始的原则是 0 是 int 而非指针。

经验上讲，同样的情况对 NULL 也是存在的。对 NULL 而言，仍有一些细节上的不确定性，因为赋予 NULL 一个除了 int （即 long ）以外的整数类型是被允许的。这不常见，但是这真的是没有问题的，因为此处的焦点不是 NULL 的确切类型而是 0 和 NULL 都不属于指针类型。

在 C++98 中，这意味着重载指针和整数类型的函数的行为会令人吃惊。传递 0 或者 NULL 作为参数给重载函数永远不会调用指针重载的那个函数：

```
void f(int);                                    // 函数f的三个重载
void f(bool);
void f(void*);

f(0);                                           // 调用 f(int)，而非f(void*)

f(NULL);                                        // 可能无法编译，但是调用f(int)
                                                // 不可能调用 f(void*)
```

f(NULL) 行为的不确定性的确反映了在实现 NULL 的类型上存在的自由发挥空间。如果 NULL 被定为 0L （即 0 作为一个 long 整形），函数的调用是有歧义的，因为 long 转化为 int ， long 转化为 bool ， 0L 转换为 void* 都被认为是同样可行的。关于这个函数调用有意思的事情是在源代码的字面意思（使用 NULL 调用 f ， NULL 应该是个空指针）和它的真实意义（一个整数在调用 f ， NULL 不是空指针）存在着冲突。这种违背直觉的行为正是 C++98 程序员不被允许重载指针和整数类型的原因。这个原则对于 C++11 依然有效，因为尽管有本条款的力荐，仍然还有一些开发者继续使用 0 和 NULL ，虽然 nullptr 是一个更好的选择。

nullptr 的优势是它不再是一个整数类型。诚实的讲，它也不是一个指针类型，但是你可以把它想象成一个可以指向任意类型的指针。 nullptr 的类型实际上是 std::nullptr_t ， std::nullptr_t 定义为 nullptr 的类型，这是一个完美的循环定义。 std::nullptr_t 可以隐式的转换为所有的原始的指针类型，这使得 nullptr 表现的像可以指向任意类型的指针。

使用 nullptr 作为参数去调用重载函数 f 将会调用 f(void*) 重载体，因为 nullptr 不能被视为整数类型的：

```
f(nullptr);                                    //调用f(void*)重载体
```

使用 `nullptr` 而不是 `0` 或者 `NULL` ，可以避免重载解析上的令人吃惊行为，但是它的优势不仅限于此。它可以提高代码的清晰度，尤其是牵扯到 `auto` 类型变量的时候。例如，你在一个代码库中遇到下面代码：

```
auto result = findRecord( /* arguments */);

if(result == 0){
    ...
}
```

如果你不能轻松地的看出 `findRecord` 返回的是什么，要知道 `result` 是一个指针还是整数类型并不是很简单的。毕竟， `0` （被用来测试 `result` 的）即可以当做指针也可以当做整数类型。另一方面，你如果看到下面的代码：

```
auto result = findRecord( /* arguments */);

if(reuslt == nullptr){
    ...
}
```

明显就没有歧义了： `result` 一定是个指针类型。

当模板进入我们考虑的范围， `nullptr` 的光芒则显得更加耀眼了。假想你有一些函数，只有当对应的互斥量被锁定的时候，这些函数才可以被调用。每个函数的参数是不同类型的指针：

```
int    f1(std::shared_ptr<Widget> spw);        // 只有对应的
double f2(std::unique_ptr<Widget> upw);        // 互斥量被锁定
bool   f3(Widget* pw);                         // 才会调用这些函数
```

想传递空指针给这些函数的调用看上去像这样：

```
std::mutex f1m, f2m, f3m;                          // 对应于f1, f2和f3的互斥量

using MuxGuard =                                   // C++11 版typedef；参加条款9
  std::lock_guard<std::mutex>;
...
{
  MuxGuard g(f1m);                                 // 为f1锁定互斥量
  auto result = f1(0);                             // 将0当做空指针作为参数传给f1
}                                                  // 解锁互斥量

...

{
  MuxGuard g(f2m);                                 // 为f2锁定互斥量
  auto result = f2(NULL);                          // 将NULL当做空指针作为参数传给f2
}                                                  // 解锁互斥量

...

{
  MuxGuard g(f3m);                                 // 为f3锁定互斥量
  auto result = f3(nullptr);                       // 将nullptr当做空指针作为参数传给f3
}                                                  // 解锁互斥量
```

在前两个函数调用中没有使用`nullptr`是令人沮丧的，但是上面的代码是可以工作的，这才是最重要的。然而，代码中的重复模式——锁定互斥量，调用函数，解锁互斥量——才是更令人沮丧和反感的。避免这种重复风格的代码正是模板的设计初衷，因此，让我们使用模板化上面的模式：

```
template<typename FuncType,
         typename MuxType,
         typename PtrType>
auto lockAndCall(FuncType func,
                 MuxType& mutex,
                 PtrType ptr) -> decltype(func(ptr))
{
  MuxGuard g(mutex);
  return func(ptr);
}
```

如果这个函数的返回值类型（`auto ...->decltype(func(ptr))`）让你挠头不已，你应该到条款3寻求一下帮助，在那里我们已经做过详细的介绍。在`C++14`中，你可以看到，返回值可以通过简单的`decltype(auto)`推导得出：

```
template<typename FuncType,
         typename MuxType,
         typename PtrType>
decltype(auto) lockAndCall(FuncType func,                    // C++14
                  MuxType& mutex,
                  PtrType ptr)
{
  MuxGuard g(mutex);
  return func(ptr);
}
```

给定 `lockAndCall` 模板（上边的任意版本），调用者可以写像下面的代码：

```
auto result1 = lockAndCall(f1, f1m, 0);                    // 错误
...
auto result2 = lockAndCall(f2, f2m, NULL);                 // 错误
...
auto result3 = lockAndCall(f3, f2m, nullptr);              // 正确
```

他们可以这样写，但是就如注释中指明的，三种情况里面的两种是无法编译通过。在第一个调用中，当把 `0` 作为参数传给 `lockAndCall` ,模板通过类型推导得知它的类型。 `0` 的类型总是 `int` ，这就是对 `lockAndCall` 的调用实例化的时候的类型。不幸的是，这意味着在 `lockAndCall` 中调用 `func` ，被传入的是 `int` ，这个 `f1` 期望接受的参数 `std::share_ptr<Widget>` 是不不兼容的。传入到 `lockAndCall` 的 `0` 尝试来表示一个空指针，但是正眞不传入的是一个普通的 `int` 类型。尝试将 `int` 作为 `std::share_ptr<Widget>` 传给 `f1` 会导致一个类型冲突错误。使用 `0` 调用 `lockAndCall` 会失败，因为在模板中，一个 `int` 类型传给一个要求参数是 `std::share_ptr<Widget>` 的函数。

对调用 `NULL` 的情况的分析基本上是一样的。当 `NULL` 传递给 `lockAndCall` 时，从参数 `ptr` 推导出的类型是整数类型，当 `ptr` ——一个 `int` 或者类 `int` 的类型——传给 `f2` ，一个类型错误将会发生，因为这个函数期待的是得到一个 `std::unique_ptr<Widget>` 类型的参数。

相反，使用 `nullptr` 是没有问题的。当 `nullptr` 传递给 `lockAndCall` ， `ptr` 的类型被推导为 `std::nullptr_t` 。当 `ptr` 被传递给 `f3` ，有一个由 `std::nullptr_t` 到 `Widget*` 的隐形转换，因为 `std::nullptr_t` 可以隐式转换为任何类型的指针。

眞正的原因是，对于 `0` 和 `NULL` ，模板类型推导出了错误的类型（他们的眞正类型，而不是它们作为空指针而体现出的退化的内涵），这是在需要用到空指针时使用 `nullptr` 而非 `0` 或者 `NULL` 最引人注目的原因。使用 `nullptr` ，模板不会造成额外的困扰。另外结合 `nullptr` 在重载中不会导致像 `0` 和 `NULL` 那样的诡异行为的事实，胜负已定。当你需要用到空指针时，使用 `nullptr` 而不是 `0` 或者 `NULL` 。

要记住的东西
相较于 `0` 和 `NULL` ，优先使用 `nullptr`
避免整数类型和指针类型之间的重载

条款9：优先使用声明别名而不是 typedef

我有信心说，大家都同意使用 STL 容器是个好的想法，并且我希望，条款18可以说服你使用 std::unique_ptr 也是个好想法，但是我想绝对我们中间没有人喜欢写像这样 std::unique_ptr<std::unordered_map<std::string, std::string>> 的代码多于一次。这仅仅是考虑到这样的代码会增加得上“键盘手”的风险。

为了避免这样的医疗悲剧，推荐使用一个 typedef：

```
typedef
  std::unique_ptr<std::unordered_map<std::string, std::string>>
  UPtrMapSS;
```

但是 typedef 家族是有如此浓厚的 C++98 气息。他们的确可以在 C++11 下工作，但是 C++11 也提供了声明别名（ alias declarations ）：

```
using UptrMapSS =
  std::unique_ptr<std::unordered_map<std::string, std::string>>;
```

考虑到 typedef 和声明别名具有完全一样的意义，推荐其中一个而排斥另外一个的坚实技术原因是容易令人质疑的。这样的质疑是合理的。

技术原因当然存在，但是在我提到之前。我想说的是，很多人发现使用声明别名可以使涉及到函数指针的类型的声明变得容易理解：

```
// FP等价于一个函数指针，这个函数的参数是一个int类型和
// std::string常量类型，没有返回值
typedef void (*FP)(int, const std::string&);      // typedef

// 同上
using FP = void (*)(int, const std::string&);     // 声明别名
```

当然，上面任何形式都不是特别让人容易下咽，并且很少有人会花费大量的时间在一个函数指针类型的标识符上，所以这很难当做选择声明别名而不是 typedef 的不可抗拒的原因。

但是，一个不可抗拒的原因是真实存在的：模板。尤其是声明别名有可能是模板化的（这种情况下，它们被称为模板别名（ alias template ）），然而 typedef 这是只能说句“臣妾做不到”。模板别名给 C++11 程序员提供了一个明确的机制来表达在 C++98 中需要黑客式的将 typedef 嵌入在模板化的 struct 中才能完成的东西。举个栗子，给一个使用个性化的分配器 MyAlloc 的链接表定义一个标识符。使用别名模板，这就是小菜一碟：

```
template<typname T>                              // MyAllocList<T>
using MyAllocList = std::list<T, MyAlloc<T>>;    // 等同于
                                                 // std::list<T,
                                                 //   MyAlloc<T>>

MyAllocList<Widget> lw;                          // 终端代码
```

使用 `typedef` ，你不得不从草稿图开始去做一个蛋糕：

```
template<typename T>                             // MyAllocList<T>::type
struct MyAllocList {                             // 等同于
  typedef std::list<T, MyAlloc<T>> type;         // std::list<T,
};                                               // MyAlloc<T>>

MyAllocList<Widget>::type lw;                    // 终端代码
```

如果你想在一个模板中使用 `typedef` 来完成创建一个节点类型可以被模板参数指定的链接表的任务，你必须在 `typedef` 名称之前使用 `typename` ：

```
template<typename T>                             // Widget<T> 包含
class Widget{                                    // 一个 MyAloocList<T>
private:                                         // 作为一个数据成员
  typename MyAllocList<T>::type list;
  ...
};
```

此处， `MyAllocList<T>::type` 表示一个依赖于模板类型参数 `T` 的类型，因此 `MyAllocList<T>::type` 是一个依赖类型（ `dependent type` ）， `C++` 中许多令人喜爱的原则中的一个就是在依赖类型的名称之前必须冠以 `typename` 。

如果 `MyAllocList` 被定义为一个声明别名，就不需要使用 `typename` （就像笨重的 `::type` 后缀）：

```
template<typname T>
using MyAllocList = std::list<T, MyAlloc<T>>;   // 和以前一样

template<typename T>
class Widget {
private:
  MyAllocList<T> list;                          // 没有typename
  ...                                           // 没有::type
};
```

对你来说，`MyAllocList<T>`（使用模板别名）看上去依赖于模板参数`T`，正如`MyAllocList<T>::type`（使用内嵌的`typdef`）一样，但是你不是编译器。当编译器处理`Widget`遇到`MyAllocList<T>`（使用模板别名），编译器知道`MyAllocList<T>`是一个类型名称，因为`MyAllocList`是一个模板别名：它必须是一个类型。`MyAllocList<T>`因此是一个非依赖类型（`non-dependent type`），指定符`typename`是不需要和不允许的。

另一方面，当编译器在`Widget`模板中遇到`MyAllocList<T>`（使用内嵌的`typename`）时，编译器并不知道它是一个类型名，因为有可能存在一个特殊化的`MyAllocList`，只是编译器还没有扫描到，在这个特殊化的`MyAllocList`中`MyAllocList<T>::type`表示的并不是一个类型。这听上去挺疯狂的，但是不要因为这种可能性而怪罪于编译器。是人类有可能会写出这样的代码。

例如，一些被误导的鬼魂可能会杂糅出像这样代码：

```
class Wine {...};

template<>                                  // 当T时Wine时
class MyAllocList<Wine>{                    // MyAllocList 是特殊化的
private:
  enum class WineType                       // 关于枚举类参考条款10
  { White, Red, Rose };

  WineType type;                            // 在这个类中，type是个数据成员
  ...
};
```

正如你看到的，`MyAllocList<Wine>::type`并不是指一个类型。如果`Widget`被使用`Wine`初始化，`Widget`模板中的`MyAllocList<T>::type`指的是一个数据成员，而不是一个类型。在`Wedget`模板中，`MyAllocList<T>::type`是否指的是一个类型忠实地依赖于传入的`T`是什么，这也是编译器坚持要求你在类型前面冠以`typename`的原因。

如果你曾经做过模板元编程（`TMP`），你会强烈地额反对使用模板类型参数并在此基础上修改为其他类型的必要性。例如，给定一个类型`T`，你有可能想剥夺`T`所包含的所有的`const`或引用的修饰符，即你想将`const std::string&`变成`std::string`。你也有可能想给一个类型加上`const`或者将它变成一个左值引用，也就是将`Widget`变成`const Widget`或者`Widget&`。（如果你没有做过`TMP`,这太糟糕了，因为如果你想成为一个眞正牛叉的`C++`程序员，你至少需要对`C++`这方面的基本概念足够熟悉。你可以同时看一些TMP的例子，包括我上面提到的类型转换，还有条款23和条款27。）

`C++11`给你提供了工具来完成这类转换的工作，表现的形式是`type traits`,它是`<type_traits>`中的一个模板的分类工具。在这个头文件中有数十个类型特征，但是并不是都可以提供类型转换，不提供转换的也提供了意料之中的接口。给定一个你想竟选类型转换的类型`T`，得到的类型是`std::transformation<T>::type`。例如：

```
std::remove_const<T>::type                // 从 const T 得到 T
std::remove_reference<T>::type            // 从 T& 或 T&& 得到 T
std::add_lvalue_reference<T>::type        // 从 T 得到 T&
```

注释仅仅总结了这些转换干了什么，因此不需要太咬文嚼字。在一个项目中使用它们之前，我知道你会参考准确的技术规范。

无论如何，我在这里不是只想给你大致介绍一下类型特征。反而是因为注意到，类型转换总是以 `::type` 作为每次使用的结尾。当你对一个模板中的类型参数（你在实际代码中会经常用到）使用它们时，你必须在每次使用前冠以 `typename` 。这其中的原因是 `C++11` 的类型特征是通过内嵌 `typedef` 到一个模板化的 `struct` 来实现的。就是这样的，他们就是通过使用类型同义技术来实现的，就是我一直在说服你远不如模板别名的那个技术。

这是一个历史遗留问题，但是我们略过不表（我打赌，这个原因真的很枯燥）。因为标准委员会姗姗来迟地意识到模板别名是一个更好的方式，对于 `C++11` 的类型转换，委员会使这些模板也成为 `C++14` 的一部分。别名有一个统一的形式：对于 `C++11` 中的每个类型转换 `std::transformation<T>::type` ，有一个对应的 `C++14` 的模板别名 `std::transformation_t` 。用例子来说明我的意思：

```
std::remove_const<T>::type                // C++11: const T -> T
std::remove_const_t<T>                    // 等价的C++14

std::remove_reference<T>::type            // C++11: T&/T&& -> T
std::remove_reference_t<T>                // 等价的C++14

std::add_lvalue_reference<T>::type        // C++11: T -> T&
std::add_lvalue_reference_t<T>            // 等价的C++14
```

`C++11` 的结构在 `C++14` 中依然有效，但是我不知道你还有什么理由再用他们。即便你不熟悉 `C++14` ，自己写一个模板别名也是小儿科。仅仅 `C++11` 的语言特性被要求，孩子们甚至都可以模拟一个模式，对吗？如果你碰巧有一份 `C++14` 标准的电子拷贝，这依然很简单，因为需要做的即使一些复制和粘贴操作。在这里，我给你开个头：

```
template<class T>
using remove_const_t = typename remove_const<T>::type;

template<class T>
using remove_reference_t = typename remove_reference<T>::type;

template<class T>
using add_lvalue_reference_t =
  typename add_lvalue_reference<T>::type;
```

看到没有？不能再简单了。

要记住的东西
`typedef` 不支持模板化，但是别名声明支持
模板别名避免了 `::type` 后缀，在模板中， `typedef` 还经常要求使用 `typename` 前缀
`C++14` 为 `C++11` 中的类型特征转换提供了模板别名

