目录

第一章类型推导（Deducing Types）

C++98 包含一个简单的类型推导规则集：函数模板。C++11 对规则集做了一些修改并添加了两条：auto 和 decltype。C++14 又对 auto 和 decltype 的使用进行了扩展。随着类型推导的广泛使用，你可以从一些明显或冗长的类型拼写中解放出来。它使得 C++编写的软件具备更强的适配能力，因为改变在源码一处改变类型，其他地方会自动推导出类型。但它也使得代码理解起来比较困难，因为编译器推导出来的类型不一定是你想要的类型。

如果不能很好的理解类型推导是如何工作的，就不能做到高效的 Modern C++编程。有太多的地方用到了类型推导：如，函数模板的调用、auto 出现的地方、decltype 表达式，还有 C++14 中的 decltype(auto)构造。

本章节描述了每一位 C++开发人员需要了解的类型推导相关信息，解释了模板的类型如何推导，auto 如何推导，decltype 如何实现。同时也告诉你怎么强制让编译器显示类型推导的结果，以便保证编译器推导出你想要的类型。

条款 1：理解模板类型推导

当一个复杂系统的用户可以忽略系统是如何工作的，那就太好了，因为系统如何工作往往是包含了一堆很复杂的设计。基于这个准则，C++的模板类型推导可以说是非常的成功。成千上万的程序员们可以将实参传递给模板函数，并得到正确的结果，尽管他们中的很多人难以描述出那些函数的类型是如何推导出来的。

如果你也是前面说到的那一类人，有好消息也有坏消息。好消息是模板的类型推导是 Modern C++的 auto 特性的基础。如果你对 C++98 的模板类型推导熟悉，那么你就很容易掌握 C++11 的 auto 类型推导。坏消息是模板类型推导的规则应用到 auto 上时，有时候不是那么直观。所以，重要的一点，就是要真正理解那些被 auto 用到的模板类型推导规则。本条款就涵盖了你需要知道的内容。

通用实参

如果你想要大致看一段伪代码的话，请看下面一段函数模板：

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

一种调用方式如下：

```
f(expr); // call f with some expression
```

编译阶段，编译器用 expr 来推导两个类型：T 和 ParamType。这些类型经常是不一样的，因为 ParamType 一般带有限定符，如 const 或引用。例如，如果模板定义如下：

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

并且我们这样调用，

```
int x = 0;
f(x); // call f with an int
```

T 推导出类型为 int，而 ParamType 推导出类型为 const int&。

大家一般很自然的期望 T 推导出来的类型与传递给函数的实参类型一样，即 T 就是 expr 的类型。在上述例子中，就是这样的情况：x 是 int 类型，而 T 推导出的类型也是 int。但是类型推导经常不是这样的情况。T 的类型推导不仅仅依赖于 expr 的类型，还依赖于 ParamType 的格式。有三种情况：

- ParamType 是一个指针或引用类型，但不是一个通用引用。（通用应用在条款 24 描述，在此处，你只要知道通用引用是不同于左值引用或右值引用的。）
- ParamType 是一个通用引用。
- ParamType 既不是指针也不是引用。

因此，我们需要检查三种推导场景，每一种都是基于模板的通用定义格式和它的调用方式：

```
template<typename T>
void f(ParamType param);
f(expr); // deduce T and ParamType from expr
```

情形 1：ParamType 是一个指针或引用类型，但不是一个通用引用

最简单的情况就是，ParamType 是一个引用或指针，但不是通用引用。这种情况下，类型推导按下面进行：

1. 如果 expr 是一个引用，忽略引用部分。
2. 将 expr 的类型和 ParamType 进行模式匹配，来决定 T 的类型。

例如，下面是我们定义的模板，

```
template<typename T>
void f(T& param); // param is a reference
```

我们声明下面这些变量，

```
int x = 27;          // x is an int
const int cx = x;    // cx is a const int
const int& rx = x;   // rx is a reference to x as a const int
```

对以上变量的函数调用，param 和 T 的类型推导如下：

```
f(x);    // T is int, param's type is int&
f(cx);   // T is const int,
         // param's type is const int&
f(rx);   // T is const int,
         // param's type is const int&
```

注意，在第二和第三个调用中，因为 cx 和 rx 指定了 const，T 被推导出 const int，从而得出形参类型为 const int&。这个对调用者很重要。当他们给引用形参传递一个 const 对象时，他们期望对象保持不可更改的特性，即形参一个 reference-to-const 类型。这就是为什么传递一个 const 对象给 T&形参模板是安全的：对象的 const 特性推导变成了 T 类型的一部分。

在第三个例子中，即使 rx 的类型是一个引用，T 也是推导出非引用，那是因为类型推导过程中，rx 的引用特性被忽略了。

这些例子展示的都是左值引用形参，而对右值引用形参，类型推导方式一模一样。当然，只有右值实参才能传递给右值引用形参，但这个限定对类型推导没有影响。

如果我们将 f 的形参由 T&改为 const T&，情况将有一点点改变，但不会太离奇。cx 和 rx 的 const 特性仍然保留，但因为 parma 是一个 reference-to-const，所以不需要再将 const 推导为 T 的一部分。

```
template<typename T>
void f(const T& param);  // param is now a ref-to-const

int x = 27;              // as before
const int cx = x;        // as before
const int& rx = x;       // as before

f(x);                    // T is int, param's type is const int&
f(cx);                   // T is int, param's type is const int&
f(rx);                   // T is int, param's type is const int&
```

跟前面一样，类型推导时，rx 的引用特性被忽略。

如果 param 是一个指针（或一个指向 const 的指针）而不是引用，类型推导本质上还是一样的方式。

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

int x = 27;          // 同上
const int *px = &x;  // px is a ptr to x as a const int


f(&x);               // T is int, param's type is int*
f(px);               // T is const int,param's type is 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，根据写出来的格式就能看出推导的类型的确有点无趣。所有的事情都这么显而易见！这恰恰是你在类型推导系统中所期望的情形。

情形 2：ParamType 是一个通用引用

对于使用通用引用形参的模板，情况就不是那么显而易见了。这些形参声明类似于右值引用（即，在一个类型参数是 T 的函数模板中，一个通用引用形参声明的类型就是 T&&），但当传递一个左值实参进来时，模板的行为是不同的。完整的说明在条款 24，下面只是提要说明：

- 如果 expr 是一个左值，T 和 ParamType 都被推导为左值引用。这有些不同寻常。第一，在模板类型推导中，这个是 T 被推导为引用的唯一情况。第二，尽管 ParamType 声明使用的是右值引用的语法，它的类型推导出来却是一个左值引用。
- 如果 expr 是一个右值，参照**情形 1**。