条款10：优先使用作用域限制的 enmus 而不是无作用域的 enum

一般而言，在花括号里面声明的变量名会限制在括号外的可见性。但是这对于 C++98 风格的 enums 中的枚举元素并不成立。枚举元素和包含它的枚举类型同属一个作用域空间，这意味着在这个作用域中不能再有同样名字的定义：

```
enum Color { black, white, red};                // black, white, red 和
                                                // Color  同属一个定义域

auto white = false;                             // 错误！因为 white
                                                // 在这个定义域已经被声明过
```

事实就是枚举元素泄露到包含它的枚举类型所在的作用域中，对于这种类型的 enum 官方称作无作用域的（ unscoped ）。在 C++11 中对应的使用作用域的enums（ scoped enums ）不会造成这种泄露：

```
enum class Color { black, white, red};          // black, white, red
                                                // 作用域为 Color

auto white = false;                             // fine, 在这个作用域内
                                                // 没有其他的 "white"

Color c = white;                                // 错误！在这个定义域中
                                                // 没有叫"white"的枚举元素

Color c = Color::white;                         // fine

auto c = Color::white;                          // 同样没有问题（和条款5
                                                // 的建议项吻合）
```

因为限制作用域的 enum 是通过"enum class"来声明的，它们有时被称作枚举类（ enum class ）。

限制作用域的 enum 可以减少命名空间的污染，这足以是我们更偏爱它们而不是不带限制作用域的表亲们。除此之外，限制作用域的 enums 还有一个令人不可抗拒的优势：它们的枚举元素可以是更丰富的类型。无作用域的 enum 会将枚举元素隐式的转换为整数类型（从整数出发，还可以转换为浮点类型）。因此像下面这种语义上荒诞的情况是完全合法的：

```
enum Color { black, white, red };                  // 无限制作用域的enum

std::vector<std::size_t>                           // 返回x的质因子的函数
  primeFactors(std::size_t x);

Color c = red;
...

if (c < 14.5 ){                                    // 将Color和double类型比较！

    auto  factors =                                // 计算一个Color变量的质因子
      primeFactors(c);
}
```

在 "enum" 后增加一个 "class" ，就可以将一个无作用域的 enum 转换为一个有作用域的 enum ，变成一个有作用域的 enum 之后，事情就变得不一样了。在有作用域的 enum 中不存在从枚举元素到其他类型的隐式转换：

```
enum class Color { black, white, red };            // 有作用域的enum

Color c = Color::red;                              // 和前面一样，但是
...                                                // 加上一个作用域限定符

if (c < 14.5){                                     // 出错！不能将Color类型
                                                   // 和double类型比较
  auto factors =                                   // 出错！不能将Color类型传递给
    primeFactors(c);                               // 参数类型为std::size_t的函数
...
 }
```

如果你就是想将 Color 类型转换为一个其他类型，使用类型强制转换（ cast ）可以满足你这种变态的需求:

```
    if(static_cast<double>(c) < 14.5) {            // 怪异但是有效的代码

    auto factors =                                 // 感觉不可靠
      primeFactors(static_cast<std::size_t(c));    // 但是可以编译
    ...
}
```

相较于无定义域的 enum ，有定义域的 enum 也许还有第三个优势，因为有定义域的 enum 可以被提前声明的，即可以不指定枚举元素而进行声明:

```
enum Color;                                        // 出错！

enum class Color;                                  // 没有问题
```

这是一个误导。在 C++11 中，没有定义域的 enum 也有可能被提前声明，但是需要一点额外的工作。这个工作时基于这样的事实： C++ 中的枚举类型都有一个被编译器决定的潜在的类型。对于一个无定义域的枚举类型像 Color ，

```
enum Color {black, white, red };
```

编译器有可能选择 char 作为潜在的类型，因为仅仅有三个值需要表达。然而一些枚举类型有很大的取值的跨度，如下：

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
            };
```

这里需要表达的值范围从 0 到 0xFFFFFFFF 。除非是在一个不寻常的机器上（在这台机器上， char 类型至少有 32 个 bit ），编译器一定会选择一个取值范围比 char 大的整数类型来表示 Status 的类型。

为了更高效的利用内存，编译器通常想为枚举类型选择可以充分表示枚举元素的取值范围但又占用内存最小的潜在类型。在某些情况下，为了代码速度的优化，可以回牺牲内存大小，在那种情况下，编译器可能不会选择占用内存最小的可允许的潜在类型，但是编译器依然希望能过优化内存存储的大小。为了使这种功能可以实现， C++98 仅仅支持枚举类型的定义（所有枚举元素被列出来），而枚举类型的声明是不被允许的。这样可以保证在枚举类型被用到之前，编译器已经给每个枚举类型选择了潜在类型。

不能事先声明枚举类型有几个不足。最引人注意的就是会增加编译依赖性。再次看看 Status 这个枚举类型：

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              indeterminate = 0xFFFFFFFF
            };
```

这个枚举体可能会在整个系统中都会被使用到，因此被包含在系统每部分都依赖的一个头文件当中。如果一个新的状态需要被引入：

```
enum Status { good = 0,
              failed = 1,
              incomplete = 100,
              corrupt = 200,
              audited = 500,
              indeterminate = 0xFFFFFFFF
            };
```

就算一个子系统——甚至只有一个函数！——用到这个新的枚举元素，有可能导致整个系统的代码需要被重新编译。这种事情是人们憎恨的。在 C++11 中，这种情况被消除了。例如，这里有一个完美的有效的有作用域的 enum 的声明，还有一个函数将它作为参数：

```
enum class Status;                  // 前置声明

void continueProcessing(Status s);  // 使用前置声明的枚举体
```

如果 Status 的定义被修改，包含这个声明的头文件不需要重新编译。更进一步，如果 Status 被修改（即，增加 audited 枚举元素），但是 continueProcessing 的行为不受影响（因为 continueProcessing 没有使用 audited ），continueProcessing 的实现也不需要重新编译。

但是如果编译器需要在枚举体之前知道它的大小，C++11 的枚举体怎么做到可以前置声明，而 C++98 的枚举体无法实现？原因是简单的，对于有作用域的枚举体的潜在类型是已知的，对于没有作用域的枚举体，你可以指定它。

对有作用域的枚举体，默认的潜在的类型是 int：

```
enum class Status;                  // 潜在类型是int
```

如果默认的类型不适用于你，你可重载它：

```
enum class Status: std::uint32_t;   // Status潜在类型是
                                    // std::uint32_t
                                    // （来自<cstdint>）
```

无论哪种形式，编译器都知道有作用域的枚举体中的枚举元素的大小。

为了给没有作用域的枚举体指定潜在类型，你需要做相同的事情，结果可能是前置声明：

```
enum Color: std::uint8_t;           // 没有定义域的枚举体
                                    // 的前置声明，潜在类型是
                                    // std::uint8_t
```

潜在类型的指定也可以放在枚举体的定义处：

```
enum class Status: std::uint32_t{ good = 0,
                                  failed = 1,
                                  incomplete = 100,
                                  corrupt = 200,
                                  audited = 500,
                                  indeterminate = 0xFFFFFFFF
                                };
```

从有定义域的枚举体可以避免命名空间污染和不易受无意义的隐式类型转换影响的角度看，你听到至少在一种情形下没有定义域的枚举体是有用的可能会感到惊讶。这种情况发生在引用 `C++11` 的 `std::tuples` 中的某个域时。例如，假设我们有一个元组，元组中保存着姓名，电子邮件地址，和用户在社交网站的影响力数值：

```
using UserInfo =                // 别名，参见条款9
  std::tuple<std::string,       // 姓名
             std::string,         // 电子邮件
             std::size_t> ;       // 影响力
```

尽管注释已经说明元组的每部分代表什么意思，但是当你遇到像下面这样的源代码时，可能注释没有什么用：

```
UserInfo uInfo;                 // 元组类型的一个对象
...

auto val = std::get<1>(uInfo);  // 得到第一个域的值
```

作为一个程序员，你有很多事要做。你真的想去记住元组的第一个域对应的是用户的电子邮件地址？我不这么认为。使用一个没有定义域的枚举体来把名字和域的编号联系在一起来避免去死记这些东西：

```
enum UserInfoFields {uiName, uiEmail, uiReputation };

UserInfo uInfo;                         // 和前面一样
...

auto val = std::get<uiEmail>(uInfo);    // 得到电子邮件域的值
```

上面代码正常工作的原因是 `UserInfoFields` 到 `std::get()` 要求的 `std::size_t` 的隐式类型转换。

如果使用有作用域的枚举体的代码就显得十分冗余：

```
enum class UserInfoFields { uiName, uiEmail, uiReputaion };

UserInfo uInfo;                         // 和前面一样
...

auto val =
  std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>(uInfo);
```

写一个以枚举元素为参数返回对应的 `std::size_t` 的类型的值可以减少这种冗余性。`std::get` 是一个模板，你提供的值是一个模板参数（注意用的是尖括号，不是圆括号），因此负责将枚举元素转化为 `std::size_t` 的这个函数必须在编译阶段就确定它的结果。就像条款15解释的，这意味着它必须是一个 `constexpr` 函数。

实际上，它必须是一个 `constexpr` 函数模板，因为它应该对任何类型的枚举体有效。如果我们打算实现这种一般化，我们需要一般化返回值类型。不是返回 `std::size_t` ，我们需要返回枚举体的潜在类型。通过 `std::underlying_type` 类型转换来实现（关于类型转换的信息，参见条款9）。最后需要将这个函数声明为 `noexcept` （参见条款14），因为我们知道它永远不会触发异常。结果就是这个函数模板可以接受任何的枚举元素，返回这个元素的在编译阶段的常数值：

```
template<typename E>
constexpr typename std::underlying_type<E>::type
  toUType(E enumerator) noexcept
{
  return
    static_cast<typename
                std::underlying_type<E>::type>(enumerator);
}
```

在 `C++14` 中， `toUType` 可以通过将 `std::underlying_type<E>::type` 替代为 `std::underlying_type_t` （参见条款9）：

```
template<typename E>                            // C++14
constexpr std::underlying_type_t<E>
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

更加优雅的 `auto` 返回值类型（参见条款3）在 `C++14` 中也是有效的：

```
template<typename E>
constexpr auto
  toUType(E enumerator) noexcept
{
  return static_cast<std::underlying_type_t<E>>(enumerator);
}
```

无论写哪种形式，`toUType` 允许我们想下面一样访问一个元组的某个域：

```
auto val = std::get<toUType(UserInfoFields::uiEmail)>(uInfo);
```

这样依然比使用没有定义域的枚举体要复杂，但是它可以避免命名空间污染和不易引起注意的枚举元素的的类型转换。很多时候，你可能会决定多敲击一些额外的键盘来避免陷入一个上古时代的枚举体的技术陷阱中。

要记住的东西
`C++98` 风格的 `enum` 是没有作用域的 `enum`
有作用域的枚举体的枚举元素仅仅对枚举体内部可见。只能通过类型转换（ `cast` ）转换为其他类型
有作用域和没有作用域的 `enum` 都支持指定潜在类型。有作用域的 `enum` 的默认潜在类型是 `int` 。没有作用域的 `enum` 没有默认的潜在类型。
有作用域的 `enum` 总是可以前置声明的。没有作用域的 `enum` 只有当指定潜在类型时才可以前置声明。

条款11：优先使用delete关键字删除函数而不是private却又不实现的函数

如果你要给其他开发者提供代码，并且还不想让他们调用特定的函数，你只需要不声明这个函数就可以了。没有函数声明，没有就没有函数可以调用。这是没有问题的。但是有时候 C++ 为你声明了一些函数，如果你想阻止客户调用这些函数，就不是那么容易的事了。

这种情况只有对“特殊的成员函数”才会出现，即这个成员函数是需要的时候 C++ 自动生成的。条款17详细地讨论了这种函数，但是在这里，我们仅仅考虑复制构造函数和复制赋值操作子。这一节致力于 C++98 中的一般情况，这些情况可能在 C++11 中已经不复存在。
在 C++98 中，如果你想压制一个成员函数的使用，这个成员函数通常是复制构造函数，赋值操作子，或者它们两者都包括。

在 C++98 中阻止这类函数被使用的方法是将这些函数声明为 private ，并且不定义它们。例如，在 C++ 标准库中， IO 流的基础是类模板 basic_ios 。所有的输入流和输出流都继承（有可能间接地）与这个类。拷贝输入和输出流是不被期望的，因为不知道应该采取何种行为。比如，一个 istream 对象，表示一系列输入数值的流，一些已经被读入内存，有些可能后续被读入。如果一个输入流被复制，是不是应该将已经读入的数据和将来要读入的数据都复制一下呢？处理这类问题最简单的方法是定义这类问题不存在， IO 流的复制就是这么做的。

为了使 istream 和 ostream 类不能被复制， basic_ios 在 C++98 中是如下定义的（包括注释）：

```
template <class charT, class traits = char_traits<charT> >
class basic_ios :public ios_base {
public:
  ...

  private:
    basic_ios(const basic_ios& );                 // 没有定义
    basic_ios& operator(const basic_ios&);        // 没有定义
};
```

将这些函数声明为私有来阻止客户调用他们。故意不定义它们是因为，如果有函数访问这些函数（通过成员函数或者友好类）在链接的时候会导致没有定义而触发的错误。

在 C++11 中，有一个更好的方法可以基本上实现同样的功能：用 = delete 标识拷贝复制函数和拷贝赋值函数为删除的函数 deleted functions 。在 C++11 中 basic_ios 被定义为：

```
template <class charT, class traits = char_traits<charT> >
class basic_ios : public ios_base {
public:
  ...
  basic_ios(const basic_ios& ) = delete;
  basic_ios& operator=(const basic_ios&) = delete;
  ...
};
```

删除的函数和声明为私有函数的区别看上去只是时尚一些，但是区别比你想象的要多。删除的函数不能通过任何方式被使用，即便是其他成员函数或者友好函数试图复制 `basic_ios` 对象的时候也会导致编译失败。这是对 `C++98` 中的行为的升级，因为在 `C++98` 中直到链接的时候才会诊断出这个错误。

方便起见，删除函数被声明为公有的，而不是私有的。这样设计的原因是，当客户端程序尝试使用一个成员函数的时候， `C++` 会在检查删除状态之前检查可访问权限。当客户端代码尝试访问一个删除的私有函数时，一些编译器仅仅会警报该函数为私有，尽管这里函数的可访问性并不本质上影响它是否可以被使用。当把私有未定义的函数改为对应的删除函数时，牢记这一点是很有意义的，因为使这个函数为公有的可以产生更易读的错误信息。

删除函数一个重要的优势是任何函数都可以是删除的，然而仅有成员函数才可以是私有的。举个例子，加入我们有个非成员函数，以一个整数位参数，然后返回这个参数是不是幸运数字：

```
bool isLucky(int number);
```

`C++` 继承于 `C` 意味着，很多其他类型被隐式的转换为 `int` 类型，但是有些调用可以编译但是没有任何意义：

```
if(isLucky('a')) ...                // a 是否是幸运数字？

if(isLucky(ture)) ...               // 返回true?

if(isLucky(3.5)) ...                // 我们是否应该在检查它是否幸运之前裁剪为3？
```

如果幸运数字一定要是一个整数，我们希望能到阻止上面那种形式的调用。

完成这个任务的一个方法是为想被排除出去的类型的重载函数声明为删除的：

```
bool isLucky(int number);           // 原本的函数

bool isLucky(char) = delete;        // 拒绝char类型

bool isLucky(bool) = delete;        // 拒绝bool类型

bool isLucky(double) = delete;      // 拒绝double和float类型
```

（对 `double` 的重载的注释写到： `double` 和 `float` 类型都讲被拒绝可能会令你感到吃惊，当时当你回想起来，如果给 `float` 一个转换为 `int` 或者 `double` 的可能性， `C++` 总是倾向于转化为 `double` 的，就不会感到奇怪了。以 `float` 类型调用 `isLucky` 总是调用对应的 `double` 重载，而不是 `int` 类型的那个重载。结果就是将 `double` 类型的重载删除将会组织 `float` 类型的调用编译。）

尽管删除函数不能被使用，但是它们仍然是你程序的一部分。因此，在重载解析的时候仍会将它们考虑进去。这也就是为什么有了上面的那些声明，对 `isLucky` 不被期望的调用会被拒绝：

```
if (isLucky('a')) ...               // 错误！调用删除函数

if (isLucky(true)) ...              // 错误！

if (isLucky(3.5f)) ...              // 错误！
```

还有一个删除函数可以完成技巧（而私有成员函数无法完成）是可以阻止那些应该被禁用的模板实现。举个例子，假设你需要使用一个内嵌指针的模板（虽然第4章建议使用智能指针而不是原始的指针）：

```
template<typename T>
void processPointer(T* ptr);
```

在指针的家族中，有两个特殊的指针。一个是 `void*` 指针，因为没有办法对它们解引用，递增或者递减它们等操作。另一个是 `char*` 指针，因为它们往往表示指向 `C` 类型的字符串，而不是指向独立字符的指针。这些特殊情况经常需要特殊处理，在 `processPointer` 模板中，假设对这些特殊的指针合适的处理方式拒绝调用。也就是说，不可能以 `void*` 或者 `char*` 为参数调用 `processPointer` 。

这是很容易强迫实现的。仅仅需要删除这些实现：

```
template<>
void processPointer<void>(void*) = delete;

template<>
void processPointer<char>(char*) = delete;
```

现在，使用 `void*` 或者 `char*` 调用 `processPointer` 是无效的，使用 `const void*` 或者 `const char*` 调用也需要是无效的，因此这些实现也需要被删除：

```
template<>
void processPointer<const void>(const void*) = delete;

template<>
void processPointer<const char>(const char*) = delete;
```

如果你想更彻底一点，你还要删除对 `const volatile void*` 和 `const volatile char*` 的重载，你就可以在其他标准的字符类型的指针 `std::wchar_t, std::char16_t` 和 `std::char32_t` 上愉快的工作了。

有趣的是，如果你在一个类内部有一个函数模板，你想通过声明它们为私有来禁止某些实现，但是你通过这种方式做不到，因为赋予一个成员函数模板的某种特殊情况下拥有不同于模板主体的访问权限是不可能。举个例子，如果 `processPointer` 是 `Widget` 内部的一个成员函数模板，你想禁止使用 `void*` 指针的调用，下面是一个 `C++98` 风格的方法，下面代码依然无法通过编译：

```
class Widget{
public:
  ...
  template<typename T>
  void processPointer(T* ptr)
  { ... }

private:
  template<>                                          // 错误！
  void processPointer<void>(void*)

};
```

这里的问题是，模板的特殊情况必须要写在命名空间的作用域内，而不是类的作用域内。这个问题对于删除函数是不存在的，因为它们不再需要一个不同的访问权限。它们可以再类的外面被声明为是被删除的（也就是在命名空间的作用域内）：

```
class Widget{
public:
  ...
  template<typename T>
  void processPointer(T* ptr)
  { ... }
  ...

};

template<>                                                  // 仍然是公用的，但是已被删除
void Widget::processPointer<void>(void*) = delete;
```

真相是，`C++98` 中声明私有函数但是不定义是想达到 `C++11` 中删除函数同样效果的尝试。作为一个模仿品，`C++98` 的方式并不如它要模仿的东西那么好。它在类的外边和内部都是是无法工作的，当它工作时，知道链接的时候可能又不工作了。所以还是坚持使用删除函数吧。

要记住的东西
优先使用删除函数而不是私有而不定义的函数
任何函数都可以被声明为删除，包括非成员函数和模板实现

条款12：使用override关键字声明覆盖的函数

C++ 中的面向对象的变成都是围绕类，继承和虚函数进行的。其中最基础的一部分就是，派生类中的虚函数会覆盖掉基类中对应的虚函数。但是令人心痛的意识到虚函数重载是如此容易搞错。这部分的语言特性甚至看上去是按照墨菲准则设计的，它不需要被遵从，但是要被膜拜。

因为覆盖“ overriding ”听上去像重载“ overloading ”，但是它们完全没有关系，我们要有一个清晰地认识，虚函数（覆盖的函数）可以通过基类的接口来调用一个派生类的函数：

```
class Base{
public:
  virtual void doWork();                          // 基类的虚函数
  ...
};

class Derived: public Base{
public:
  virtual void doWork();                          // 覆盖 Base::doWork
                                                  // ("virtual" 是可选的)
  ...
};

std::unique_ptr<Base> upb =                       // 产生一个指向派生类的基类指针
                                                  // 关于 std::make_unique 的信息参考
条款21
  std::make_unique<Derived>();

...

upb->doWork();                                    // 通过基类指针调用 doWork()，
                                                  // 派生类的对应函数别调用
```

如果要使用覆盖的函数，几个条件必须满足：

- 基类中的函数被声明为虚的。
- 基类中和派生出的函数必须是完全一样的（出了虚析构函数）。
- 基类中和派生出的函数的参数类型必须完全一样。
- 基类中和派生出的函数的常量特性必须完全一样。
- 基类中和派生出的函数的返回值类型和异常声明必须使兼容的。

以上的约束仅仅是 C++98 中要求的部分， C++11 有增加了一条：

- 函数的引用修饰符必须完全一样。成员函数的引用修饰符是很少被提及的 `C++11` 的特性，所以你之前没有听说过也不要惊奇。这些修饰符使得将这些函数只能被左值或者右值使用成为可能。成员函数不需要声明为虚就可以使用它们：

```
class Widget{
public:
  ...
  void doWork() &;                          // 只有当 *this 为左值时
                                            // 这个版本的 doWorkd()
                                            // 函数被调用

  void doWork() &&;                         // 只有当 *this 为右值
                                            // 这个版本的 doWork()
                                            // 函数被调用
};
...
Widget makeWidget();                        // 工厂函数，返回右值

Widget w;                                   // 正常的对象（左值）

...

w.doWork();                                 // 为左值调用 Widget::doWork()
                                            //（即 Widget::doWork &）

makeWidget().doWork();                      // 为右值调用 Widget::doWork()
                                            //（即 Widget::doWork &&）
```

  稍后我们会更多介绍带有引用修饰符的成员函数的情况，但是现在，我们只是简单的提到：如果一个虚函数在基类中有一个引用修饰符，派生类中对应的那个也必须要有完全一样的引用修饰符。如果不完全一样，派生类中的声明的那个函数也会存在，但是它不会覆盖基类中的任何东西。

对覆盖函数的这些要求意味着，一个小的错误会产生一个很大不同的结果。在覆盖函数中出现的错误通常还是合法的，但是它导致的结果并不是你想要的。所以当你犯了某些错误的时候，你并不能依赖于编译器对你的通知。例如，下面的代码是完全合法的，乍一看，看上去也是合理的，但是它不包含任何虚覆盖函数——没有一个派生类的函数绑定到基类的对应函数上。你能找到每种情况里面的问题所在吗？即为什么派生类中的函数没有覆盖基类中同名的函数。

```
class Base {
public:
  virtual void mf1() const;
  virtual void mf2(int x);
  virtual void mf3() &;
  void mf4() const;
};

class Derived: public Base {
 public:
   virtual void mf1();
   virtual void mf2(unsigned int x);
   virtual void mf3() &&;
   void mf4() const;
};
```

需要什么帮助吗？

- `mf1` 在 `Base` 中声明常成员函数，但是在 `Derived` 中没有
- `mf2` 在 `Base` 中以 `int` 为参数，但是在 `Derived` 中以 `unsigned int` 为参数
- `mf3` 在 `Base` 中有左值修饰符，但是在 `Derived` 中是右值修饰符
- `mf4` 没有继承 `Base` 中的虚函数

你可能会想，“在实际中，这些代码都会触发编译警告，因此我不需要过度忧虑。”也许的确是这样，但是也有可能不是这样。经过我的检查，发现在两个编译器上，上边的代码被全然接受而没有发出任何警告，在这两个编译器上所有警告是都会被输出的。（其他的编译器输出了这些问题的警告信息，但是输出的信息也不全。）

因为声明派生类的覆盖函数是如此重要，有如此容易出错，所以 `C++11` 给你提供了一种可以显式的声明一个派生类的函数是要覆盖对应的基类的函数的：声明它为 `override` 。把这个规则应用到上面的代码得到下面样子的派生类：

```
class Derived: public Base {
public:
  virtual void mf1() override;
  virtual void mf2(unsigned int x) override;
  virtual void mf3() && override;
  virtual void mf4() const override;
};
```

这当然是无法通过编译的，因为当你用这种方式写代码的时候，编译器会把覆盖函数所有的问题揭露出来。这正是你想要的，所以你应该把所有覆盖函数声明为 `override` 。

使用 `override` ，同时又能通过编译的代码如下（假设目的就是 `Derived` 类中的所有函数都要覆盖 `Base` 对应的虚函数）：

```
class Base {
public:
  virtual void mf1() const;
  virtual void mf2(int x);
  virtual void mf3() &;
  virtual void mf4() const;
};

class Derived: public Base {
public:
  virtual void mf1() const override;
  virtual void mf2(int x) override;
  virtual void mf3() & override;
  void mf4() const override;                    // 加上"virtual"也可以
                                                // 但是不是必须的
};
```

注意在这个例子中，代码能正常工作的一个基础就是声明 `mf4` 为 `Base` 类中的虚函数。绝大部分关于覆盖函数的错误发生在派生类中，但是也有可能在基类中有不正确的代码。

对于派生类中覆盖体都声明为 `override` 不仅仅可以让编译器在应该要去覆盖基类中函数而没有去覆盖的时候可以警告你。它还可以帮助你预估一下更改基类里的虚函数的标识符可能会引起的后果。如果在派生类中到处使用了 `override` ，你可以改一下基类中的虚函数的名字，看看这个举动会造成多少损害（即，有多少派生类无法通过编译），然后决定是否可以为了这个改动而承受它带来的问题。如果没有 `override` ，你会希望此处有一个无所不包的测试单元，因为，正如我们看到的，派生类中那些原本被认为要覆盖基类函数的部分，不会也不需要引发编译器的诊断信息。

条款13：优先使用const_iterator而不是iterator

`const_iterator`在STL中等价于指向`const`的指针。被指向的数值是不能被修改的。标准的做法是应该使用`const`的迭代器的地方，也就是尽可能的在没有必要修改指针所指向的内容的地方使用`const_iterator`。

这对于C++98和C++11是正确，但是在C++98中，`const_iterator`s只有部分的支持。一旦有一个这样的迭代器，创建它们并非易事，使用也会受限。举一个例子，假如你希望从`vector<int>`搜索第一次出现的1983(这一年"C++"替换"C + 类"而作为一个语言的名字)，然iterator后在搜到的位置插入数值1998(这一年第一个ISO C++标准被接受)。如果在vector中并不存在1983，插入操作的位置应该是vector的末尾。在C++98中使用`iterator`，这会非常容易：

```
std::vector<int> values;
…
std::vector<int>::iterator it =
std::find(values.begin(),values.end(), 1983);
values.insert(it, 1998);
```

在这里`iterator`并不是合适的选择，因为这段代码永远都不会修改`iterator`指向的内容。重新修改代码，改成`const_iterator`s是不重要的，但是在C++98中，有一个改动看起来是合理的，但是仍然是不正确的：

```
typedef std::vector<int>::iterator IterT;    // typetypedef
std::vector<int>::const_iterator ConstIterT; // defs
std::vector<int> values;
…
ConstIterT ci =
std::find(static_cast<ConstIterT>(values.begin()), // cast
static_cast<ConstIterT>(values.end()), 1983);      // cast
values.insert(static_cast<IterT>(ci), 1998);       // 可能无法编译
                                                   // 参考后续解释
```

`typedef`并不是必须的，当然，这会使得代码更加容易编写。（如果你想知道为什么使用`typedef`而不是使用规则9中建议使用的别名声明，这是因为这个例子是C++98的代码，别名声明的特性是C++11的。）

在`std::find`中的强制类型转换是因为`values`是在C++98中是非`const`的容器，但是并没有比较好的办法可以从一个非`const`容器中得到一个`const_iterator`。强制类型转换并非必要的，因为可以从其他的办法中得到`const_iterator`（比如，可以绑定`values`到一个`const`的

引用变量，然后使用这个变量代替代码中的 `values` ），但是不管使用哪种方式，从一个非 `const` 容器中得到一个 `const_iterator` 牵涉到太多。

一旦使用了 `const_iterator` ，麻烦的事情会更多，因为在C++98中，插入或者删除元素的定位只能使用 `iterator` ， `const_iterator` 是不行的。这就是为什么在上面的代码中，我把 `const_iterator` （从 `std::find` 中小心翼翼的拿到的）有转换成了 `iterator` ： `insert` 给一个 `const_iterator` 会编译不过。

老实说，我上面展示的代码可能就编译不过，这是因为并没有合适的从 `const_iterator` 到 `interator` 的转换，甚至是使用 `static_cast` 也不行。甚至最暴力的 `reinterpret_cast` 也不成。（这不是C++98的限制，同时C++11也同样如此。 `const_iterator` 转换不成 `iterator` ，不管看似有多么合理。）还有一些方法可以生成类似 `const_iterator` 行为的 `iterator` ，但是它们都不是很明显，也不通用，本书中就不讨论了。除此之外，我希望我所表达的观点已经明确： `const_iterator` 在C++98中非常麻烦事，是万恶之源。那时候，开发者在必要的地方并不使用 `const_iterator` ，在C++98中 `const_iterator` 是非常不实用的。

所有的一切在C++11中发生了变化。现在 `const_iterator` 既容易获得也容易使用。容器中成员函数 `cbegin` 和 `cend` 可以产生 `const_iterator` ，甚至非 `const` 的容器也可以这样做，STL成员函数通常使用 `const_iterator` 来进行定位（也就是说，插入和删除insert and erase）。修订原来的C++98的代码使用C++11的 `const_iterator` 替换原来的 `iterator` 是非常的简单的事情：

```
std::vector<int> values; // 和之前一样
…
auto it = // use cbegin
std::find(values.cbegin(),values.cend(), 1983); // and cend
values.insert(it, 1998);
```

现在代码使用 `const_iterator` 非常的实用！

在C++11中只有一种使用 `const_iterator` 的短处就是在编写最大化泛型库的代码的时候。代码需要考虑一些容器或者类似于容器的数据结构提供 `begin` 和 `end` （加上cbegin, cend, rbegin等等）作为非成员函数而不是成员函数。例如这种情况针对于内建的数组，和一些第三方库中提供一些接口给自由无约束的函数来使用。最大化泛型代码使用非成员函数而不是使用成员函数的版本。

Charpter4 智能指针

诗人和作曲家喜欢写一些关于love的作品，也有可能写一些关于计数(counting)的作品,很少有两者兼顾的。总有些例外，如Elizabeth Barrett Browning:"How do I love thee? Let me count the ways",又如Paul Simon:"There must be 50 ways to leave your lover.",被这些诗句启发，我们来尝试列举下为什么原生指针(raw pointer)不那么讨人喜欢(love)的理由:

1.从它的声明看不出它指向的是一个单个的对象还是一个数组

2.当你使用完它的时候，从它的声明看不出来你是否应该把它销毁，例如，当指针拥有(owns)它当前指向的对象时

3.当你确定要销毁它指向的内容的时候，又要犯难了，因为你不知道要使用delete，还是要使用另外一个不同的销毁机制(如将该指针传递到一个指定的析构函数里)

4.当你终于要使用delete决定要销毁它了,因为第1条，你又不知道该使用delete还是delete[],因为一旦使用错误，结果会是不确定的

5.最后，你终于确定了指针指向的内容是啥了，也确定了改用什么样的方式来销毁；问题又来了，因为你不能保证在你的程序的每条路径中，你的销毁代码只执行一次，不执行的话会造成内存泄露，多执行哪怕一次会产生不确定的行为

6.目前没有方法来确定一个指针是悬挂指针,即确定一个指针不再拥有它指向的对象。当一个指针指向的对象被销毁了，该指针就变成了悬挂指针。

原生指针是一款很强大的工具，但是依据进数十年的经验，可以确定的一点是:稍有不慎，这个工具就会反噬它的使用者。

终于，来解决上述难题的智能指针出现了，智能指针表现起来很像原生指针，它相当于是原生指针的一层再包装(wrapper)，但是规避了许多使用原生指针带来的陷阱。你应该尽量使用智能指针，它几乎能做到原生指针能做到的所有功能，却很少给你犯错的机会。

在C++11标准中规定了四个智能指针:std::auto_ptr, std::unique_ptr, std::shared_ptr, 以及std::weak_ptr.它们都用来设计辅助管理动态分配对象的生命周期，即，确保这些对象在正确的时间(包括发生异常时)用正确的方式进行回收，以确保不会产生内存泄露.

C++98尝试用std::auto_ptr来标准化后来成为C++11中的std::unique_ptr的行为，为了达到目标，move语法是不可少的，但是，C++98当时还没有move语法，所以做了个妥协方案:利用拷贝操作来模拟move.这导致了一些很让人吃惊的代码(如拷贝一个std::auto_ptr会将它设置为null!)和一些让使用者觉得沮丧的使用限制(不能在容器中使用std::auto_ptr)

std::unique_ptr做到了std::auto_ptr所能做到的所有事情，而且它的实现还更高效。

智能指针的API有着显著的区别，他们之间唯一共同的一点功能就是默认的构造方法。因为这种API详细的介绍满大街都是啊，所以我把重点放到了这些API介绍所没有的知识，如:值得注意的使用场景，运行性能分析等等。掌握这些信息你就不只会可以单单的使用它们，更是学会了如何有效的运用它们。

Item18:

当你要使用一个智能指针时，首先要想到的应该是 `std::unique_ptr` .下面是一个很合理的假设:默认情况下， `std::unique_ptr` 和原生指针同等大小，对于大多数操作(包括反引用)，它们执行的底层指令也一样。这就意味着，尽管在内存回收直来直往的情况下， `std::unique_ptr` 也足以胜任原生指针轻巧快速的使用要求。

`std::unique_ptr` 具现了独占(exclusive ownership)语义,一个非空的 `std::unique_ptr` 永远拥有它指向的对象，move一个 `std::unique_ptr` 会将所有权从源指针转向目的指针(源指针指向为null)。拷贝一个 `std::unique_ptr` 是不允许的，假如说真的可以允许拷贝 `std::unique_ptr` ，那么将会有两个 `std::unique_ptr` 指向同一块资源区域，每一个都认为它自己拥有且可以摧毁那块资源。因此， `std::unique_ptr` 是一个move-only类型。当它面临析构时，一个非空的 `std::unique_ptr` 会摧毁它所拥有的资源。默认情况下， `std::unique_ptr` 会使用delete来释放它所包裹的原生指针指向的空间。

`std::unique_ptr` 的一个常见用法是作为一个工厂函数返回一个继承层级中的一个特定类型的对象。假设我们有一个投资类型的继承链。

[18-1.png]

```
class Investment { ... };
class Stock:public Investment { ... };
class Bond:public Investment { ... };
class RealEstate:public Investment { ... };
```

生产这种层级对象的工厂函数通常在堆上面分配一个对象并且返回一个指向它的指针。当不再需要使用时，调用者来决定是否删除这个对象。这是一个绝佳的 `std::unique_ptr` 的使用场景。因为调用者获得了由工厂函数分配的对象的所有权(并且是独占性的)，而且 `std::unique_ptr` 在自己即将被销毁时，自动销毁它所指向的空间。一个为Investment层级对象设计的工厂函数可以声明如下：