譬如：

```
template<typename T>
void f(T&& param);   // param是一个通用引用

int x = 27;          // 同上
const int cx = x;    // 同上
const int& rx = x;   // 同上

f(x);           // x是左值，所以T是int&，param类型也是int&
f(cx);          // cx是左值，所以T是const int&，param类型也是const int&
f(rx);          // rx是左值，所以T是const int&，param类型也是const int&
f(27);          // 27是右值，所以T是int，param类型也就是int&&
```

条款 24 准确地解释了为什么这些例子是这样推导的。这儿的关键点是，通用引用形参的类型推导规则与左值引用或右值引用形参是不一样的。特别的是，当使用了通用引用，类型推导要区分左值实参和右值实参，这种情况是不会发生在非通用引用上的。

情形 3：ParamType 既不是指针也不是引用

当 ParamType 既不是指针也不是引用时，我们按传值处理：

```
template<typename T>
void f(T param); // param is now passed by value
```

这就意味着，无论传入的是什么，param 都将生成一个拷贝——一个全新的对象。事实上，基于这一点，可以得出通过 expr 推导出 T 的规则：

1. 同上，如果 expr 的类型是一个引用，忽略引用部分。
2. 忽略 expr 的引用特性之后，如果 expr 是 const，同样忽略掉，如果是 volatile，也忽略掉。（volatile 对象是特殊的，它们一般只用于实现设备驱动，具体的细节部分，请参考条款 40）

因此：

```
int x = 27;           // 同上
const int cx = x;     // 同上
const int& rx = x;    // 同上

f(x);                 // T和param的类型都是int
f(cx);                // T和param的类型还都是int
f(rx);                // T 和 param 的类型依然都是 int
```

注意，即使 cx 和 rx 提供了 const 值，param 也不是 const。这个很容易理解，param 是一个与 cx 和 rx 完全独立的对象——一个 cx 或 rx 的拷贝，事实上，cx 和 rx 不能修改跟 param 能否修改没有任何关系。这就是为什么 expr 的 const 特定（抑或 volatile 特性）在推导 param 类型时被忽略：expr 不能修改不代表它的拷贝不能被修改。

认识到 const（以及 volatile）只有在传值形参才被忽略，这一点非常重要。就像我们前面提到的，对于那些 references-to-const 或 pointers-to-const 的形参，const 特性在类型推导时被保留，但考虑这种情况，expr 是一个指向 const 对象的 const 指针：

```
template<typename T>
void f(T param);          // param is still passed by value

const char* const ptr =  // ptr is const pointer to const object
    "Fun with pointers";

f(ptr);                   // pass arg of type const char * const
```

此处，星号右边的 const 声明 ptr 为 const：ptr 既不能改变指向其他地址，也不能设置为 null，（星号左边的 const 说明 ptr 指向的字符串是 const，因此不能被修改。）当 ptr 传给 f

时，构成指针的所有比特位拷贝给 param，也就是 ptr 指针本身按值传递。根据传值形参的类型推导规则，ptr 的 const 特性被忽略，param 推导出类型为 const char*，即一个指向 const 字符串的可变指针。ptr 指向内容的 const 特性在类型推导时被保留，但当拷贝至指针 param 时，ptr 本身的 const 特性被忽略。

数组实参

前面的情况基本覆盖了主流的模板类型推导，但是有一个特别需要注意的情况，那就是，数组类型与指针类型是不一样的，即使有时它们看上去是可以内部转换的。造成这种错觉的一个基本情况就是，在很多上下文环境里，一个数组退化成一个指向它第一个元素的指针，这种退化允许下面的这种代码能否通过编译：

```
const char name[] = "J. P. Briggs";    // name's type is const char[13]
const char * ptrToName = name;         // 数组退化成指针
```

此处，const char* 指针 prtToName 以 const char[13]类型的 name 来初始化，这两个类型（const char*和 const char[13]）不是一样的，但因为 array-to-pointer 退化规则，代码可以编译。

但是，如果将一个数组传递给具备 by-value 形参的模板，会怎样呢？

```
template<typename T>
void f(T param); // template with by-value parameter


f(name);          // T 和 param 推导出什么类型？
```

首先，我们注意到，没有一个函数的形参是数组。是的，下面的语法是合法的，

```
void myFunc(int param[]);
```

但是数组声明是当做指针声明来处理的，意味着，myFunc 等同于下面这样的声明：

```
void myFunc(int* param); // same function as above
```

这种数组形参等同于指针形参的特性，是由 C++的基石 C 发展而来，并且它造成一种错觉，似乎数组和指针类型是一样的。

因为数组形参声明被当做指针形参，数组类型按值传递给模板函数，将被推导为指针类型，也就意味着，模板 f 的调用中，它的类型参数 T 被推导为 const char*：

```
f(name); // name 是数组，但 T 被推导为 const char*
```

但是来一个特例。尽管函数不能声明形参为真正的数组，但它们可以声明形参为数组的

引用！所以，如果修改模板 f 的形参为引用，

```
template<typename T>
void f(T& param); // template with by-reference parameter
```

我们再传递一个数组给它，

```
f(name); // pass array to f
```

T 的类型推导出来就是实际的数组类型！这个类型包含数组长度，所以在这个例子中，T 被推导为 const char[13]，而 f 的形参类型为 const char(&)[13]（一个数组引用）。是的，这个语法看上去怪怪的，但是理解了这些却可以升华你的精神。

有趣的是，利用声明数组引用的能力，可以创建出推导数组元素个数的一个模板：

```
// 返回数组的大小作为编译时的常量.
//（数组形参没有名称，因为我们只关心它的元素个数）
template<typename T, std::size_t N>       // 关于constexpr和noexcept
constexpr std::size_t arraySize(T (&)[N]) noexcept // 查看下面的条款
{
    return N;
}
```

正如条款 15 解释的，声明函数为 constexpr，可以使得函数返回值在编译期间可用。这样就可以计算一个花括号初始化的数组大小，并且初始化一个同样元素个数的数组。

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };    // keyVals有7个元素
int mappedVals[arraySize(keyVals)];             // mappedVals也一样有7个元素
```

当然，作为一个 modern C++开发人员，你应该很自然地使用 std::array 来创建数组：

```
std::array<int, arraySize(keyVals)> mappedVals; // mappedVals有7个元素
```

因为 arraySize 被声明为 noexcept，所以有助于编译器生成更好的代码，具体细节，请参考条款 14。

函数实参

在 C++里面，不是仅仅只有数组可以退化为指针，函数类型也可以退化为函数指针，并且，我们讨论的所有关于数组的类型推导规则都适用于函数类型推导和函数指针退化，因此：