```
template<typename... Ts>
std::unique_ptr<Investment> makeInvestment(Ts&&... params);// return std::unique_ptr
    // to an object created
    // from the given args
```

调用者可以在一处代码块中使用返回的 `std::unique_ptr` ：

```
{
    ...
    auto pInvestment = makeInvestment( arguments );
    //pInvestment is of type std::unique_ptr<Investment>
    ...
}//destroy *pInvestment
```

他们也可以使用在拥有权转移的场景中，例如当工厂函数返回的 `std::unique_ptr` 可以移动到一个容器中，这个容器随即被移动到一个对象的数据成员上，该对象随后即被销毁。当该对象被销毁后，该对象的 `std::unique_ptr` 数据成员也随即被销毁，它的析构会引发工厂返回的资源被销毁。如果拥有链因为异常或者其他的异常控制流(如，函数过早返回或者for循环中的break语句)中断，最终拥有资源的 `std::unique_ptr` 仍会调用它的析构函数(注解：这条规则仍有例外：大多数源自于程序的非正常中断。一个从一个线程主函数(如程序的初始线程的main函数)传递出来的异常，或者一个违背了noexpect规范(请看Item 14)的异常,本地对象不会得到析构，如果 `std::abort` 或者其他的exit函数(如 `std::_Exit` , `std::exit` ,或者 `std::quick_exit`)被调用，那么它们肯定不会被析构)， `std::unique_ptr` 管理的资源也因此得到释放。

默认情况下，析构函数会使用delete。但是，我们也可以在它的构造过程中指定特定的析构方法(custom deleters):当资源被回收时，传入的特定的析构方法(函数对象，或者是特定的lambda表达式)会被调用。对于我们的例子来说，如果被makeInvestment创建的对象不应该直接被deleted，而是首先要有一条log记录下来，我们就可以这样实现makeInvestment（当你看到意图不是很明显的代码时，请注意看注释）

```
auto delInvmt = [](Investment* pInvestment){
    makeLogEntry(pInvestment);
    delete pInvestment;
};//custom deleter(a lambda expression)
template<typename... Ts>
std::unique_ptr<Investment, decltype(delInvmt)>//revised return type
makeInvestment(Ts&&... params)
{
    std::unique_ptr<Investment, decltype(delInvmt)> pInv(nullptr, delInvmt);//ptr to b
e returned
    if ( /* a Stock object should be created */ )
    {
       pInv.reset(new Stock(std::forward<Ts>(params)...));
    }
    else if ( /* a Bond object should be created */ )
    {
       pInv.reset(new Bond(std::forward<Ts>(params)...));
    }
    else if ( /* a RealEstate object should be created */ )
    {
       pInv.reset(new RealEstate(std::forward<Ts>(params)...));
    }
    return pInv;
}
```

我之前说过，当使用默认的析构方法时(即，delete)，你可以假设 `std::unique_ptr` 对象的大小和原生指针一样。当 `std::unique_ptr` 用到了自定义的deleter时，情况可就不一样了。函数指针类型的deleter会使得 `std::unique_ptr` 的大小增长到一个字节到两个字节。对于deleters是函数对象的 `std::unique_ptr` ,大小的改变依赖于函数对象内部要存储多少状态。无状态的函数对象(如，没有captures的lambda expressions) 不会导致额外的大小开销。这就意味着当一个自定义的deleter既可以实现为一个函数对象或者一个无捕获状态的lambda表达式时，lambda是第一优先选择:

```
auto delInvmt1 = [](Investment* pInvestment)
                 {
                     makeLogEntry(pInvestment);
                     delete pInvestment;
                 }
//custom deleter as stateless lambda
template<typename... Ts>
std::unique_ptr<Investment, decltype(delInvmt1)>
makeInvestment(Ts&&.. args);//return type has size of Investment*

void delInvmt2(Investment* pInvestment)
{
    makeLogEntry(pInvestment);
    delete pInvestment;
}

template<typename... Ts>
std::unique_ptr<Investment,(void *)(Investment*)>
makeInvestment(Ts&&... params);//return type has size of Investment* plus at least siz
e of function pointer!
```

带有过多状态的函数对象的deleters是使得 `std::unique_ptr` 的大小得到显著的增加。如果你发现一个自定义的deleter使得你的 `std::unique_ptr` 大到无法接受，请考虑重新改变你的设计。

`std::unique_ptr` 会产生两种格式，一种是独立的对象(std::unique_ptr)，另外一种是数组(`std::unique_ptr<T[]>`).因此，std::unique_ptr指向的内容从来不会产生任何歧义性。它的API是专门为了你使用的格式来设计的.例如，单对象格式中没有过索引操作符(操作符[]),数组格式则没有解引用操作符(操作符*和操作符->)

`std::unique_ptr` 的数组格式对你来说可能是华而不实的东东，因为和原生的array相比， `std::array` , `std::vector` 以及 `std::string` 几乎是更好的数据结构选择。我所想到的唯一的std::unique_ptr有意义的使用场景是，你使用了C-like API来返回一个指向堆内分配的数组的原生指针，而且你像对之接管拥有权。

C++11使用 `std::unique_ptr` 来表述独占所有权。但是它的一项最引人注目的特性就是它可以轻易且有效的转化为 `std::shared_ptr` ：

```
std::shared_ptr<Investment> sp = makeInvestment(arguments);//converts std::unique_ptr
to std::shared_ptr
```

这就是 `std::unique_ptr` 很适合作为工厂函数返回值类型的原因。工厂函数不知道调用者想使用独占性的拥有语义还是共享式的拥有语义(即 `std::share_ptr`).通过返回 `std::unique_ptr` ，工厂函数将选择权移交给了调用者,调用者在需要的时候可以将 `std::unique_ptr` 转化为它最富有灵活性的兄弟(如果想了解更多关于 `std::shared_ptr` ,请移步Item 19)

要记住的东西
`std::unique_ptr` 是一个具有开销小，速度快， `move-only` 特定的智能指针，使用独占拥有方式来管理资源。
默认情况下，释放资源由delete来完成，也可以指定自定义的析构函数来替代。但是具有丰富状态的deleters和以函数指针作为deleters增大了 `std::unique_ptr` 的存储开销
很容易将一个 `std::unique_ptr` 转化为 `std::shared_ptr`

Item 19:使用std::shared_ptr来管理共享式的资源

使用垃圾回收机制的程序员指责并且嘲笑C++程序员阻止内存泄露的做法。“你们tmd是原始人!”他们嘲笑道。“你们有没有看过1960年Lisp语言的备忘录？应该用机器来管理资源的生命周期，而不是人类。”C++程序员开始翻白眼了:"你们懂个屁，如果备忘录的内容意味着唯一的资源是内存而且回收资源的时机是不确定性的，那么我们宁可喜欢具有普适性和可预测性的析构函数."但是我们的回应部分是虚张声势。垃圾回收确实非常方便，手动来控制内存管理周期听起来像是用原始工具来做一个记忆性的内存回路。为什么我们不两者兼得呢？做出一个既可以想垃圾回收那样自动，且可以运用到所有资源，具有可预测的回收时机(像析构函数那样)的系统。

`std::shared_ptr` 就是C++11为了达到上述目标推出的方式。一个通过 `std::shared_ptr` 访问的对象被指向它的指针通过共享所有权(shared ownership)方式来管理.没有一个特定的 `std::shared_ptr` 拥有这个对象。相反，这些指向同一个对象的 `std::shared_ptr` 相互协作来确保该对象在不需要的时候被析构。当最后一个 `std::shared_ptr` 不再指向该对象时(例如，因为 `std::shared_ptr` 被销毁或者指向了其他对象)， `std::shared_ptr` 会在此之前摧毁这个对象。就像GC一样，使用者不用担心他们如何管理指向对象的生命周期，而且因为有了析构函数，对象析构的时机是可确定的。

一个 `std::shared_ptr` 可以通过查询资源的引用计数(reference count)来确定它是不是最后一个指向该资源的指针，引用计数是一个伴随在资源旁的一个值，它记录着有多少个 `std::shared_ptr` 指向了该资源。 `std::shared_ptr` 的构造函数会自动递增这个计数，析构函数会自动递减这个计数，而拷贝构造函数可能两者都做(比如，赋值操作 `sp1=sp2` ,sp1和sp2都是 `std::shared_ptr` 类型，它们指向了不同的对象，赋值操作使得sp1指向了原来sp2指向的对象。赋值带来的连锁效应使得原来sp1指向的对象的引用计数减1，原来sp2指向的对象的引用计数加1.)如果 `std::shared_ptr` 在执行减1操作后发现引用计数变成了0，这就说明了已经没有其他的 `std::shared_ptr` 在指向这个资源了，所以 `std::shared_ptr` 直接析构了它指向的空间。

引用计数的存在对性能会产生部分影响

- `std::shared_ptrs` 是原生指针的两倍大小，因为它们内部除了包含了一个指向资源的原生指针之外，同时还包含了指向资源的引用计数
- 引用计数的内存必须被动态分配.概念上来说，引用计数会伴随着被指向的对象，但是被指向的对象对此一无所知。因此，他们没有为引用计数准备存储空间。(一个好消息是任何对象，即使是内置类型，都可以被 `std::shared_ptr` 管理。)Item21解释了

用 `std::make_shared` 来创建 `std::shared_ptr` 的时候可以避免动态分配的开销，但是有些情况下 `std::make_shared` 也是不能被使用的。不管如何，引用计数都是存储为动态分配的数据

- 引用计数的递增或者递减必须是原子的，因为在多线程环境下，会同时存在多个写者和读者。例如，在一个线程中，一个 `std::shared_ptr` 指向的资源即将被析构(因此递减它所指向资源的引用计数)，同时，在另外一个线程中，一个 `std::shared_ptr` 指向了同一个对象，它此时正进行拷贝操作(因此要递增同一个引用计数)。原子操作通常要比非原子操作执行的慢，所以尽管引用计数通常只有一个word大小，但是你可假设对它的读写相对来说比较耗时。

当我写到: `std::shared_ptr` 构造函数在构造时"通常"会增加它指向的对象的引用计数时，你是不是很好奇？创建一个新的指向某对象的 `std::sharedptr` 会使得指向该对象的 `std::sharedptr` 多出一个，为什么我们不说构造一个 `std::sharedptr` 总是会增加引用计数？

Move构造函数是我为什么那么说的原因。从另外一个 `std::shared_ptr` move构造(Move-constructing)一个 `std::shared_ptr` 会使得源 `std::shared_ptr` 指向为null,这就意味着新的 `std::shared_ptr` 取代了老的 `std::shared_ptr` 来指向原来的资源，所以就不需要再修改引用计数了。Move构造 `std::shared_ptr` 要比拷贝构造 `std::shared_ptr` 快：copy需要修改引用计数，然而拷贝缺不需要。对于赋值构造也是一样的。最后得出结论，move构造要比拷贝构造快，Move赋值要比copy赋值快。

像 `std::unique_ptr` (Item 18)那样, `std::shared_ptr` 也把delete作为它默认的资源析构机制。但是它也支持自定义的deleter.然后，它支持这种机制的方式不同于 `std::unique_ptr` .对于 `std::unique_ptr` ,自定义的deleter是智能指针类型的一部分，对于 `std::shared_ptr` ,情况可就不一样了:

```
auto loggingDel = [](widget *pw)
                    {
                        makeLogEntry(pw);
                        delete pw;
                    }//自定义的deleter(如Item 18所说)
std::unique_ptr<Widget, decltype(loggingDel)>upw(new Widget, loggingDel);//deleter类型是智能指针类型的一部分

std::shared_ptr<Widget> spw(new Widget, loggingDel);//deleter类型不是智能指针类型的一部分
```

std::shared_prt的设计更加的弹性一些，考虑到两个std::shared_ptr,每一个都支持不同类型的自定义deleter(例如，两个不同的lambda表达式):

```
auto customDeleter1 = [](Widget *pw) {...};
auto customDeleter2 = [](Widget *pw) {...};//自定义的deleter,属于不同的类型

std::shared_prt<Widget> pw1(new Widget, customDeleter1);
std::shared_prt<Widget> pw2(new Widget, customDeleter2);
```

因为pw1和pw2属于相同类型，所以它们可以放置到属于同一个类型的容器中去:

```
std::vector<std::shared_ptr<Widget>> vpw{ pw1, pw2 };
```

它们之间可以相互赋值，也都可以作为一个参数类型为 `std::shared_ptr<Widget>` 类型的函数的参数。所有的这些特性，具有不同类型的自定义deleter的 `std::unique_ptr` 全都办不到，因为自定义的deleter类型会影响到 `std::unique_ptr` 的类型。

与 `std::unique_ptr` 不同的其他的一点是，为 `std::shared_ptr` 指定自定义的deleter不会改变 `std::shared_ptr` 的大小。不管deleter如何，一个 `std::shared_ptr` 始终是两个pointer的大小。这可是个好消息，但是会让我们一头雾水。自定义的deleter可以是函数对象，函数对象可以包含任意数量的data.这就意味着它可以是任意大小。涉及到任意大小的自定义deleter的 `std::shared_ptr` 如何保证它不使用额外的内存呢？

它肯定是办不到的，它必须使用额外的空间来完成上述目标。然而，这些额外的空间不属于 `std::shared_ptr` 的一部分。额外的空间被分配在堆上，或者在 `std::shared_ptr` 的创建者使用了自定义的allocator之后，位于该allocator管理的内存中。我之前说过，一个 `std::shared_ptr` 对象包含了一个指针，指向了它所指对象的引用计数。此话不假，但是却有一些误导性，因为引用计数是一个叫做控制块(control block)的很大的数据结构。每一个由 `std::shared_ptr` 管理的对象都对应了一个控制块。改控制块不仅包含了引用计数，还包含了一份自定义deleter的拷贝(在指定好的情况下).如果指定了一个自定义的allocator,也会被包含在其中。控制块也可能包含其他的额外数据，比如Item 21条所说，一个次级(secondary)的被称作是weak count的引用计数，在本Item中我们先略过它。我们可以想象出 `std::shared_ptr<T>` 的内存布局如下所示:

一个对象的控制块被第一个创建指向它的 `std::shared_ptr` 的函数来设立.至少这也是理所当然的。一般情况下，函数在创建一个 `std::shared_ptr` 时，它不可能知道这时是否有其他的 `std::shared_ptr` 已经指向了这个对象，所以在创建控制块时，它会遵循以下规则：

- `std::make_shared` (请看Item 21)总是会创建一个控制块。它制造了一个新的可以指向的对象，所以可以确定这个新的对象在 `std::make_shared` 被调用时肯定没有相关的控制块。
- 当一个 `std::shared_ptr` 被一个独占性的指针(例如，一个 `std::unique_ptr` 或者 `std::auto_ptr`)构建时，控制块被相应的被创建。独占性的指针并不使用控制块，所以被指向的对象此时还没有控制块相关联。(构造的一个过程是，由 `std::shared_ptr` 来接管了被指向对象的所有权，所以原来的独占性指针被设置为null).

- 当一个 `std::shared_ptr` 被一个原生指针构造时，它也会创建一个控制块。如果你想要基于一个已经有控制块的对象来创建一个 `std::shared_ptr` ，你可能传递了一个 `std::shared_ptr` 或者 `std::weak_ptr` 作为 `std::shared_ptr` 的构造参数，而不是传递了一个原生指针。 `std::shared_ptr` 构造函数接受 `std::shared_ptr` 或者 `std::weak_ptr` 时，不会创建新的控制块，因为它们(指构造函数)会依赖传递给它们的智能指针是否已经指向了带有控制块的对象的情况。

当使用了一个原生的指针构造多个 `std::shared_ptr` 时，这些规则的存在会使得被指向的对象包含多个控制块，带来许多负面的未定义行为。多个控制块意味着多个引用计数，多个引用计数意味着对象会被摧毁多次(每次引用计数一次)。这就意味着下面的代码着实糟糕透顶：

```
auto pw = new Widget;                           //pw是一个原生指针
...
std::shared_ptr<Widget> spw1(pw, loggingDel);//为*pw创建了一个控制块
...
std::shared_ptr<Widget> spw2(pw, loggingDel);//为pw创建了第二个控制块!
```

创建原生指针pw的行为确实不太好，这样违背了我们一整章背后的建议(请看开章那几段话来复习)。但是先不管这么多，创建pw的那行代码确实不太建议，但是至少它没有产生程序的未定义行为.

现在的情况是，因为spw1的构造函数的参数是一个原生指针，所以它为指向的对象(就是pw指向的对象: `*pw`)创造了一个控制块(伴随着一个引用计数)。到目前为止，代码还没有啥问题。但是随后，spw2也被同一个原生指针作为参数构造,它也为 `*pw` 创造了一个控制块(还有引用计数). `*pw` 因此拥有了两个引用计数。每一个最终都会变成0，最终会引起两次对 `*pw` 的析构行为。第二次析构就要对未定义的行为负责了。

对于 `std::shared_ptr` 在这里总结两点.首先，避免给std::shared_ptr构造函数传递原生指针。通常的取代做法是使用std::make_shared(请看Item 21).但是在上面的例子中，我们使用了自定义的deleter,这对于std::make_shared是不可能的。第二，如果你必须要给std::shared_ptr构造函数传递一个原生指针，那么请直接传递new语句，上面代码的第一部分如果被写成下面这样：

```
std::shared_ptr<Widget> spw1(new Widget,loggingDel);//direct use of new
```

这样就不大可能从同一个原生指针来构造第二个 `std::shared_ptr` 了。而且，创建spw2的代码作者会用spw1作为初始化(spw2)的参数(即，这样会调用std::shared_ptr的拷贝构造函数)。这样无论如何都不有问题:

```
std::shared_ptr<Widget> spw2(spw1);//spw2 uses same control block as spw1
```

使用this指针时，有时也会产生因为使用原生指针作为 `std::shared_ptr` 构造参数而导致的产生多个控制块的问题。假设我们的程序使用 `std::shared_ptr` 来管理Widget对象，并且我们使用了一个数据结构来管理跟踪已经处理过的Widget对象：

```
std::vector<std::shared_ptr<Widget>> processedWidgets;
```

进一步假设Widget有一个成员函数来处理:

```
class Widget{
public:
    ...
    void process();
    ...
};
```

这有一个看起来很合理的Widget::process实现

```
void Widget::process()
{
    ...                          //process the Widget
    processedWidgets.emplace_back(this);//add it to list
                                        //processed Widgets;
                                        //this is wrong!
}
```

注释里面说这样做错了，指的是传递this指针，并不是因为使用了 `emplace_back` (如果你对 `emplace_back` 不熟悉，请看Item 42.)这样的代码会通过编译，但是给一个 `std::shared_ptr` 传递this就相当于传递了一个原生指针。所以 `std::shared_ptr` 会给指向的Widget(*this)创建了一个新的控制块。当你意识到成员函数之外也有 `std::shared_ptr` 早已指向了Widget，这就粗大事了，同样的道理，会导致发生未定义的行为。

`std::shared_ptr` 的API包含了修复这一问题的机制。这可能是C++标准库里面最诡异的方法名字了： `std::enabled_from_this` .它是一个基类的模板，如果你想要使得被std::shared_ptr管理的类安全的以this指针为参数创建一个 `std::shared_ptr` ,就必须要继承它。在我们的例子中，Widget会以如下方式继承 `std::enable_shared_from_this` ：

```
class Widget: public std::enable_shared_from_this<Widget>{
    public:
        ...
        void process();
        ...
};
```

正如我之前所说的，`std::enable_shared_from_this`是一个基类模板。它的类型参数永远是它要派生的子类类型，所以widget继承自`std::enable_shared_from_this<widget>`。如果这个子类继承自以子类类型为模板参数的基类的想法让你觉得不可思议，先放一边吧，不要纠结。以上代码是合法的，并且还有相关的设计模式，它有一个非常名字，虽然像`std::enable_shared_from_this`一样古怪,名字叫The Curiously Recurring Template Pattern(CRTP).欲知详情请使用你的搜索引擎。我们下面继续讲`std::enable_shared_from_this`.

`std::enable_shared_from_this`定义了一个成员函数来创建指向当前对象的`std::shared_ptr`，但是它并不重复创建控制块。这个成员函数的名字是`shared_from_this`,当你实现一个成员函数，用来创建一个`std::shared_ptr`来指向this指针指向的对象,可以在其中使用`shared_from_this`。下面是Widget::process的一个安全实现:

```
void Widget::process()
{
    //as before, process the Widget
    ...
    //add std::shared_ptr to current object to processedWidgets
    processedWidgets.emplace_back(shared_from_this());
}
```

`shared_from_this`内部实现是，它首先寻找当前对象的控制块，然后创建一个新的`std::shared_ptr`来引用那个控制块。这样的设计依赖一个前提，就是当前的对象必须有一个与之相关的控制块。为了让这种情况成真，事先必须有一个`std::shared_ptr`指向了当前的对象(比如说，在这个调用`shared_from_this`的成员函数的外面)，如果这样的`std::shared_ptr`不存在(即，当前的对象没有相关的控制块)，虽然shared_from_this通常会抛出异常，产生的行为仍是未定义的。

为了阻止用户在没有一个`std::shared_ptr`指向该对象之前，使用一个里面调用`shared_from_this`的成员函数，继承自`std::enable_shared_from_this`的子类通常会把它们的构造函数声明为private,并且让它们的使用者利用返回`std::shared_ptr`的工厂函数来创建对象。举个栗子，对于Widget来说，可以像下面这样写：

```
class Widget: public std::enable_shared_from_this<Widget>{
public:
    //工厂函数转发参数到一个私有的构造函数
    template<typename... Ts>
    static std::shared_ptr<Widget> create(Ts&&... params);
    ...
    void process();            //as before
    ...
private:
    ...                              //构造函数
}
```

直到现在，你可能只能模糊的记得我们关于控制块的讨论源自于想要理解 `std::shared_ptr` 性能开销的欲望。既然我们已经理解如何避免创造多余的控制块，下面我们回归正题吧。

一个控制块可能只有几个字节大小，尽管自定义的deleters和allocators可能会使得它更大。通常控制块的实现会比你想象中的更复杂。它利用了继承，甚至还用到虚函数(确保指向的对象能正确销毁。)这就意味着使用 `std::shared_ptr` 会因为控制块使用虚函数而导致一定的机器开销。

当我们读到了动态分配的控制块，任意大小的deleters和allocators,虚函数机制，以及引用计数的原子操纵，你对 `std::shared_ptr` 的热情可能被泼了一盆冷水，没关系.它做不到对每一种资源管理的问题都是最好的方案。但是相对于它提供的功能， `std::shared_ptr` 性能的耗费还是很合理。通常情况下， `std::shared_ptr` 被 `std::make_shared` 所创建，使用默认的deleter和默认的allocator,控制块也只有大概三个字节大小。它的分配基本上是不耗费空间的(它并入了所指向对象的内存分配，欲知详情，请看Item 21.)解引用一个 `std::shared_ptr` 花费的代价不会比解引用一个原生指针更多。执行一个需要操纵引用计数的过程(例如拷贝构造和拷贝赋值，或者析构)需要一直两个原子操作，但是这些操作通常只会映射到个别的机器指令，尽管相对于普通的非原子指令他们可能更耗时，但它们终究仍是单个的指令。控制块中虚函数的机制在被 `std::shared_ptr` 管理的对象的生命周期中一般只会被调用一次：当该对象被销毁时。

花费了相对很少的代价，你就获得了对动态分配资源生命周期的自动管理。大多数时间，想要以共享式的方式来管理对象，使用 `std::shared_ptr` 是一个大多数情况下都比较好的选择。如果你发现自己开始怀疑是否承受得起使用 `std::shared_ptr` 的代价时，首先请重新考虑是否真的需要使用共享式的管理方法。如果独占式的管理方式可以或者可能实用， `std::unique_ptr` 或者是更好的选择。它的性能开销于原生指针大致相同，并且从 `std::unique_ptr` “升级”到s `td::shared_ptr` 是很简单的，因为 `std::shared_ptr` 可以从一个 `std::unique_ptr` 里创建。

反过来可就不一定好用了。如果你把一个资源的生命周期管理交给了 `std::shared_ptr` ，后面没有办法在变化了。即使引用计数的值是1，为了让 `std::unique_ptr` 来管理它，你也不能重新声明资源的所有权。资源和指向它的 `std::shared_ptr` 之间的契约至死方休。不许离婚，取消或者变卦。

还有一件事情 `std::shared_ptr` 不好用，那就是用在数组上面。可 `std::unique_ptr` 不同的一点就是， `std::shared_ptr` 的API设计为指向单个的对象。没有像 `std::shared_ptr<T[]>` 这样的用法。经常有一些自作聪明的程序员使用 `std::shared_ptr<T>` 来指向一个数组,指定了一个自定义的deleter来做数组的删除操作(即delete[]).这样做可以通过编译，但是却是个坏主意，原因有二，首先， `std::shared_ptr` 没有重载操作符[],所以如果是通过数组访问需要通过丑陋的基于指针的运算来进行，第二， `std::shared_ptr` supports derived-to-base pointer conversions that make sense for single objects, but that open holes in the type system when applied to arrays. (For this reason, the `std::unique_ptr<T[]>` API prohibits such

conversions.)更重要的一点是，鉴于C++11标准给了比原生数组更好的选择(例如，`std::array`，`std::vector`，`std::string`)，给数组来声明一个智能指针通常是不当设计的表现。

要记住的东西
`std::shared_ptr` 为了管理任意资源的共享式内存管理提供了自动垃圾回收的便利
`std::shared_ptr` 是 `std::unique_ptr` 的两倍大，除了控制块，还有需要原子引用计数操作引起的开销
资源的默认析构一般通过delete来进行，但是自定义的deleter也是支持的。deleter的类型对于 `std::shared_ptr` 的类型不会产生影响
避免从原生指针类型变量创建 `std::shared_ptr`

Item 20：Use `std::weak_ptr` for `std::shared_ptr` like pointers that can dangle.

说起来有些矛盾，可以很方便的创建一个表现起来想 `std::shared_ptr` 的智能指针，但是它却不会参于被指向资源的共享式管理。换句话说，一个类似于 `std::shared_ptr` 的指针不影响它所指对象的引用计数。这种类型的智能指针必须面临一个 `std::shared_ptr` 未曾面对过的问题:它所指向的对象可能已经被析构。一个真正的智能指针通过持续跟踪判断它是否已经悬挂(dangle)来处理这种问题，悬挂意味着它指向的对象已经不复存在。这就是 `std::weak_ptr` 的功能所在

你可能怀疑 `std::weak_ptr` 怎么会有用，当你检查了下 `std::weak_ptr` 的API之后，你会觉得更奇怪。它的API看起来一点都不智能。 `std::weak_ptr` 不能被解引用，也不能检测判空。这是因为 `std::weak_ptr` 不能被单独使用，它是 `std::shared_ptr` 作为参数的产物。

这种关系与生俱来， `std::weak_ptr` 通常由一个 `std::shared_ptr` 来创建，它们指向相同的地方， `std::shared_ptr` 来初始化它们，但是 `std::weak_ptr` 不会影响到它所指向对象的引用计数:

```
auto spw = std::make_shared<Widget>();//spw 被构造之后
                              //被指向的Widget对象的引用计数为1
                              //(欲了解std::make_shared详情，请看Item21)
...
std::weak_ptr<Widget> wpw(spw);//wpw和spw指向了同一个Widget,但是RC(这里指引用计数，下同)仍旧是1
...
spw = nullptr;//RC变成了0，Widget也被析构，wpw现在处于悬挂状态
```

悬挂的std::weak_ptr可以称作是过期了(expired),可以直接检查是否过期：

```
if(wpw.expired())...          //如果wpw悬挂...
```

但是我们最经常的想法是：查看 `std::weak_ptr` 是否已经过期，如果没有过期的话，访问它所指向的对象。想的容易做起来难啊。因为 `std::weak_ptr` 缺少解引用操作，也就没办法写完成这样操作的代码。即使又没法做到，将检查和解引用分开的写法也会引入一个竞态存在：在调用expired以及解引用操作之间，另外一个线程可能对被指向的对象重新赋值或者摧毁了最后一个指向对象的 `std::shared_ptr` ,这样就导致了被指向的对象的析构。这种情况下，你的解引用操作会产生未定义行为。

我们需要的是将检查 `std::weak_ptr` 是否过期，以及如果未过期的话获得访问所指对象的权限这两种操作合成一个原子操作。这是通过由 `std::weak_ptr` 创建出一个 `std::shared_ptr` 来完成的。根据当 `std::weak_ptr` 已经过期，仍以它为参数创建 `std::shared_ptr` 会发生的情况的不同，这种创建有两种方式。一种方式是通过 `std::weak_ptr::lock` ,它会返回一个 `std::shared_ptr` ,当 `std::weak_ptr` 已经过期时， `std::shared_ptr` 会是null：

```
std::shared_ptr<Widget> spw1 = wpw.lock();//如果wpw已经过期
//spw1的值是null
auto spw2 = wpw.lock();//结果同上，这里使用了auto
```

另外一种方式是以 `std::weak_ptr` 为参数，使用 `std::shared_ptr` 构造函数。这种情况下，如果 `std::weak_ptr` 过期的话，会有异常抛出：

```
std::shared_ptr<Widget> spw3(wpw);//如果wpw过期的话
//抛出std::bad_weak_ptr异常
```

你可能会产生疑问， `std::weak_ptr` 到底有啥用。下面我们举个例子，假如说现在有一个工厂函数，根据一个唯一的ID，返回一个指向只读对象的智能指针。根据Item 18关于工厂函数返回类型的建议，它应该返回一个 `std::unique_ptr` ：

```
std::unique_ptr<const Widget> loadWidget(WidgetID id);
```

如果loadWidget调用的代价不菲(比如，它涉及到了文件或数据库的I/O操作)，而且ID的使用也比较频繁，一个合理的优化就是再写一个函数，不仅完成loadWidget所做的事情，而且要缓存loadWidget的返回结果。把每一个请求过的Widget对象都缓存起来肯定会导致缓存自身的性能出现问题，所以，一个合理的做法是当被缓存的Widget不再使用时将它销毁。

对于这样的一个带有缓存的工厂函数，返回 `std::unique_ptr` 类型不是一个很好的选择。可以确定的两点是：调用者接收指向缓存对象的智能指针，调用者来决定这些缓存对象的生命周期；但是，缓存也需要一个指向所缓存对象的指针。因为当工厂函数的调用者使用完了一个工厂返回的对象，这个对象会被销毁，对应的缓存项会悬挂，所以缓存的指针需要有检测它现在是否处于悬挂状态的能力。因此缓存使用的指针应该是std::weak_ptr类型，它有检测悬挂的能力。这就意味着工厂函数的返回类型应该是 `std::shared_ptr` ,因为只有当一个对象的生命周期被 `std::shared_ptr` 所管理时， `std::weak_ptr` 才能检测它自身是否处于悬挂状态。

下面是一个较快却欠缺完美的缓存版本的loadWidget的实现：

```
std::shared_ptr<const Widget> fastLoadWidget(WidgetId id)
{
    static std::unordered_map<WidgetID,
    std::weak_ptr<const Widget>> cache;
    auto objPtr = cache[id].lock();//objPtr是std::shared_ptr类型
    //指向了被缓存的对象(如果对象不在缓存中则是null)

    if(!objPtr){
        objPtr = loadWidget(id);
        cache[id] = objPtr;
    }//如果不在缓存中，载入并且缓存它
    return objPtr;
}
```

C++11利用了hash表容器(`std::unordered_map`),尽管它没有提供所需的WidgetID哈希算法以及相等比较函数。

我为啥要说fastLoadWidget实现欠缺完美，因为它忽略了一个事实，缓存可能把一些已经过期的 `std::weak_ptr` (对应的Widget不会被使用了，已经被销毁了)。所以它的实现还可以再改善下，但是我们还是不要深究了，因为深究对我们继续深入了解 `std::weak_ptr` 没有用处。我们下面探究第二个使用 `std::weak_ptr` 的场景：在观察者模式中，主要的组成部分是：状态可能会发生变化的subjects，以及当状态变化时需要得到通知的observers.在大多数实现中，每一个subject包含了指向它的observers的数据成员.这就使得subject很容易发送出状态变化的通知。subject对于控制他们的observer的生命周期(observer何时被析构)毫无兴趣.但是，它们必须知道，如果一个observer析构了，subject就不能尝试去访问它了。一个合理的设计是：每一个subject拥有一个 `std::weak_ptr` ，指向了它的observer,这样在可以在访问之间，先检查一下指针是否处于悬挂状态。

下面讲到最后一个 `std::weak_ptr` 的例子，有这样一个数据结构，包含A,B和C。A和C共享B的所有权，它们各自包含了一个 `std::shared_ptr` 指向B

![20-1.png]

如果现在有需要使B拥有反向指针指向A,那么指针应该是什么类型？

![20-2.png]

下面有三种选择：

- 一个原生指针。如果这么做，A如果被析构了，但是C会继续指向B,B包含的指向A的指针现在处于悬挂状态。而B对此毫不知情，所以B有可能不小心反引用了那个悬挂指针，这样会产生未定义的行为。
- 一个 **`std::shared_ptr`** 。在这种设计下，A和B包含了 `std::shared_ptr` 互相指向对方。结果就引发了一个 `std::shared_ptr` 的环(A指向B,B指向A),这个环会使得A和B都不能得到析构。即使程序其他的数据结构都不能访问到A和B(例如，C如果不再指向B)，A和B的引用计数仍然是1.如果这种情况发生了，A和B都会是内存泄露的情况,实际上，程序永远无法

再访问到它们，它们也永远无法得到回收。

- 一个 **`std::weak_ptr`** 。这样避免了以上所有的问题。如果A被回收，B指向它的指针将会悬挂，B也有能力检测到这一状态。此外，就算A和B互相指向对方，B的指针也不会影响到A的引用计数。当没有 `std::shared_ptr` 指向A时，也不会阻止A的析构。

使用 `std::weak_ptr` 毫无疑问是最好的选择。然而，值得注意的是，使用 `std::weak_ptr` 来破坏预期的 `std::shared_ptr` 形成的环不是那么普遍。在定义的比较严格的数据结构，比如说树,子节点一般被父节点所拥有。当父节点被析构时，子节点也应该会被析构。从父节点指向子节点的链接因此最好使用std::unique_ptr.因为子节点不应该比父节点存在的时间过长，从子节点指向父节点的链接可以安全的使用原生指针来实现。因此也不会出现子节点解引用一个指向父节点的悬挂指针。

当然，并不是所有的以指针为基础的数据结构都是严格的层级关系。如果不是的话，就像刚才所说的缓存以及观察者列表的情形，使用 `std::weak_ptr` 是最棒的选择了。

从效率的观点来看， `std::weak_ptr` 和 `std::shared_ptr` 的情况基本相同，。 `std::weak_ptr` 对象的大小和 `std::shared_ptr` 对象相同，它们都利用了同样的控制块(请看Item 19),并且诸如构造，析构以及赋值都涉及到引用计数的原子操作。这可能让你吃了一惊，因为我在本章开始的时候说 `std::weak_ptr` 不参与引用计数的操作。可能没有表达完整我的意思。我要写的意思是 `std::weak_ptr` 不参与对象的共享所有权，因此不影响被指向对象的引用计数。但是，实际上在控制块中存在第二个引用计数， `std::weak_ptr` 来操作这个引用计数。欲知详情，请看Item 21.

要记住的东西
`std::weak_ptr` 用来模仿类似std::shared_ptr的可悬挂指针
潜在的使用 `std::weak_ptr` 的场景包括缓存，观察者列表，以及阻止 `std::shared_ptr` 形成的环

Item 21 优先使用 `std::make_unique` 和 `std::make_shared` 而不是直接使用new

我们先给 `std::make_unique` 以及 `std::make_shared` 提供一个公平的竞争环境，以此开始。 `std::make_shared` 是C++ 11标准的一部分，但是，遗憾的是， `std::make_unique` 不是的。它刚成为C++ 14的一部分。如果你在使用C++11.不要怕，因为你可以很容易自己写一个基本版的 `std::make_unique` ，我们瞧：