```
void someFunc(int, double);   // someFunc is a function;
                              // type is void(int, double)

template<typename T>
void f1(T param);             // in f1, param passed by value
```

```
template<typename T>
void f2(T& param);              // in f2, param passed by ref

f1(someFunc);                   // param deduced as ptr-to-func;
                                // type is void (*)(int, double)
f2(someFunc);                   // param deduced as ref-to-func;
                                // type is void (&)(int, double)
```

这个在实践中很少有不同，但如果你知道了 arrary-to-pointer 的退化，你可能也就知道了 function-to-pointer 的退化。

到此为止，你已经了解到了 auto 相关的模板类型推导规则。我把重要的部分在下面着重列出来。当根据左值来推导通用引用类型时，那种特殊处理令人感觉有些混乱，不过，数组和函数的退化到指针（decay-to-pointer）规则更令人感觉混乱。有时候你只想简单地向编译器大吼一声，“告诉我你推导出了什么类型！”。如果出现这种情况，请转到条款 4，因为它致力于让编译器告诉你结果。

小结

需要铭记的要点
● 在模板的类型推导期间，带有引用的实参被当做无引用处理，即引用特性被忽略。 ● 当推导通用引用形参的类型时，左值实参会特殊处理。 ● 当推导传值(by-value)形参的类型时，const 和/或 volatile 实参被当做非 const 和非 volatile 处理。 ● 在模板的类型推导期间，数组或函数是实参退化为指针，除非是传给引用。

条款 2：理解 auto 类型推导

如果你已经读过了条款 1 的模板类型推导，那么已经知道了 auto 类型推导的大部分情况，因为除了一种例外情况，auto 类型推导就是模板类型推导。但什么才是那种例外呢？模板类型推导涉及到模板、函数和形参，但 auto 不用处理这些。

这是正确的，但没有关系，在模板和 auto 类型推导之间有个直接映射关系，它们之间有一个转换规则。

在条款 1，模板类型推导是用下面的模板函数来解释的

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

这个通用调用是：

```
f(expr); // call f with some expression
```

在调用 f 的地方，编译器使用 expr 来推导 T 和 ParamType 的类型。当一个变量用 auto 来声明时，auto 扮演的是 T 在模板里的角色，而变量的类型就是 ParamType 角色。用代码演示比描述更简单，所以，考虑下面的例子：

```
auto x = 27;
```

此处，x 类型仅仅简单指定为 auto，另一方面，下面这个声明，

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

类型指定为 const auto，而再下面，

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

类型指定为 const auto&。在上面例子中，为了推导 x、cx 和 rx 的类型，编译器处理起来就好像是每个声明对应有一个模板，同时用关联的那个初始化表达式来调用模板：

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

func_for_x(27);                     // 推导x类型的概念上调用
                                    // param的推导类型就是x的类型

template<typename T>                // 推导cx类型的概念上模板
void func_for_cx(const T param);

func_for_cx(x);                     // 推导cx类型的概念上调用
                                    // param的推导类型就是cx的类型
```

```
template<typename T>                    // 推导rx类型的概念上模板
void func_for_rx(const T& param);

func_for_rx(x);                         // 推导rx类型的概念上调用
                                        // param 的推导类型就是 rx 的类型
```

就像我说过的，除了一种例外情况（我们后面讨论），auto 类型推导就跟模板类型推导一样。

基于函数模板中 ParamType 的特性和 param 的类型指示符，条款 1 将模板类型推导分为三种情形。在使用 auto 的声明中，类型指示符替代了 ParamType，所以这儿也有三种情形：

- 情形 1：类型指示符是一个指针或引用，但不是一个通用引用。
- 情形 2：类型指示符是一个通用引用。
- 情形 3：类型指示符既不是指针也不是引用。

我们已经看到了情形 1 和 3 的例子：

```
auto x = 27;          // 情形 3 (x is neither ptr nor reference)
const auto cx = x;    // 情形 3 (cx isn't either)
const auto& rx = x;   // 情形 1 (rx is a non-universal ref.)
```

情形 2 就像你预想的那样：

```
auto&& uref1 = x;     // x is int and lvalue,
                      // so uref1's type is int&
auto&& uref2 = cx;    // cx is const int and lvalue,
                      // so uref2's type is const int&
auto&& uref3 = 27;    // 27 is int and rvalue,
                      // so uref3's type is int&&
```

条款 1 提到了，非引用类型指定的数组和函数会退化为指针，这个同样适用于 auto 类型推导：

```
const char name[] =       // name's type is const char[13]
    "R. N. Briggs";
auto arr1 = name;         // arr1's type is const char*
auto& arr2 = name;        // arr2's type is const char (&)[13]
void someFunc(int, double);   // someFunc is a function;
                              // type is void(int, double)
auto func1 = someFunc;        // func1's type is
                              // void (*)(int, double)
auto& func2 = someFunc;       // func2's type is
                              // void (&)(int, double)
```

就像你看到的，auto 类型推导跟模板类型推导工作类似，它们就像一块硬币的两面。

它们只有一种情况是不同的。我们首先从声明一个初始值为 27 的 int 开始讲解，C++98 给你提供两种语义选择：

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

C++11, 通过标准支持的统一初始化方式（即花括号），添加以下形式：

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

以上四种语法，仅得到一种结果：一个初始化为 27 的 int。

但正如条款 5 解释的，用 auto 代替固定类型来声明变量是具有很多优势的，所以，用 auto 替换上面变量声明中的 int，直观的替换代码如下：

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

这些声明都可以编译通过，但他们的含义与替换之前已经有所不同，前两个语句的确是声明了一个值为 27 的 int，不过，后两个语句声明的是包含单个元素（值为 27）的 std::initializer_list<int>！

```
auto x1 = 27;          // type is int, value is 27
auto x2(27);           // 同上
auto x3 = { 27 };      // type is std::initializer_list<int>,
                       // value is { 27 }
auto x4{ 27 };         // 同上
```

这是因为 auto 有个特殊的类型推导规则。当用花括号来初始化一个 auto 声明的变量时，推导类型是 std::initializer_list。如果这样的类型不能被推导（比如，因为花括号初始化列表中的值包含不同的类型），代码编译出错：

```
auto x5 = { 1, 2, 3.0 }; // error! can't deduce T for
                         // std::initializer_list<T>
```

就像评论中指出的，类型推导在这种情形下是失败的，但重要的是要认识到，此处产生了两种类型推导，一种是根据 auto 使用：x5 的类型被推导，因为 x5 用花括号初始化，所以 x5 必须被推导为 std::initializer_list，但 std::initializer_list 是一个模板，实例化需要类型 T，也就是说 T 必须被推导出来，这时就产生第二种类型推导：模板类型推导，在这个例子中，推导是失败的，因为花括号初始化的值包含多个类型。