```
template<typename T, typename... Ts>
std::unique_ptr<T> make_unique<Ts&&... params>
{
    return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
}
```

如你所见， `make_unique` 只是完美转发了它的参数到它要创建的对象的构造函数中去，由new出来的原生指针构造一个 `std::unique_ptr` ，并且将之返回。这中格式的构造不支持数组以及自定义deleter(请看 Item18),但是它说明只需稍加努力，便可自己创造出所需要的 `make_unique` (备注：为了尽可能花最小大家创造出一个功能齐全的 `make_unique` ，搜索产生它的标准文档，并且拷贝一份文档中的实现。这里所需要的文档是日期为2013-04-18，Stephan T.Lavavej所写的N3656).请记住不要把你自己实现的版本放在命名空间std下面，因为假如说日后你升级到C++ 14的标准库市县，你可不想自己实现的版本和标准库提供的版本产生冲突。

`std::make_unique` 以及 `std::make_shared` 是3个make函数的其中2个：make函数接受任意数量的参数，然后将他们完美转发给动态创建的对象的构造函数，并且返回指向那个对象的智能指针。第三个make函数是 `std::allocate_shared` ,除了第一个参数是一个用来动态分配内存的allocator对象，它表现起来就像 `std::make_shared` .

即使是最普通的是否使用make函数来创建智能指针之间的比较，也表明了为什么使用make函数是比较可行的做法。考虑一下代码:

```
auto upw1(std::make_unique<Widget>());//使用make函数
std::unique_ptr<Widget> upw2(new Widget);//不使用make函数

auto spw1(std::make_shared<Widget>());//使用make函数
std::shared_ptr<Widget> spw2(new Widget);//不使用make函数
```

我已经高亮显示了必要的差别(不好意思，它这里高亮的是Widget,在代码里高亮暂时俺还做不到--译者注)：使用new需要重复写一遍type，而使用make函数不需要。重复敲type违背了软件工程中的一项基本原则：代码重复应当避免。源代码里面的重复会使得编译次数增加,导致对象的代码变得臃肿，由此产生出的code base(code base的含义请至http://en.wikipedia.org/wiki/Codebase--译者注)变得难以改动以及维护。它经常会导致产生不一致的代码。一个code base中的不一致代码会导致bug.并且，敲某段代码两遍会比敲一遍更费事，谁不想写程序时敲比较少的代码呢。

第二个偏向make函数的原因是为了保证产生异常后程序的安全。设想我们有一个函数根据某个优先级来处理Widget：

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

按值传递 `std::shared_ptr` 可能看起来很可疑，但是Item41解释了如果processWidget总是要创建一个 `std::shared_ptr` 的拷贝(例如，存储在一个数据结构中，来跟踪已经被处理过的Widget)，这也是一个合理的设计.

现在我们假设有一个函数来计算相关的优先级

```
int computePriority()
```

如果我们调用processWidget时，使用new而不是 `std::make_shared`：

```
processWidget(std::shared_ptr<Widget>(new Widget),computePriority())
//可能会导致内存泄露！
```

就像注释里面所说的，这样的代码会产生因new引发的Widget对象的内存泄露。但是怎么会这样？函数的声明和调用函数的代码都使用了 `std::shared_ptr` ,设计 `std::shared_ptr` 的目的就是防止内存泄露。当指向资源的最后一个 `std::shared_ptr` 即将离去时，资源会自动得到析构。不管是什么地方，每个人都在用 `std::shared_ptr` ，为什么还会发生内存泄露？

这个问题的答案和编译器将源代码翻译为object code(目标代码，想要知道object code是什么,请看这个问题http://stackoverflow.com/questions/466790/assembly-code-vs-machine-code-vs-object-code)有关系。在运行时(runtime:In computer science, run time, runtime or execution time is the time during which a program is running (executing), in contrast to other phases of a program's lifecycle such as compile time, link time and load time.)。在函数被调用前，函数的参数必须被推算出来，所以在调用processWidget的过程中，processWidget开始执行之前，下面的事情必须要发生：

- "new Widget"表达式必须被执行，即，一个Widget必须在堆上被创建
- 负责管理new所创建的指针的 `std::shared_ptr<Widget>` 的构造函数必须被执行
- computePriority必须被执行

并没有要求编译器产生出对这些操作做到按顺序执行的代码。"new Widget"必须要在std::shared_ptr的构造函数被调用之前执行，因为new的结果作为该构造函数的一个参数,因为computePriority可能在这些调用之前执行，或者之后，更关键的是，或者在它们之间。这样的话，编译器可能按如下操作的顺序产生出代码:

1. 执行"new Widget".
2. 执行computePriority.
3. 执行std::shared_ptr的构造函数.

如果这样的代码在runtime被产生出来,computePriority产生出了一个异常，那么在Step 1中动态分配的Widget可能会产生泄漏.因为它永远不会存储在Step 3中产生的本应负责管理它的 `std::shared_ptr` 中。

使用 `std::make_shared` 可以避免这个问题。调用的代码看起来如下所示：

```
processWidget(std::make_shared<Widget>(),computePriority);//不会有内存泄漏的危险
```

在runtime的时候， `std::make_shared` 或者computePriority都有可能被第一次调用。如果是 `std::make_shared` 先被调用,被动态分配的Widget安全的存储在返回的 `std::shared_ptr` 中(在computePriority被调用之前)。如果computePriority产生了异常， `std::shared_ptr` 的析构函数会负责把它所拥有的Widget回收。如果computePriority首先被调用并且产生出一个异常， `std::make_shared` 不会被调用，因此也不必担心动态分配的Widget会产生泄漏的问题。

如果我们将std::shared_ptr和std::make_shared替换为std::unique_ptr和对应的std::make_unique,同样的分析也会适用。适用std::make_unique而不使用new的原因和使用std::make_shared的目的相同，都是出于写出异常安全(exception-safe)代码的考虑。

一个使用 `std::make_shared` （和直接使用new相比）的显著特性就是提升了效率。使用std::make_shared允许编译器利用简洁的数据结构产生出更简洁，更快的代码。考虑下面直接使用new的效果

```
std::shared_ptr<Widget> spw(new Widget);
```

很明显的情况是代码只需一次内存分配，但实际上它执行了两次。Item 19解释了每一个std::shared_ptr都指向了一个包含被指向对象的引用计数的控制块，控制块的分配工作在std::shared_ptr的构造函数内部完成。直接使用new，就需要一次为Widget分配内存，第二次需要为控制块分配内存。

如果使用的是std::make_shared,

```
auto spw = std::make_shared<Widget>();
```

一次分配足够了。这是因为std::make_shared分配了一整块空间，包含了Widget对象和控制块。这个优化减少了程序的静态大小，因为代码中只包含了一次分配调用，并且加快了代码的执行速度，因为内存只被分配一次。此外，使用std::make_shared避免了在控制块中额外添加的一些记录信息的需要，潜在的减少了程序所需的总内存消耗。

上文的 `std::make_shared` 效率分析同样使用于std::allocate_shared，所以std::make_shared的性能优点也可以延伸到std::allocate_shared函数。

上面说了这么多偏爱make函数，而不是直接用new的理由，每一个都理直气壮。但是，抛开什么软件工程，异常安全和性能的优点，这个Item教程的目的是偏向make函数，并不是要我们完全依赖它们。这是因为有一些情况下，make函数不能或者不应该被使用。

例如，make函数都不支持指定自定义的deleter(请看Item18和Item19).但是std::unique_ptr以及std::shared_ptr都有构造函数来支持这样做。比如，给定一个Widget的自定义deleter

```
auto  widgetDeleter = [](Widget* pw){...};
```

直接使用new创建一个智能指针来直接使用它

```
std::unique_ptr<Widget, decltype(widgetDeleter)> upw(new Widget, widgetDeleter);
std::shared_ptr<Widget> spw(new Widget, widgetDeleter);
```

用make函数可做不了这种事情。

make函数的第二个限制来自于它们实现的句法细节。Item 7解释了当创建了一个对象，该对象的类型重载了是否以std::initializer_list为参数的两种构造函数，使用大括号的方式来构造对象偏向于使用以std::initializer_list为参数的构造函数。而使用括号来构造对象偏向于调用非std::initializer_list的构造函数。make函数完美转发它的参数给对象的构造函数，但是，它使用的是括号还是大括号方式呢？对于某些类型，这个问题的答案产生的结果大有不同。举个例子，在下面的调用中：

```
auto upv = std::make_unique<std::vector<int>>(10,20)
auto spv = std::make_shared<std::vector<int>>(10,20);
```

产生的智能指针所指向的std::vector是拥有10个元素，每个元素的值都是20，还是拥有两个值，分别是10和20？或者说结果是不确定性的？

好消息是结果是确定性的：两个调用都产生了同样的std::vector:拥有10个元素，每个元素的值被设置成了20.这就意味着在make函数中，完美转发使用的是括号而非大括号格式。坏消息是如果你想要使用大括号格式来构造指向的对象，你必须直接使用new.使用make函数需要完美转发大括号initializer的能力，但是，正如Item 30所说的那样，大括号initializer是没有办法

完美转发的。但是，Item 30同时描述了一个变通方案：使用auto类型推导从大括号initializer(请看Item 2)中来创建一个std::initializer_list对象，然后将auto创建出来的对象传递给make函数：

```
//使用std::initializer_list创建
auto initList = {10, 20};
//使用std::initializer_list为参数的构造函数来创建std::vector
auto spv = std::make_shared<std::vector<int>>(initList);
```

对于std::unique_ptr,这里只是存在两个场景(自定义的deleter以及大括号initializer)make函数不适用。但对于std::shared_ptr来说，问题可不止两个了。还有另外两个，但是都可称之为边缘情况，但确实有些程序员会处于这种边缘情况，你也有可能会碰到。

一些对象定义它们自己的new和deleter操作符。这些函数的存在暗示了为这种类型的对象准备的全局的内存分配和回收方法不再适用。通常情况下，这种自定义的new和delete都被设计为只分配或销毁恰好是一个属于该类的对象大小的内存，例如，Widget的new和deleter操作符经常被设计为：只是处理大小就是sizeof(Widget)的内存块的分配和回收。而std::shared_ptr支持的自定义的分配(通过std::allocate_shared)以及回收(通过自定义的deleter)的特性，上文描述的过程就支持的不好了，因为std::allocate_shared所分配的内存大小不仅仅是动态分配对象的大小，它所分配的大小等于对象的大小加上一个控制块的大小。所以，使用make函数创建的对象类型如果包含了此类版本的new以及delete操作符，此时(使用make)确实是个坏主意。

使用std::make_shared相对于直接使用new的大小及性能优点源自于：std::shared_ptr的控制块是和被管理的对象放在同一个内存区块中。当该对象的引用计数变成了0，该对象被销毁（析构函数被调用）。但是，它所占用的内存直到控制块被销毁才能被释放，因为被动态分配的内存块同时包含了两者。

我之前提到过，控制块除了它自己的引用计数，还记录了一些其它的信息。引用计数记录了多少个std::shared_ptr引用了当前的控制块，但控制块还包含了第二个引用计数，记录了多少哥std::weak_ptr引用了当前的控制块。第二个引用计数被称之为weak count（备注：在实际情况中，weak count不总是和引用控制块的std::weak_ptr的个数相等，库的实现往weak count添加了额外的信息来生成更好的代码(facilitate better code generation).但为了本Item的目的，我们忽略这个事实，假设它们是相等的）.当std::weak_ptr检查它是否过期(请看Item 19)时,它看看它所引用的控制块中的引用计数(不是weak count)是否是0(即是否还有std::shared_ptr指向被引用的对象，该对象是否因为引用为0被析构)，如果是0，std::weak_ptr就过期了，否则反之。

只要有一个std::weak_ptr还引用者控制块(即，weak count大于0)，控制块就会继续存在，包含控制块的内存就不会被回收。被std::shared_ptr的make函数分配的内存，直至指向它的最后一个std::shared_ptr和最后一个std::weak_ptr都被销毁时，才会得到回收。

当类型的对象很大，而且最后一个std::shared_ptr的析构于最后一个std::weak_ptr析构之间的间隔时间很大时，该对象被析构与它所占用的内存被回收之间也会产生间隔：

```
class ReallyBigType{...};
auto pBigObj = std::make_shared<ReallyBigType>();
//使用std::make_shared来创建了一个很大的对象

... //创建了一些std::shared_ptr和std::weak_ptr来指向那个大对象

... //最后一个指向对象的std::shared_ptr被销毁了
//但是仍有指向它的std::weak_ptr存在

...//在这段时间内，之前为大对象分配的内存仍未被回收

...//最后一个指向该对象的std::weak_ptr在次被析构了；控制块和对象的内存也在此释放
```

如果直接使用了new，一旦指向ReallyBigType的最后一个std::shared_ptr被销毁，对象所占的内存马上得到回收.（本质上使用了new,控制块和动态分配的对象所处的内存不在一起，可以单独回收）

```
class ReallyBigType{...}; //as before
std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);//使用new创建了一个大对象

... //就像之前那样，创建一些std::shared_ptr和std::weak_ptr指向该对象。

... //最后一个指向对象的std::shared_ptr被销毁了
//但是仍有指向它的std::weak_ptr存在
//但是该对象的内存在此也会被回收

...//在这段时间内，只有为控制块分配的内存未被回收

...//最后一个指向该对象的std::weak_ptr在次被析构了；控制块的内存也在此释放
```

你发现自己处于一个使用std::make_shared不是很可行甚至是不可能的境地，你想到了之前我们提到的异常安全的问题。实际上直接使用new时，只要保证你在一句代码中，只做了将new的结果传递给一个智能指针的构造函数，没有做其它事情。这也会阻止编译器在new的使用和调用用来管理new的对象的智能指针的构造函数之间，插入可能会抛出异常的代码。

举个栗子，对于我们之间检查的那个异常不安全的processWidget函数，我们在之上做个微小的修订。这次，我们指定一个自定的deleter:

```
void processWidget(std::shared_ptr<Widget> spw,
                   int priority); //as before
void cusDel(Widget *ptr);//自定义的deleter
```

这里有一个异常不安全的调用方式：

```
processWidget(std::shared_ptr<Widget>(new Widget,cusDel),
computePriority())//as before,可能会造成内存泄露
```

回想：如果computerPriority在"new Widget"之后调用，但是在std::shared_ptr构造函数执行之前，并且如果computePriority抛出了一个异常，那么动态分配的Widget会被泄露。

在此我们使用了自定义的deleter,所以就不能使用std::make_shared了，想要避免这个问题，我们就得把Widget的动态分配以及std::shared_ptr的构造单独放到一句代码中，然后以该句代码得到的std::shared_ptr来调用std::shared_ptr.这就是技术的本质，尽管过会儿你会看到我们对此稍加改进来提升性能。

```
std::shared_ptr<Widget> spw(new Widget, cusDel);
processWidget(spw, computePriority());//正确的，但不是最优的：看下面
```

确实可行，因为即使构造函数抛出异常，std::shared_ptr也已经接收了传给它的构造函数的原生指针的所有权.在本例中，如果spw的构造函数抛出异常(例如，假如因为无力去给控制块动态分配内存)，它依然可以保证cusDel可以在“new Widget”产生的指针上面调用。

在异常非安全的调用中，我们传递了一个右值给processWidget，

```
processWidget(std::shared_ptr<Widget>(new Widget, cusDel),    //arg是一个右值
computePriority());
```

而在异常安全的调用中，我们传递了一个左值：

```
processWidget(spw, computePriority());//arg是一个左值
```

这就是造成性能问题的原因。

因为processWidget的std::shared_ptr参数按值传递，从右值构造只需要一个move，然而从左值构造却需要一个copy操作。对于std::shared_ptr来说，区别是显著的，因为copy一个std::shared_ptr需要对它的引用计数进行原子加1，然后move一个std::shared_ptr不需要对引用计数做任何操作。对于异常安全的代码来说，若想获得和非异常安全代码一样的性能表现，我们需要对spw用std::move,把它转化成一个右值(看Item 23)：

```
processWidget(std::move(spw), computePriority());
//即异常安全又获得了效率
```

是不是很有趣，值得一看。但是这种情况不是很常见。因为你也很少有原因不使用make函数。如果不是非要用其他的方式不可，我还是推荐你尽量使用make函数。

要记住的东西
和直接使用new相比，使用make函数减少了代码的重复量，提升了异常安全度，并且，对于std::make_shared以及std::allocate_shared来说，产生的代码更加简洁快速
也会存在使用make函数不合适的场景：包含指定自定义的deleter,以及传递大括号initializer的需要
对于std::shared_ptr来说，使用make函数的额外的不使用场景还包含(1)带有自定义内存管理的class(2)内存非常紧俏的系统，非常大的对象以及比对应的std::shared_ptr活的还要长的std::weak_ptr

Item 22: 当使用Pimpl的时候在实现文件中定义特殊的成员函数

如果你曾经因为程序过多的build次数头疼过，你肯定对于Pimpl(pointer to implementation)做法很熟悉。它的做法是：把对象的成员变量替换为一个指向已经实现类(或者是结构体)的指针.将曾经在主类中的数据成员转移到该实现类中，通过指针来间接的访问这些数据成员。举个列子，假设Widget看起来像是这样：

```
class Widget{            //in header "widget.h"
public:
    Widget();
    ...
private:
    std::string name;
    std::vector<double> data:
    Gadget g1,g2,g3;
    //Gadget is some user-defined  type
}
```

因为Widget的数据成员是std::string, std::vector以及Gadget类型，为了编译Widget,这些类型的头文件必须被包含进来,使用Widget的客户必须#include ,,以及gadget.h。这些头文件增加了使用Widget的客户的编译时间，并且让客户依赖这些头文件的内容。如果一个头文件里的内容发生了改变，使用Widget的客户必须被重新编译。虽然标准的头文件和不怎么发生变化，但gadget.h的头文件可能经常变化。

应用C++ 98的Pimpl做法，我们将数据成员变量替换为一个原生指针，指向了一个只是声明并没有被定义的结构体

```
class Widget{            //still in header "widget.h"
public:
    Widget();
    ~Widget();    //dtor is needed-see below
    ...
private:
    struct Impl;        //declare implementation struct
    Impl *pImpl;        //and pointer to it
}
```

Widget已经不再引用std::string,std::vector以及gadget类型，使用Widget的客户就可以不#include这些头文件了。这加快了编译速度，并且Widget的客户也不受到影响。

一种只声明不定义的类型被称作incomplete type.Widget::Impl就是incomplete type。对于incomplete type,我们能做的事情很少，但是我们可以声明一个指向它的指针。Pimpl做法就是利用了这一点。

Pimpl做法的第一步是：声明一个成员变量，它是一个指向incomplete type的指针。第二步是，为包含以前在原始类中的数据成员的对象(本例中的*pImpl)做动态分配内存和回收内存。分配以及回收的代码在实现文件中。本例中，对于Widget而言,这些操作在widget.cpp中进行：

```
#include "widget.h"            //in impl,file "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl{
    std::string name;           //definition of Widget::Impl with data members formerly in Widget
    std::vector<double> data;
    Gadget g1,g2,g3;
}

Widget::Widget():pImpl(new Impl)    //allocate data members for this Widget object
{}

Widget::~Widget()    //destroy data members for this object
{
    delete pImpl;
}
```

在上面的代码中，我还是使用了#include指令，表明对于std::string,std::vector以及Gadget头文件的依赖还继续存在。然后，这些依赖从widget.h(被使用Widget的客户所使用，对它们可见)转移到了widget.cpp(只对Widget的实现者可见)中，我已经高亮了(不好意思，在代码中高亮这种语法俺做不到啊---译者注)分配和回收Impl对象的代码。因为需要在Widget析构是，对Impl对象的内存进行回收，所以Widget的析构函数是必须要写的。

但是我给你展示的是C++ 98的代码，散发着上个世纪的上古气息.使用了原生指针，原生的new和原生的delete，全都是原生的啊！本章的内容是围绕着“智能指针大法好,退原生指针保平安”的理念。如果我们想要在一个Widget的构造函数中动态分配一个Widget::Impl对象，并且在Widget析构时，也析构Widget::Impl对象。那么std::unique_ptr(请看Item 18)就是我们想要的最合适的工具。将原生pImpl指针替换为std::unique_ptr。头文件的内容变为这样：

```
class Widget{
public:
    Widget();
    ...
private:
    struct Impl;
    std::unique_ptr<Impl> pImpl;//use smart pointer
    //instead of raw pointer
}
```

实现文件的内容则变成了这样：

```
#include "widget.h"            //in "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl{
    std::string name;            //as before
    std::vector<double> data;
    Gadget g1,g2,g3;
}

Widget::Widget()
:pImpl(std::make_unique<Impl>())    //per Item 21,create
{}                                        //std::unique_ptr
                                       //via std::make_unique
```

你会发现Widget的析构函数不复存在。这是因为我们不需要在析构函数里面写任何代码了。std::unique_ptr在自身销毁时自动析构它指向的区域，所以我们不需要自己回收任何东西。这就是智能指针的一项优点：它们消除了我们需要手动释放资源的麻烦。

但是呢，使用Widget的客户的一句很平凡的用法，就编译出错了啊

```
#include "widget.h"
Widget w; //error
```

你所受到的错误信息内容依赖于你所使用的编译器类型，但是产生的内容大致都是：在incomplete type上使用了sizeof和delete.这些操作在该类型上是禁止的。

Pimpl做法结合std::unique_ptr竟然会产生错误，这很让人震惊啊。因为(1)std::unique_ptr自身标榜支持incomplete type.(2)Pimpl做法是std::unique_ptr众多的使用场景之一。幸运的是，让代码工作起来也是很简单地。我们首先需要理解为啥会出错。

在执行w被析构(如当出作用域时)的代码时，报了错误。在此时，Widget的析构函数被调用。在定义使用std::unique_ptr的Widget的时候，我们并没有声明析构函数，因为我们不需要在Widget的析构函数内写任何代码。依据编译器自动生成特殊成员函数(请看Item 17)的普通规则，编译器为我们生成了一个析构函数。在那个自动生成的析构函数中，编译器插入代码，调用Widget的数据成员pImpl的析构函数。pImpl是一个 `std::unique_ptr<Widget::Impl>` ,即，一个使用默认deleter的std::unique_ptr.默认deleter是一个函数，对std::unique_ptr里面的原生指针调用delete.然而，在调用delete之前，编译器通常会让默认deleter先使用C++ 11的static_assert来确保原生指针指向的类型不是imcomplete type(staticassert编译时候检查,assert运行时检查---译者注).当编译器生成Widget w的析构函数时，调用的static_assert检查就会失败，导致出现了错误信息。在w被销毁时，这些错误信息才会出现，但是因为与其他的编译器生成的特殊成员函数相同，Widget的析构函数也是inline的。出错指向w被创建的那一行，因为改行创建了w,导致后来(w出作用域时)w被隐性销毁。

为了修复这个问题，你需要确保，在产生销毁 `std::unique_ptr<Widget::Impl>` 的代码时，Widget::Impl是完整的类型。当它的定义被编译器看到时，它就是完整类型了。而Widget::Impl在widget.cpp中被定义。所以编译成功的关键在于，让编译器只在widget.cpp内，在widget::Impl被定义之后，看到Widget的析构函数体(该函数体就是放置编译器自动生成销毁std::unique_ptr数据成员的代码的地方)。

像那样安排很简单，在widget.h中声明Widget的析构函数，但是不要在其中定义：

```
class Widget{            //as before, in "widget.h"
public:
    Widget();
    ~Widget();        //declaration only
    ...
private:                //as before
    struct Impl;
    std::unique_ptr<Impl> pImpl;
}
```

在widget.cpp里面的Widget::Impl定义之后再定义析构函数：

```
#include "widget.h"            //as before in "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl{
    std::string name;            //as before definition of
    std::vector<double> data;    //Widget::Impl
    Gadget g1,g2,g3;
}

Widget::Widget()
:pImpl(std::make_unique<Impl>())    //as before

Widget::~Widget(){}    //~Widget definition
```

这样的话就没问题了，增加的代码量也很少。但是如果你想要强调，编译器生成的析构函数会做正确的事情，你声明析构函数的唯一原因是，想要在Widget.cpp中生成它的定义，你可以用“=default”定义析构函数体：

```
Widget::~Widget()=default;        //same effect as above
```

使用Pimpl做法的类是自带支持move语义的候选者，因为编译器生成的move操作正是我们想要的：对潜在的std::unique_ptr上执行move操作。就像Item 17解释的那样，Widget声明了析构函数，编译器就不会自动生成move操作了。所以如果你想要支持move,你必须自己去声明这些函数。鉴于编译器生成的版本就可以胜任，你可能会像下面那样实现：

```
class Widget{            //still in "widget.h"
public:
    Widget();
    ~Widget();
    ...
    Widget(Widget&& rhs) = default;        //right idea,
    Widget& operator=(Widget&& rhs) = default //wrong code!
    ...
private:                //as before
    struct Impl;
    std::unique_ptr<Impl> pImpl;
};
```

这样的做法会产生和声明一个没有析构函数的class一样，产生同样的问题，产生问题的原因本质也一样。对于编译器生成的move赋值操作符，它对pImpl再赋值之前，需要先销毁它所指向的对象，然而在Widget头文件中，pImpl指向的仍是一个incomplete type.对于编译器生成的move构造函数。问题在于编译器会在move构造函数内抛出异常的事件中，生成析构pImpl的代码，对pImpl析构(destroying pImpl)需要Impl的类型是完整的。

问题一样，解决方法自然也一样：将move操作的定义写在实现文件widget.cpp中： widget.h:

```
class Widget{            //still in "widget.h"
public:
    Widget();
    ~Widget();
    ...
    Widget(Widget&& rhs);               //declarations
    Widget& operator=(Widget&& rhs); //only
    ...
private:                //as before
    struct Impl;
    std::unique_ptr<Impl> pImpl;
};
```

widget.cpp

```
#include <string>            //as before,
...                                 //in "widget.cpp"
sturct:Widget::Impl {...}; //as before

Widget::Widget()                //as before
:pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default; //as before

Widget::Widget(Widget&& rhs) = default;
Widget& Widget::operator=(Widget&& rhs) = default:
//definitions
```

Pimpl做法是一种减少class的实现和class的使用之间编译依赖的一种方式，但是，从概念上来讲，这种做法并不改变类的表现方式。原来的Widget类包含了std::string,std::vector以及Gadget数据成员，并且，假设Gadget，像std::string和std::vector那样，可以被拷贝。所以按理说Widget也要支持拷贝操作。我们必须要自己手写这些拷贝函数了，因为(1)对于带有move-only类型(像std::unique_ptr)的类，编译器不会生成拷贝操作的代码.(2)即使生成了，生成的代码只会拷贝std::unique_ptr(即，执行浅拷贝)，而我们想要的是拷贝指针所指向的资源(即，执行深拷贝)。

现在的做法我们已经熟悉了，在头文件中声明这些函数，然后在实现文件中实现这些函数；

widget.h:

```
class Widget{            //still in "widget.h"
public:
    ...                    //other funcs, as before
    Widget(const Widget& rhs);               //declarations
    Widget& operator=(const Widget& rhs); //only
private:                //as before
    struct Impl;
    std::unique_ptr<Impl> pImpl;
};
```

widget.cpp

```
#include <string>            //as before,
...                                 //in "widget.cpp"
sturct:Widget::Impl {...}; //as before

Widget::~Widget() = default; //other funcs,as before

Widget::Widget(const Widget& rhs); //copy ctor
: pImpl(std::make_unique<Impl>(*rhs.pImpl))
{}

Widget& Widget::operator=(const Widget& rhs)//copy operator=
{
    *pImpl = *rhs.pImpl;
    return *this;
}
```

两个函数的实现都比较常见。每种情况下，我们都是从源对象(rhs)到目的对象(*this)，简单的拷贝了Impl的结构体的内容.我们利用了这样的事实：编译器会为Impl生成拷贝操作的代码，这些操作会自动将stuct的内容逐项拷贝，就不需要我们手动来做了。我们因此通过调用Widget::Impl的编译器生成的拷贝操作符来实现了Widget拷贝操作符。在copy构造函数中，注意到我们遵循了Item 21的建议，不直接使用new，而是优先使用了std::make_unique.

在上面的例子中，为了实现Pimpl做法，std::unique_ptr是我们使用的智能指针类型，因为对象内(在Widget内)的pImpl对对应的实现对象(Widget::Impl对象)拥有独占所有权。但是，很有意思的是，当我们对pImpl使用std::shared_ptr来替代std::unique_ptr,我们发现本Item的建议不再适用了。没必要在Widget.h中声明析构函数，没有了用户自己声明的析构函数，编译器会很乐意生成move操作代码，而且生成的代码表现的行为正合我们意。widget.h变得如下所示：

```
class Widget{                    //in "widget.h"
public:
    Widget();
    ...                          //no declarations for dtor
                                 //or move operations
private:
    struct Impl;                         //std::shared_ptr
    std::shared_ptr<Impl> pImpl;    //instead of std::unique_ptr
};
```

`#include widget.h` 的客户代码：

```
Widget w1;
auto w2(std::move(w1));            //move-consturct w2

w1 = std::move(w2);                //move-assign w1
```

所有代码都会如我们所愿通过编译，w1会被默认构造，它的值会被move到w2,之后w2的值又被move回w1.最后w1和w2都得到析构(这使得指向的Widget::Impl对象被析构)

对pImpl应用std::unique_ptr和std::shared_ptr的表现行为不同的原因是：它们之间支持自定义的deleter的方式不同。对于std::unique_ptr，deleter的类型是智能指针的一部分，这就使得编译器生成更小的运行时数据结构以及更快的运行时代码成为可能。更好的效率的结果是要求当编译器生成特殊函数(如析构以及move操作)被使用时，std::unique_ptr所指向的类型必须是完整的。对于std::shared_ptr来说，deleter的类型不是智能指针的一部分。虽然会造成比较大的运行时数据结构和慢一些的代码。但是在调用编译器生成的特殊函数时，指向的类型不需要是完整的。

对于Pimpl做法，在 `std::unique_ptr` 和 `std::shared_ptr` 的特点之间，其实并没有一个真正的权衡。因为Widget和Widget::Impl之间是独占的拥有关系， `std::unique_ptr` 在此项工作中很合适。然而，在一些其他的场景中，共享式拥有关系存在， `std::shared_ptr` 才是一个合适的选择，就没必要像依靠 `std::unique_ptr` 这样的函数定义的做法了。

要记住的东西
Pimpl做法通过减少类的实现和类的使用之间的编译依赖减少了build次数
对于 `std::unique_ptr` pImpl指针，在class的头文件中声明这些特殊的成员函数，在class的实现文件中定义它们。即使默认的实现方式(编译器生成的方式)可以胜任也要这么做
上述建议适用于 `std::unique_ptr` ,对 `std::shared_ptr` 无用

当你第一次学习move语义和完美转发时，它们看起来很直截了当：

- **Move**语义使编译器能够把昂贵的拷贝操作替换为代价较小的move操作。 和拷贝构造函数以及拷贝赋值运算符能赋予你控制拷贝对象的能力一样，move构造函数以及move赋值运算符提供给你对move语义的控制。Move语义使得move-only类型的创建成为可能，比如说 `std::unique_ptr` ， `std::future` 以及 `std::thread` .
- 完美转发让我们可以写出接受任意参数的函数模板，并且将之转发到其他的函数，使得target函数接受的参数和forwarding函数接受的参数相同。

右值引用对于这两个看起来毫无关系的概念来说，就像是粘合两者的胶水。它作为潜在的语言机制，为move语义和完美转发的实现提供支持。

你对这些特性越有经验,你就越发现你对它们的第一印象就像是刚刚发现了冰山一角。move语义，完美转发以及右值引用跟它们看起来比有细微差别，比如说，move语义并不move任何东西，完美转发是不完美的。move操作的代价并不总是比拷贝低；就算当它们确实代价底时，也没有达到你想象的低的程度；它也并不总是在move有效的上下文中被调用。结构 `type&&` 并不一定总是代表一个右值引用。

不管你怎么去探索这些特性，看起来它们总是还有一些你还没注意到的地方。幸运的是，它们的知识不是永无止境的。本章会带你直达基础。看完本章节，C++ 11的这部分内容对你来说就变得栩栩如生。比如说，你就会知道std::move和std::forward的常见用法，带有迷惑性的 `type&&` 用法对你来说变得很平常。你也会理解move操作的各种让人感到奇怪的表现的原因.这些都会水到渠成。到那时，你又会回到了起点，因为move语义，完美转发，以及右值引用又一次看起来是那么的直截了当。但这次，它们(直截了当)的状态会一直保持下去。

在本章的所有Item中，你必须要牢记一点，作为函数的参数，永远是一个左值，即使它(在函数的参数列表中)的类型是一个右值引用。例如：

```
void f(Widget&& w);
```

参数w是一个左值，即使它的类型是一个对Widget的右值引用。(如果你对此感到不理解，请重新回顾一下第2页(原文的页码 --不负责任的译者说)所讲的左值与右值的概览内容)

Item 23:Understand std::move and std::forward

首先通过了解它们(指std::move和std::forward)不做什么来认识std::move和std::forward是非常有用的。std::move不move任何东西。std::forward也不转发任何东西。在运行时，他们什么都不做。不产生可执行代码，一个比特/Users/shikunfeng/Documents/neteaseWork/timeline_15_05_18/src/main/webapp/tmpl/web2/widget/event2.ftl的代码也不产生。

std::move和std::forward只是执行转换的函数(确切的说应该是函数模板)。std::move无条件的将它的参数转换成一个右值，而std::forward当特定的条件满足时，才会执行它的转换。这就是它们本来的样子.这样的解释产生了一些新问题，但是，基本上，就是这么一回事。

为了让这个故事显得更加具体，下面是C++ 11的std::move的一种实现样例，虽然不能完全符合标准的细节，但也非常相近了。