对花括号初始化的处理方式是 auto 和模板类型推导的唯一区别。当用花括号初始化

auto 声明的变量时，推导出的类型是一个实例化的 std::initializer_list，但如果向类似的模板传递一个同样的初始化，类型推导失败，代码编译出错：

```
auto x = { 11, 23, 9 };  // x's type is
                         // std::initializer_list<int>
template<typename T>     // template with parameter
void f(T param);         // declaration equivalent to
                         // x's declaration
f({ 11, 23, 9 });        // error! can't deduce type for T
```

不过，如果你指定模板的 parma 是一个 std::initializer_list<T>，模板类型推导将推导出 T 的类型：

```
template<typename T>
void f(std::initializer_list<T> initList);
f({ 11, 23, 9 });    // T deduced as int, and initList's
                     // type is std::initializer_list<int>
```

所以，auto 和模板类型推导唯一的不同之处就是，auto 假定花括号初始化代表一个 std::initializer_list，但模板类型推导不是。

你可能想知道，为什么 auto 类型推导有个花括号初始化的特殊规则，而模板类型推导缺没有。我也想知道这点，哎，但我还没能找到一个令人信服的解释。但规则就是规则，这也意味着你必须记住，当你用 auto 声明变量并用花括号初始化时，推导出来的类型永远是 std::initializer_list。特别重要的是，如果你认可统一初始化的哲学——使用花括号初始化是理所当然的，你的潜意识里必须接受这一点。在 C++11 编程中，一个经典错误就是，突然声明了一个 std::initializer_list 变量，而你想要的却是其他类型。这个缺陷也导致一些开发人员只有在必须的时候才用花括号初始化。（条款 7 将讨论何时是必须的）

对于 C++11，这儿已经讲解完毕，但对于 C++14，还有后续内容。C++14 允许 auto 用来推导函数返回类型（参考条款 3），而且 C++14 的 lambdas 也可能用 auto 声明形参。不过，这些 auto 使用情况应用的是模板类型推导，不是 auto 类型推导，所以，使用 auto 返回类型的函数如果返回一个花括号初始化的话，编译会出错：

```
auto createInitList()
{
    return { 1, 2, 3 };  // error: can't deduce type
}                        // for { 1, 2, 3 }
```

同样的，在 C++14 的 lambda 形参使用 auto，编译也报错：

```
std::vector<int> v;
…
```

```
auto resetV =
    [&v](const auto& newValue) { v = newValue; }; // C++14
...
resetV({ 1, 2, 3 }); // error! can't deduce type
                     // for { 1, 2, 3 }
```

需要铭记的要点
● auto 类型推导通常是与模板类型推导一样的，但 auto 类型推导假定花括号初始化表示一个 std::initializer_list，而模板类型推导不是。 ● auto 用于函数返回类型或 lambda 形参时，应用的是模板类型推导，而不是 auto 类型推导。

条款 3：理解 decltype

decltype 是一个奇特的发明，给出一个变量名或表达式，decltype 告诉你变量名或表达式的类型。关键是，它告诉你的也正是你所期望的。然后有时候，它提供的返回类型也会令你抓狂，并且要翻阅一些参考文献或者在线 Q&A 网站来理解它。

我们首先从典型的情形开始讲解——它不会令人感到惊讶。与模板和 auto 的类型推导行为相比（条款 1 和 2），decltype 实际上就是复述一遍变量名或表达式的实际类型。

```
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
};                          // decltype(Point::y) 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&
```

看到了吗？没有任何惊奇的地方。

在 C++11 中，也许 decltype 基本用法就是声明那些返回类型依赖形参类型的函数模板。例如，假设我们想写一个函数，这个函数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。

对类型 T 的容器对象操作[]返回一个 T&，比如，std::deque 就是这样，而 std::vector 也基本是这样，除了 std::vector<bool>，操作[]不是返回 bool&，而是返回一个新的对象，具体原因以及遇到这种情况该如何处理，请参考条款 6，这儿重点要表达的是，容器的[]运算符返回类型依赖于容器。

delcltype 很容易实现这样的情形。下面是我们想要实现的模板的第一个版本，使用 decltype 来计算返回类型，模板还需要一些优化，但我们后续再说：

```
template<typename Container, typename Index>  // 正确但需要优化
auto authAndAccess(Container& c, Index i)
    -> decltype(c[i])
{
    authenticateUser();
    return c[i];
}
```

函数名前面的 auto 跟类型推导没有任何关系，而是表明使用了 C++11 的尾随返回类型语法，即函数的返回类型在形参之后声明（在“->”之后）。尾随返回类型的好处就是函数的形参可以用来指定返回类型，例如，在 authAndAccess 函数中，我们用 c 和 i 来指定返回类型，如果我们还是用传统方式在函数名之前声明返回类型的话，c 和 i 是不可用的，因为它们还没声明。

使用这个声明，无论传入的容器[]运算符返回类型是什么，authAndAccess 都是返回我们想要的类型。

C++11 允许单语句的 lambda 返回类型被推导，而 C++14 扩展到允许所有的 lambda，包括那些多语句的，以及所有的函数。这意味着，在 authAndAccess 例子中，我们可以省去尾随返回类型，只留下开头的 auto，使用那样的声明格式，auto 就意味着要类型推导，准确的说，是意味着编译器将根据函数的实现来推导返回类型：

```
template<typename Container, typename Index>  // C++14版本
auto authAndAccess(Container& c, Index i)
{
    authenticateUser();
    return c[i];              // return type deduced from c[i]
}
```

根据条款 2，函数的 auto 返回类型指定，编译器将采用模板类型推导，在这个例子中，有点问题，正如我们前面讨论的，大多数 T 类型容器的[]运算符返回 T&，但条款 1 提到，模板类型推导时，引用特性是被忽略的，思考对于下面这段代码意味着什么：

```
std::deque<int> d;
…
authAndAccess(d, 5) = 10; // 返回d[5],然后给它赋值为10，这个编译不通过！
```

此处，d[5]返回一个 int&，但 auto 返回类型推导会去掉引用，因此返回类型是 int，那么函数返回值就是 int，是个右值，而上面的代码是尝试给右值 int 赋值 10，这在 C++中是不

允许的，所以代码无法编译通过。

要让 authAndAccess 像我们期望的那样，我们需要应用 decltype 类型推导到返回类型，即指定 authAndAccess 返回表达式 c[i]返回类型。C++拥护者预测到，在一些类型被推测的情形下，需要用到 decltype 类型推导规则，所以在 C++14 中使用 decltype(auto)指示符来实现。这些乍一看是矛盾的（decltype 和 auto 一起？），实际上却是一个完美实现：auto 表明类型要被推导，而 decltype 表明推导时用 decltype 规则。因此，我们可以这样来写 authAndAccess：

```
template<typename Container, typename Index>  // C++14可以正常工作
decltype(auto)                                // 但还需要改进
authAndAccess(Container& c, Index i)
{
    authenticateUser();
    return c[i];
}
```

现在 authAndAccess 返回的就是 c[i]返回类型，特别的是，通常情况下，c[i]返回一个 T&，authAndAccess 也返回一个 T&，特殊情况下，c[i]返回一个对象，authAndAccess 也将返回一个对象。

decltype(auto)不仅仅用于函数返回类型，也可以很方便的声明变量，将 decltype 类型推导规则应用到初始化表达式：

```
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，你还困惑于两件事情，一是我提到的 authAndAccess 改进还没有描述，让我们现在处理。

再次看下 C++14 版本的 authAndAccess 声明：

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

容器传入的是 lvalue-reference-to-non-const，因为返回容器内一个元素的引用，可以允许用户修改容器内容，但这也意味着不能给这个函数传递右值，右值不能绑定到左值引用（除非是 lvalue-references-to-const，这儿不做展开）

不可否认的是，给 authAndAccess 传递右值容器是一个边界案例，一个右值容器，即一

个临时对象，在调用 authAndAccess 的语句结束后即销毁，也就意味着那个容器中一个元素引用（也就是 authAndAccess 即将返回的）在语句结束后即悬挂（dangle）（相当于野指针——译者注），但给 authAndAccess 传递一个临时对象仍然是有用的，用户可能只是简单的拷贝临时容器的一个对象，例如：

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

要支持这种应用，我们就要修订一下 authAndAccess 的声明，可以同时接受左值和右值，重载可以满足要求（一个声明左值引用形参，另一个声明右值应用形参），但那样的话，我们需要维护两个函数，避免这个的方式就是让 authAndAccess 接受能绑定左值或右值的形参，而条款 24 解释了，通用引用就能做到，因此，authAndAccess 可以声明成这样：

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container&& c, // c现在是一个通用引用
                             Index i);
```

在这个模板中，我们不知道容器的类型，同样，我们也不关心它用到的索引对象类型。对未知类型对象传值，通常会带来一些风险，比如，因为不必要的拷贝影响性能，对象切片的行为问题（参考条款 41），以及同事们的嘲笑，但是依据 STL 的索引值（即，std::stirng、std::vector、std::deque 的[]运算符中），这些索引类型是合理的，所以我们仍然坚持用传值方式。

不过，依据条款 25 的建议，我们需要改进一下模板的实现，对通用引用使用 std::forward：

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

这个可以实现任何我们想要的情况，但它需要支持 C++14 的编译器，如果没有的话，你需要实现 C++11 版本的模板，除了返回类型需要自己指定以外，其他部分与 C++14 版本相同：

```
template<typename Container, typename Index> // C++11最终版本
auto
authAndAccess(Container&& c, Index i)
```

```
    -> decltype(std::forward<Container>(c)[i])
{
    authenticateUser();
    return std::forward<Container>(c)[i];
}
```

你还惦记的另一个事情就是我在本条款开头备注的，decltype 差不多总是能生成你期望的类型，很少会令你感到奇怪，说实话，你不太会碰到这些特殊情况，除非你是一个重型库（heavy-duty library）的实现人员。

要完全搞懂 decltype 的行为，你需要熟悉一些特殊情形，它们大部分都太难懂了，本书也无法保证讨论全面，也只能看看 decltype 的本质以及它的应用。

对变量名使用 decltype 得到的是那个变量名声明的类型，变量名属于左值表达式，但不会影响 decltype 的行为，然后，对于比变量名更复杂的左值表达式，decltype 得到的类型总是一个左值引用，也就是说，对于一个非变量名的类型 T 左值表达式，decltype 报告的类型是 T&。这很少会产生影响，因为大多数左值表达式内在包含左值引用限定符，例如，返回左值的函数，总是返回左值引用。

不过，有个影响需要关注。如：

```
int x = 0;
```

x 是一个变量的名称，所有 decltype(x)是 int，但是用括号包装一下 x——“(x)”——生成一个比名称复杂的表达式，作为名称，x 是一个左值，而 C++定义表达式(x)也是左值，所以，decltype((x))得到 int&。名称外面加括号将改变 decltype 的结果！

在 C++11 中，这个只是有一点奇怪，但结合 C++14 的 decltype(auto)，这个意味着，你在返回语句上的一个小小改变，将影响推导出来的函数返回类型：

```
decltype(auto) f1()
{
    int x = 0;
    …
    return x; // decltype(x)是int, 所以f1返回int
}

decltype(auto) f2()
{
    int x = 0;
    …
    return (x); // decltype((x))是int&, 所以f2返回int&
}
```

注意，f2 不仅仅是返回类型于 f1 不同，而且还是返回一个临时变量的引用！这样的代码将产生无法预知的结果——肯定不是你想要的结果。

最主要的经验教训就是使用 decltype(auto)时要特别小心。用来推导类型的表达式里面，一个看上去微不足道的细节，都可能影响 decltype(auto)的结果。为了确保推导出你预期的类型，请使用条款 4 描述的技术。

但是，不要因小失大，decltype（包括单独使用以及和 auto 结合使用）的确偶尔会产生令人惊讶的类型推导结果，但毕竟不是普遍情况，一般来说，decltype 都是得到你所预期的类型，尤其是应用到变量名时，因为那种情况下，decltype 的确仅仅报告变量名的声明类型。

需要铭记的要点
● decltype 推导一个变量或表达式类型时，不会有任何更改。（相对 auto 或模板，不会有忽略引用或 const 之类的——译者注） ● 对于非变量名的类型 T 表达式，decltype 总是报告为类型 T& ● C++14 支持 decltype(auto)，与 auto 类似，根据初始化来推导类型，但使用的是 decltype 规则。（即第一条说的，不会有改变，原封不动——译者注）

条款 4：了解如何查看推导出的类型

选择何种类型推导结果查看工具，取决于你想在软件开发过程的哪个阶段查看，我们要探讨三种阶段：编写代码、编译和运行。

IDE 编辑器（编写代码阶段——译者注）

IDE 的代码编辑器一般会在你将鼠标移上去时，显示程序实体（如变量、形参、函数等）的类型，例如，给出以下代码

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

IDE 编辑器可能会显示出 x 的推导类型为 int，y 的类型是 const int*。

要实现这一点，你的代码必须处于一个或多或少可编译的状态，因为只有在 IDE 内部运行的 C++编译器（或者至少是前台运行的）可以提供此类信息，如果编译器不能有效分析你的代码，然后执行类型推导，就不能显示出它推导出什么类型。