```
template<typename T>
typename remove_reference<T>::type&&
move(T&& param)
{
    using ReturnType =                          //alias declaration;
    typename remove_reference<T>::type&&;//see Item 9
    return static_cast<ReturnType>(param);
}
```

我为你高亮的两处代码(我做不到啊!--菜b的译者注)。首先是函数的名字move，因为返回的类型非常具有迷惑性，我可不想让你一开始就晕头转向。另外一处是最后的转换，包含了move函数的本质。正如你所看到的，std::move接受了一个对象的引用做参数(准确的来说，应该是一个universal reference.请看Item 24。这个参数的格式是T&& param，但是请不要误解为move接受的参数类型就是右值引用，请继续往下看----菜b译者注)，并且返回指向同一个对象的引用。

函数返回值的"&&"部分表明std::move返回的是一个右值引用。但是呢，正如Item 28条解释的那样，如果T的类型恰好是一个左值引用，T&&的类型就会也会是左值引用。为了阻止这种事情的发生，我们用到了type trait(请看Item 9),在T上面应用std::remove_reference，它的效果就是“去除”T身上的引用，因此保证了"&&"应用到了一个非引用的类型上面。这就确保了std::move真正的返回的是一个右值引用(rvalue reference)，这很重要，因为函数返回的rvalue reference就是右值(rvalue).因此，std::move就做了一件事情：将它的参数转换成了右值(rvalue).

说一句题外话，std::move可以更优雅的在C++14中实现。感谢返回函数类型推导(function return type deduction 请看Item 3),感谢标准库模板别名(alias template) `std::remove_reference_t` (请看Item 9), `std::move` 可以这样写：

```
template<typename T>                //C++14; still in
decltype(auto) move(T && param)    //namespace std
{
    using ReturnType = remove_reference_t<T>&&;
    return static_cast<ReturnType>(param);
}
```

看起来舒服多了，不是吗？

因为std::move除了将它的参数转换成右值外什么都不做，所以有人说应该给它换个名字，比如说叫 `rvalue_cast` 可能会好些。话虽如此，它现在的名字仍然就是 `std::move` .所以记住 `std::move` 做什么不做什么很重要。它只作转换，不做move.

当然了，rvalues是对之执行move的合格候选者，所以对一个对象应用std::move告诉编译器，该对象很合适对之执行move操作，所以std::move的名字就有意义了：标示出那些可以对之执行move的对象。

事实上，rvalues并不总是对之执行move的合格候选者。假设你正在写一个类，它用来表示注释。此类的构造函数接受一个包含注释的std::string做参数，并且将此参数的值拷贝到一个数据成员上.受到Item 41的影响，你声明一个接收by-value参数的构造函数:

```
class Annotation{
    public:
        explicit Annotation(std::string text);//param to be copied,
        ...            //so per Item 41, pass by value
};
```

但是Annotation的构造函数只需要读取text的值。并不需要修改它。根据一个历史悠久的传统:能使用const的时候尽量使用。你修改了构造函数的声明，text改为const：

```
class Annotation{
    public:
        explicit Annotation(const std::string text);//param to be copied,
        ...            //so per Item 41, pass by value
};
```

为了避免拷贝text到对象成员变量带来拷贝代价。你继续忠实于Item 41的建议，对text应用std::move,因此产生出一个rvalue:

```
class Annotation{
    public:
        explicit Annotation(const std::string text)
        : value(std::move(text))//"move" text into value; this code
        {...}            //doesn't do what it seems to!
        ...

    private:
        std::string value;
};
```

这样的代码通过了编译，链接，最后运行。而且把成员变量value设置成text的值。代码跟你想象中的完美情况唯一不同的一点是，它没有对text执行move到value，而是拷贝了text的值到value.text确实被std::move转化成了rvalue，但是text被声明为const std::string.所以在cast之前，text是一个const std::string类型的lvalue.cast的结果是一个const std::string的rvalue,但是自始至终，const的性质一直没变。

代码运行时，编译器要选择一个std::string的构造函数来调用。有以下两种可能:

```
class string{                //std::string is actually a
    public:                 //typedef for std::basic_string<char>
        ...
        string(const string& rhs);        //copy ctor
        string(string&& rhs);             //move ctor
};
```

在Annotation的构造函数的成员初始化列表(member initialization list), `std::move(text)` 的结果是const std::string的rvalue.这个rvalue不能传递给std::string的move构造函数,因为move构造函数接收的是非const的std::string的rvalue引用。然而，因为lvalue-reference-to-const的参数类型可以被const rvalue匹配上，所以rvalue可以被传递给拷贝构造函数.因此即使text被转换成了rvalue，上文中的成员初始化仍调用了std::string的拷贝构造函数!这样的行为对于保持const的正确性是必须的。从一个对象里move出一个值通常会改变这个对象，所以语言不允许将const对象传递给像move constructor这样的会改变次对象的函数。

从本例中你可以学到两点。首先，如果你想对这些对象执行move操作，就不要把它们声明为const.对const对象的move请求通常会悄悄的执行到copy操作上。

std::forward的情况和std::move类似，但是和std::move无条件地将它的参数转化为rvalue不同,std::forward在特定的条件下才会执行转化。std::forward是一个有条件的转化。为了理解它何时转化何时不转化，我们来回想一下std::forward的典型的使用场景。最常见的场景是：一个函数模板(function template)接受一个universal reference参数，将它传递给另外一个函数(作参数):

```
void process(const Widget& lvalArg);    //process lvalues
void process(Widget&& rvalArg);    //process rvalues

template<typename T>
void logAndProcess(T&& param)        //template that passes
                                        //param to process
{
    auto now = std::chrono::system_clock::now(); //get current time
    makeLogEntry("Calling 'process'", now);
    process(std::forward<T>(param));
}
```

请看下面对logAndProcess的两个调用，一个使用的lvalue,另一个使用的rvalue:

```
Widget w;
logAndProcess(w);    //call with lvalue
logAndProcess(std::move(w));    //call with rvalue
```

在logAndProcess的实现中，参数param被传递给了函数process.process按照参数类型是lvalue或者rvalue都做了重载。当我们用lvalue调用logAndProcess时，我们自然地期望：forward给process的也是一个lvalue,当我们用rvalue来调用logAndProcess时，我们希望process的rvalue重载版本被调用。

但是就像所有函数的参数一样，param可能是一个lvalue.logAndProcess内的每一个对process的调用因此想要调用process的lvalue重载版本。为了让以上代码的行为表现正确，我们需要一个机制，param转化为rvalue当且仅当：传递给logAndProcess的用来初始化param的参数必须是一个rvalue.这正是std::forward做的事情。这就是为什么std::forward被称作是一个条件转化(conditional cast)：当参数被rvalue初始化时，才将参数转化为rvalue.

你可能想知道std::forward怎么知道它的参数是否被一个rvalue初始化。比如说，在以上的代码中，std::forward怎么知道param被一个lvalue或者rvalue初始化？答案很简单，这个信息蕴涵在logAndProcess的模板参数T中。这个参数传递给了std::forward，然后std::forward来从中解码出此信息。欲知详情，请参考Item 28。

std::move和std::forward都可以归之为cast.唯一的一点不同是，std::move总是在执行casts，而std::forward是在某些条件满足时才做。你可能觉得我们不用std::move,只使用std::forward会不会好一些。从一个纯粹是技术的角度来说，答案是肯定的：std::forward是可以都做了，std::move不是必须的。当然，可以说这两个函数都不是必须的，因为我们可以在任何地方都直接写cast代码，但是我希望我们在此达成共识：这样做很恶心。

std::move的魅力在于:方便，减少了错误的概率，而且更加简洁。举个栗子，有这样的一个class,我们想要跟踪，它的move构造函数被调用了多少次，我们这次需要的是一个static的counter，它在每次move构造函数被调用时递增。假设该class还有一个std::string类型的非静态成员，下面是一个实现move constructor(使用std::move)的常见的例子：

```
class Widget{
public:
    Widget(Widget&& rhs)
    : s(std::move(rhs.s))
    { ++moveCtorCalls; }
    ...
private:
    static std::size_t moveCtorCalls;
    std::string s;
}
```

如果要使用std::forward来实现同样的行为，代码像下面这样写：

```
class Widget{
public:
    Widget(Widget&& rhs)                //unconventional,
    : s(std::forward<std::string>(rhs.s)) //undesirable
    { ++moveCtorCalls; }                //implementation

    ...
}
```

请注意到：首先，std::move只需要一个函数参数(rhs.s), std::forward不只需要一个函数参数(rhs.s),还需要一个模板类型参数(std::string).然后,注意到我们传递给std::forward的类型是非引用类型(non-reference)，因为这就意味着传递的那个参数是一个rvalue（请看Item 28）。综上，这就意味着std::move比std::forward用起来更方便(至少少敲了不少字),免去了让我们传递一个表示函数参数是否是一个rvalue的类型参数。消除了传递错误类型(比如说,传一个std::string&,可以导致数据成员s被拷贝构造，而不是想要的move构造)的可能性。

更重要的是，std::move的使用表明了对rvalue的无条件的转换，然而，当std::forward只对被绑定了rvalue的reference进行转换。这是两个非常不同的行为。std::move就是为了move操作而生，而std::forward,就是将一个对象转发(或者说传递)给另外一个函数，同时保留此对象的左值性或右值性(lvalueness or rvalueness)。所以我们需要这两个不同的函数(并且是不同的函数名字)来区分这两个操作。

要记住的东西
std::move执行一个无条件的对rvalue的转化。对于它自己本身来说，它不会move任何东西
std::forward在参数被绑定为rvalue的情况下才会将它转化为rvalue
std::move和std::forward在runtime时啥都不做

Item 24: Distinguish universal references from rvalue references.

大多数人说真相可以让我们感到自由，但是在某些情况下，一个巧妙的谎言也可以让人觉得非常轻松。这个Item就是要编制一个“谎言”。因为我们是在和软件打交道。所以我们避开“谎言”这个词：我们是在编制一种“抽象”的意境。

为了声明一个类型T的右值引用，你写下了T&&。下面的假设看起来合理：你在代码中看到了一个"T&&"时，你看到的就是一个右值引用。但是，它可没有想象中那么简单：

```
void f(Widget&& param);            //rvalue reference
Widget&& var1 = Widget();        //rvalue reference
auto&& var2 = var1;                //not rvalue reference

template<typename T>
void f(std::vector<T>&& param)        //rvalue reference

template<typename T>
void f(T&& param);                //not rvalue reference
```

实际上，“T&&”有两个不同的意思。首先,当然是作为rvalue reference,这样的引用表现起来和你预期一致：只和rvalue做绑定，它们存在的意义就是表示出可以从中move from的对象。

“T&&”的另外一个含义是：既可以是rvalue reference也可以是lvalue reference。这样的references在代码中看起来像是rvalue reference（即"T&&"）,但是它们也可以表现得就像他们是lvalue refernces（即"T&"）那样.它们的dual nature允许他们既可以绑定在rvalues(like rvalue references)也可以绑定在lvalues(like lvalue references)上。进一步来说，它们可以绑定到const或者non-const,volatile或者non-volatile，甚至是const + volatile对象上面。它们几乎可以绑定到任何东西上面。为了对得起它的全能，我决定给它们起个名字：universal reference.(Item25将会解释universal references总是可以将std::forward应用在它们之上，本书出版之时，C++委员会的一些人开始将universal references称之为forward references).

两种上下文中会出现universal references。最普通的一种是function template parameters，就像上面的代码所描述的例子那样:

```
template<typename T>
void f(T&& param);              //param is a universal reference
```

第二种context就是auto的声明方式，如下所示：

```
auto&& var2 = var1;             //var2 is a universal reference
```

这两种context的共同点是:都有type deduction的存在。在template fucntion f中,参数param的类型是被deduce出来的，在var2的声明中，var2的类型也是被deduce出来的。和接下来的例子(也可以和上面的栗子一块儿比)对比我们会发现，下面栗子是不存在type deduction的。如果你看到"T&&",却没有看到type deduction.那么你看到的就是一个rvalue reference:

```
void f(Widget&& param);            //no type deduction
                                    //param is an rvalue reference

Widget&& var1 = Widget();         //no type deduction
                                    //var1 is an rvalue reference
```

因为universal references是references，所以它们必须被初始化。universal reference的initializer决定了它表达的是rvalue reference或者lvalue reference。如果initializer是rvalue，那么universal reference对应的是rvalue reference.如果initializer是lvalue,那么universal reference对应的就是lvalue reference.对于身为函数参数的universal reference，initializer在call site（调用处）被提供：

```
template<typename T>
void f(T&& param);         //param is a universal reference

Widget w;
f(w);                     //lvalue passed to f;param's type is Widget&(i.e., an lvalue
reference)
f(std::move(w));         //rvalue passed to f;param's type is Widget&&(i.e., an rvalue
reference)
```

对universal的reference来说，type deduction是必须的，但还是不够，它要求的格式也很严格，必须是"T&&".再看下我们之前写过的栗子：

```
template<typename T>
void f(std::vector<T>&& param);    //param is an rvalue reference
```

当f被调用时，类型T会被deduce（除非调用者显式的指明类型，这种边缘情况我们不予考虑）。param声明的格式不是T&&，而是std::vector&&.这就说明它不是universal reference,而是一个rvalue reference.如果你传一个lvalue给f，那么编译器肯定就不高兴了。

```
std::vector<int> v;
f(v);                //error! can't bind lvalue to rvalue reference
```

即使一个最简单前缀const.也可以把一个reference成为universal reference的可能抹杀：

```
template<typename T>
void f(const T&& param);        //param is an rvalue reference
```

如果你在一个template里面，并且看到了T&&这样的格式，你可能就会假设它就是一个universal reference.但是并非如此，因为还差一个必要的条件：type deduction.在template里面可不保证一定有type deduction.看个例子，std::vector里面的push_back方法。

```
template<class T, class Allocator = allocator<T>>
class vector{
public:
    void push_back(T&& x);
    ...
}
```

以上便是只有T&&格式却没有type deduction的例子，push_back的存在依赖于一个被instantiation的vector.用于instantiation的type就完全决定了push_back的函数声明。也就是说

```
std::vector<Widget> v;
```

使得std::vector的template被instantiated成为如下格式:

```
class vector<Widget, allocator<Widget>>{
public:
    void push_back(Widget&& x);    //rvalue reference
    ...
};
```

如你所见，push_back没有用到type deduction.所以这个vector的$push_back$（有两个overload的$push_back$）所接受的参数类型是rvalue-reference-to-T.

与之相反，std::vector中概念上相近的$emplace_back$函数确实用到了type deduction:

```
template<class T,class Allocator=allocator<T>>

class vector{
public:
    template <class... Args>
    void emplace_back(Args&&... args);
    ...
};
```

type parameter `Args` 独立于vector的type parameter `T`,所以每次调用 `emplace_back` 的时候， `Args` 就要被deduce一次。（实际上， `Args` 是一个parameter pack.并不是type parameter.但是为了讨论的方便，我们姑且称之为type parameter）。

我之前说universal reference的格式必须是 `T&&`，事实上，emplace_back的type parameter名字命名为Args，但这不影响args是一个universal reference,管它叫做T还是叫做Args呢，没啥区别。举个例子，下面的template接受的参数就是universal reference.一是因为格式是"type&&",二是因为param的type会被deduce(再一次提一下，除非caller显示的指明了type这种边角情况).

```
template<typename MyTemplateType>           //param is a
void someFunc(MyTemplateType&& param); //universal reference
```

我之前提到过auto变量可以是universal references.更准确的说，声明为auto&&的变量就是universal references.因为类型推导发生并且它们也有正确的格式("T&&").auto类型的universal references并不想上面说的那种用来做function template parameters的universal references那么常见，在最近的C++ 11和C++ 14中，它们变得非常活跃。C++ 14中的lambda expression允许声明auto&&的parameters.举个栗子，如果你想写一个C++ 14的lambda来记录任意函数调用花费的时间，你可以这么写:

```
auto timeFuncInvocation =
    [](auto&& func, auto&&... params)        //C++ 14
    {
        start timer;
        std::forward<decltype(func)>(func)(
            std::forward<decltype(params)>(params)... //invoke func on params
        );
        stop timer and record elapsed time
    }
```

如果你对于"std::forward<decltype(blah blah blah)"的代码的反应是“这是什么鬼!？”，这只是意味着你还没看过Item33，不要为此担心。 func是一个可以绑定到任何callable object(lvaue或者rvalue)的universal reference.args是0个或多个universal references(即a universal reference parameter pack)，它可以绑定到任意type，任意数量的objects.所以，多亏了auto universal reference,timeFuncInvocation可以记录pretty much any的函数调用(any和pretty much any的区别，请看Item 30).

本Item中universal reference的基础，其实是一个谎言，呃，或者说是一种"abstraction".隐藏的事实被称之为*reference collapsing*,Item 28会讲述。但是该事实并不会降低它的用途。rvalue references以及universal references会帮助你更准确第阅读source code(“Does that T&& I’m looking at bind to rvalues only or to everything?”),和同事沟通时避免歧义(“I’m using a universal reference here, not an rvalue reference...”),它也会使得你理解Item 25和Item 26,这两条都依赖于此区别。所以，接受理解这个abstraction吧。牛顿三大定律(abstraction)比爱因斯坦的相对论(truth)一样有用且更容易应用，universal reference的概念也可以是你理解reference collapsing 的细节。

要记住的东西
如果一个函数的template parameter有着T&&的格式，且有一个deduce type T.或者一个对象被生命为auto&&,那么这个parameter或者object就是一个universal reference.
如果type的声明的格式不完全是type&&,或者type deduction没有发生，那么type&&表示的是一个rvalue reference.
universal reference如果被rvalue初始化，它就是rvalue reference.如果被lvalue初始化，他就是lvaue reference.