对于简单类型，如 int，IDE 一般很容易显示，不过，当包含了复杂类型时，IDE 显示出来的类型可能不是特别地有用了。

编译器诊断（编译阶段——译者注）

要让编译器显示出它推导出的类型，一个有效的方法就是，错误的使用那种类型，从而导致编译报错，错误报告里面会很清楚的显示出那个出错类型。

例如，假设我们想看到上个例子中的 x 和 y 类型，我们首先声明一个未定义的类模板，如下：

```
template<typename T> // declaration only for TD;
class TD;            // TD == "Type Displayer"
```

如果尝试去实例化这个模板，将会产生一个错误信息，因为没有模板定义可以实例化。要看到 x 和 y 的类型，只要试着去用它们的类型来实例化 TD 即可：

```
TD<decltype(x)> xType; // elicit errors containing
TD<decltype(y)> yType; // x's and y's types
```

我使用的是变量名格式 variableNameType，因为这样产生的错误信息有助于帮助我找到

我想要的信息（decltype 有两种格式，变量名和表达式，用变量名话，错误信息里面的类型就是实际要知道的类型，而表达式得到的类型时引用，所以用变量名更直观。——译者注）。对于上面的代码，一种编译器的诊断结果部分如下（类型已经高亮）：

```
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
```

不同的编译器提供的信息一样，但格式不同：

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

忽略格式区别，使用这种技术手段，我测试的所有编译器都能提供有用的错误信息。

运行时输出（运行阶段——译者注）

printf 到运行的时候可以用来显示类型信息（我并不是推荐你使用 printf），但是它提供了对输出格式的完全掌控。挑战就在于创造一个适合显示的文本表示法。“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，你可能指出，我们可以这样写：

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

这个方法基于这样的原理：在类似 x 或 y 这样的一个对象上，调用 typeid 将生成一个 std::type_info 对象，并且 std::type_info 包含一个 name 成员函数，返回类型名称的一个 C 分隔字符串（即 const char*）。

调用 std::type_info::name 不能保证返回的是一目了然的信息，但实现上是能起到帮助作用的，不同编译器的帮助水平是不一样的，例如，GNU 和 Clang 编译器报告 x 类型是“i”，y 类型是“PKi”，如果你知道 i 代表 int，PK 代表“pointer to const”，刚才的结果就皆可以理解了。（两个编译器都是一个工具 c++filt，可以解码这些“乱七八糟”的类型）Microsoft 的编译器提供更加直白的输出：x 是“ int ” x ，y 是“ int const* ”。

因为这些结果对 x 和 y 而言都是正确的，你可能认为类型报告的问题就此解决了，但不能掉以轻心，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();  // factory function

const auto vw = createVec();      // init vw w/factory return
if (!vw.empty()) {
    f(&vw[0]);                    // call f
    ...
}
```

这段代码涉及到一个用户自定义类型（Widget）、一个 STL 容器（std::vector）、一个 auto 变量（vw），对于你想要了解的编译器推导类型可视化，是一个更具代表性的场景，例如，可以知道模板类型参数 T 和函数 f 的形参 param 推导出了什么类型。

直接使用 typeid，在 f 函数添加一些代码：

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

用 GNU 和 Clang 编译器执行，产生以下输出：

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

我们已经知道，PK 代表“pointer to const”，只有数字 6 是神秘的，简单来说，这是类（Widget）名称的字符个数，所以，这两个编译器告诉我们，T 和 param 的类型都是 const Widget*。

Microsoft 的编译器得到：

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

三个不同的编译器输出的是同样的信息，似乎意味着信息是准确的，但再仔细看看，在模板 f 中，param 的声明类型是 const T&，问题就在这里，T 和 param 是同样的类型，不感到奇怪吗？例如，如果 T 是 int，那么 param 的类型应该是 const int&——根本不会是一样的类型。

遗憾的是，std::type_info::name 的结果并不可靠，在这个例子中，三个编译器报告的 param 类型都是不正确的，而且，它们本质上就是不正确的，因为 std::type_info::name 的使

用说明指出，类型处理方式，好像是被当做一个传值参数传递给模板函数，正如条款 1 解释的，那样就意味着，如果类型是一个引用，它的引用特性会被忽略，并且引用移除后，类型是 const（或 volatile），const（或 volatile）特性也被忽略，这就是为什么 parma 的类型——实际是 const Widget * const &——被报告为 const Widget *，首先类型的引用特性被移除，然后 const 特性也被消除掉。

同样遗憾的是，IDE 编辑器显示的信息也不可靠，或者至少不是有用的。还是同样的例子，我知道一个 IDE 编辑器给出 T 的类型是：

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

给出 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”。还好，开发环境能对这样的代码运行比较好。

如果你更加倾向于依赖库而不是运气，你就应该知道，std::type_info::name 和 IDE 可能失败的地方，Boost TypeIndex 库（经常写做 Boost.TypeIndex）却可以成功显示。这个库并不是 C++标准的一部分，也不是 IDE 或类似于 TD 的模板。而且，事实上，Boost 库（在 boost.com）是一个跨平台的、开源的，并且在 license 下可用，这就意味着，使用 Boost 库基本和 STL 一样方便。

下面使用 Boost.TypeIndex，让函数 f 输出精准类型信息：

```
#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';
```

```
    ...
}
```

它的工作原理是，函数模板 boost::typeindex::type_id_with_cvr 接受一个类型实参（也就是我们需要知道信息的类型），并且不会去掉 const、volatile 或引用标识（因为模板名称有“with_cvr”），返回结果是 boost::typeindex::type_index 对象，该对象的 pretty_name 成员函数生成一个类型的 std::string，采用的是一个对人友好的（human-friendly）表示法。

利用这个 f 的实现，再考虑一下刚才使用 typeid 产生不正确信息的调用：

```
std::vector<Widget> createVec();  // factory function

const auto vw = createVec();      // init vw w/factory return
if (!vw.empty()) {
    f(&vw[0]);                    // call f
    ...
}
```

在 GNU 和 Clang 编译器下，Boost.TypeIndex 输出下面的精准信息：

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

在 Microsoft 编译下也是一样：

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

这种接近相同的结果非常漂亮，但是需要注意 IDE 编辑器，编译器错误信息以及类似于 Boost.TypeIndex 的库仅仅帮助你得到编译器推导出什么类型的一种工具而已，所有的都是有帮助意义的，但是到目前为止，理解条款 1-3 的类型推导信息，没有捷径可走。

小结

需要铭记的要点
● 通过 IDE 编辑器、编译器错误信息以及 Boost TypeIndex 库，推导的类型一般都能看出来。 ● 一些工具的结果可能既没有帮助也不准确，所以理解 C++的类型推导规则才是王道。

第二章auto

看上去， auto 要多简单有多简单，但是它可比看上去要微妙的多。用它来减少码字，当然没问题，但它更是防止了困扰手动类型声明的正确性和性能问题，而且，一些 auto 类型，严格地遵守规定的算法规则，推导出来的结果，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 5：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，但是这依赖于上下文语境，哎。

不要介意，我们来看看另一个简单乐趣，声明一个局部变量，通过解引用一个迭代器来初始化：

```
template<typename It> // algorithm to dwim ("do what I mean")
void dwim(It b, It e) // for all elements in range from
{                     // b to e
    while (b != e) {
        typename std::iterator_traits<It>::value_type currValue = *b;
        ...
    }
}
```

呃，“typename std::iterator_traits<It>::value_type”表示的是一个迭代器指向的值的类型吗？真的吗？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，简单乐趣是：声明一个闭包类型的局部变量。哦，对了，一个闭包的类型只有编译器知道，因此不能被显式的写出来，哎，见鬼。

见鬼，见鬼，见鬼！使用 C++ 编程得不到它应有的快乐！

是的，过去的确不是。但是有了 C++11 ，得益于 auto ，这些烦恼都没了。auto 变量从

它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，在 Modern C++时代，你可以跟未初始化变量的一箩筐问题说拜拜了。

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

声明一个局部变量，并用解引用迭代器初始化，不再是问题：

```
template<typename It> // as before
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 derefUPLess =                          // 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
```

酷毙了！C++14 更爽，因为 lambda 表达式可以用 auto：

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

尽管非常酷，也许你在想，我们不需要使用 auto 去声明一个闭包类型的变量，因为我们可以使用一个 std::function 对象。对头，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，它推广了函数指针概念。函数指针只能指向函数，然而 std::function 对象可以应用任何可以被调用的对象，即任何可以像函数那样调用的东东。就像创建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，创建一个 std::function 对象时，也必须指明它所涉及的函数类型。你可以通过 std::function 的模板参数来实现。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 function
```

你可以这样写：

```
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，闭包可以存储在 std::function 对象里面。这意味着，我们可以声明 C++11 版本不使用 auto 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; };
```

重要的是，要认识到，抛开冗长的语义和需要重复形参类型这两点，使用 std::function 和使用 auto 并不一样。一个 auto-declared 持有一个闭包的变量和闭包具有同样的类型，这样的话，占用的内存也仅仅是闭包大小。用 std::function 声明持有一个闭包的变量，实际上是 std::function 模板的一个实例，而这种实例对任何类型都有一个固定大小，这个内存大小可能不能满足闭包的存储需求，出现这种情况时，std::function 将会分配堆空间来存储这个闭包，结果就是 std::function 对象一般会比 auto 声明的对象占用更多内存，并且，由于实现细节限制了内联（restrict inlining）并产生了间接函数调用（indirect function calls），通过 std::function 对象来调用一个闭包必然会比通过 auto-declared 对象调用要慢。换言之，与 auto 方法相比，std::function 方法通常占用内存更大，运行更慢，而且还可能导致内存不足的异常。还有，就像你在上面一个例子中看到的，编写 auto 的工作量明显小于 std::function 实例。对于闭包这种情况，auto 明显比 std::function 更合适。（在条款 34 中，对于持有 std::bind 调用的返回值，auto 同样比 std::function 更合适，当然，建议还是尽可能使用 lambda 表达式，而不是 std::blind）。

除了避免未初始化变量、冗长的变量声明以及直接持有闭包的能力，auto 还有更多优势。一是避免“类型截断（type shortcuts）”打来的问题。下来是你可能见过甚至写过：

```
std::vector<int> v;
…

unsigned sz = v.size();
```

v.size() 的官方返回类型是 std::vector<int>::size_type，但是很少有开发者意识到这点。std::vector<int>::size_type 被指定为一个无符号整数类型，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结果。例如，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 的高明之处吗？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？

要是你记得 std::unorder_map 的 key 部分是 const 类型的，你就可以看出问题在哪里，哈希表（std::unorder_map 就是哈希表）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），这样就需要先复制 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，在每个循环结束的时候销毁临时对象。如果你写了这个循环，你将会感觉代码的行为令人吃惊，因为你本来只是想简单地绑定引用 p 到 m 的每个元素上。

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，也更容易敲代码。而且，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，而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。但是牢记一点，C++并没有在这个方面有什么大的突破，这种技术在编程语言中被熟知，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）也有或多或少相同的特点，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 会失去看一眼源代码就能确定对象类型的能力。然而，IDE 提示对象类型的功能一般能缓解这个问题（甚至考虑到在条款 4 中提到的 IDE 的类型显示问题），并且，很多情况下，一个对象类型的摘要视图和准确的类型一样有用。比如，只要知道这个对象是容器还是计数器或者智能指针，而不需要知道这个容器、计数器或者智能指针的准确类型。假如变量名称选择恰当，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 类型会随着初始化它的表达式自动改变，这意味着使用 auto，代码重构可以变得更简单。例如，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值存储在一个 auto 变量中，在下次编译的时候，调用代码将会自动的更新。如果返回值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。

需要铭记的要点
● auto 变量必须初始化，可以避免因为类型不匹配而导致可移植性或效率问题，能否简化代码重构过程，特别是比显式指定类型少敲代码。 ● auto-typed 变量受制于条款 2 和 6 描述的陷阱。

条款 6：当 auto 推导出非预期类型时，使用显式类型初始化

条款 5 解释了，使用 auto 声明变量，比直接显式声明类型具备一系列的技术优势，但有时候 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 并返回一个 std::vector<bool>，其中每个 bool 表征 Widget 是否提供一个特定的特性：

```
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];   // is w high priority?
…
processWidget(w, highPriority);       // process w in accord
                                      // with its priority
```

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]; // is w high priority?
```

情况变了。所有的代码还是可以编译，但是它的行为变得不可预测：

```
processWidget(w, highPriority); // undefined behavior!
```

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？答案似乎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[]并不是返回容器中元素的引用（除了 bool，std::vector::operator[]对所有的类型都返回引用），实际上，它返回的是一个 std::vector<bool>::reference 对象（是一个在 std::vector<bool> 中内嵌的类）。

std::vector<bool>::reference 的存在是因为 std::vector<bool>做了特化，bool 值做了封包处理，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++禁止对 bits 的引用。无法返回一个 bool&，std::vector<T>的 operator[]于是就返回了一个行为上和 bool& 相似的对象。要达到这种效果，std::vector<bool>::reference 对象必须能在 bool& 可用的所有语境下使用。 所以 std::vector<bool>::reference 对象具备可以隐式转换成 bool 的特性。（不是转换成 bool&，而是 bool。要解释 std::vector<bool>::reference 对象如何模拟一个 bool&行为，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分。）

带着以上信息，再看看原先的代码：

```
bool highPriority = features(w)[5];    // declare highPriority's
                                       // type explicitly
```

此处，features 返回了一个 std::vector<bool>对象，并调用 operator[]，operator[]返回一个 std::vector<bool>::reference 对象，然后隐式转换成用来初始化 highPriority 的 bool 类型，因此，就用 features 返回的 std::vector<bool>的第 5 个 bit 的值来赋值 highPriority，就像我们所预期的那样。

再看使用 auto 的 highPriority 声明，并与之对比：

```
auto highPriority = features(w)[5]; // deduce highPriority's type
```

同样的，features 返回一个 std::vector<bool>对象，并同样的调用 operator[]。operator[]还是返回一个 std::vector<bool>::reference 对象，但是现在有一处不同，因为 auto 推导 highPriority 的类型，highPriority 根本没有赋值 features 返回的 std::vector<bool> 的第 5 个 bit 的值。

它到底是什么值，依赖于 std::vector<bool>::reference 是如何实现的。一种实现是，这样的对象包含一个指向包含 bit 引用的机器字的指针，加上那个 bit 位的偏移量。假设 std::vector<bool>::reference 的实现是这种方式，考虑这个对 highPriority 的初始化意味着什么。

调用 features 会返回一个 std::vector<bool>临时对象。这个对象是没有名字的，但为了方便讨论，我暂且叫它 temp，operator[]是在 temp 上调用的，它返回的 std::vector<bool>::reference 包含一个指向数据结构(此数据结构持有 temp 管理的所有 bits)中一个 word 的指针，加上关联第 5 比特的偏移量，highPriority 是这个 std::vector<bool>::reference 对象的拷贝，所以 highPriority 也包含一个指向 temp 中的一个 word，加上关联第 5 比特的偏移量。在语句结束后，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 会造成未定义行为的原因：

```
processWidget(w, highPriority);    // undefined behavior!
                                   // highPriority contains
                                   // dangling pointer!
```

std::vector<bool>::reference 是代理类的一个例子，代理类是指一个类的存在是为了模拟另外一个类从而保持一致的行为。代理类应用于各种场景。 std::vector<bool>::reference 的存在是为了提供一个对 std::vector<bool> 调用 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（参考第四章）也是托管原始指针资源的代理类。代理类的功能是良好确定的（well-established）。实际上，“代理”模式是软件设计模式中的最坚挺的成员之一。

一些代理类的设计是对用户公开的，例如，std::shared_ptr 和 std::unique_ptr。另外一些代理类的设计或多或少有一些不可见的操作。std::vector<bool>::reference 就是这样一个“不可见”的代理，和它类似的是 std::bitset，对应的是 std::bitset::reference。

同时在 C++库里面，有些类用到了一种称作表达式模板（expression template）的技术。这些库最初是为了提高数值运算的效率。例如，给出一个 Matrix 类和 Matrix 对象 m1，m2，m3 和 m4，下面的表达式：

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

如果 Matrix 的 operator+返回一个结果的代理而不是结果本身，可以计算的更快。也就是说，对于两个 Matrix，operator+可能返回一个类似于 Sum<Matrix, Matrix>的代理类对象而不是一个 Matrix 对象。和 std::vector<bool>::reference 与 bool 的情况一样，这里会有一个从代理类到 Matrix 的隐式转换，这个可以允许 sum 由”=”右边的表达式产生的代理对象进行初始化。（那个对象的类型会编码整个初始化表达式，即，类似于 Sum<Sum<Sum<Matrix,Matrix>, Matrix>, Matrix> 这样的类型，这显然是一个需要对客户屏蔽的类型。）

作为一个通用规则，“不可见”的代理类不能和 auto 一起使用。这种类对象的生命周期一般设计成单个语句，所以创造这种类型的变量会违反基础库的设计初衷。这就是 std::vector<bool>::reference 的情况，而且我们可以看到这种违反设计的做法会导致未定义的行为。

因此，你要避免如下的代码形式：

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

但是你怎么知道什么时候用的是代理类呢？软件使用它们的时候并不可能会告知它们的存在。它们至少在概念上是不可见的！一旦你发现了，难道你就必须放弃条款 5 所提到的各种 auto 的好处吗？

我们先看看怎么解决如何发现它们的问题。尽管“不可见”代理类的设计是为了让程序员每天用的时候，不用关注它是啥东东，用到它们的库一般会撰写相关的文档。你对库的基础设计理念越熟悉，你就越不可能被库里面那些代理使用搞得狼狈不堪。

如果文档内容不多，最好是看看头文件，很少有源码会把代理类完全隐藏起来，它们都是从用户调用的函数返回，所以从函数签名就可以看出来。下面是

std::vector<bool>::operator[]的说明（spec）：

```
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 std::vector<T>的 operator[]一般是返回 T&，特殊情况是返回代理类。用的时候，多看看接口的内容，一般都可以发现是不是用了代理类。

实践中，很多开发者只有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时，才发现代理类的使用。无论是怎么发现的，一旦 auto 推导出的类型是代理类而不是被代理的类型，你的解决方案不应该是放弃使用 auto，auto 本身是没有问题的，问题在于 auto 推导的类型不是想要的类型，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类型初始化原则。

显式类型初始化原则就是，使 auto 声明一个变量，但是将初始化表达式强转成想要的类型。例如，下面将 highPriority 类型强行设定为 bool：

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

这里，features(w)[5] 还是跟平常一样，返回一个 std::vector<bool>::reference 对象，但是强制类型转换将表达式的类型强行设为 bool，然后 auto 正好推导成 highPriority 的类型。在运行的时候，从 std::vector<bool>::operator[]返回的 std::vector<bool>::reference 对象转换成 bool，作为转换的一部分，从 features 返回的指向 std::vector<bool>的有效（still-valid）指针被解引用。这样就避免了之前的那种未定义行为。索引 5 对应的 bool 值用来初始化 highPriority。

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);
```

这个原则不仅仅用于那种生成代理类的初始化情况，在你有意要创建一个类型与初始化表达式生成的类型不一样的变量时，这个原则同样很有用。例如，假设有一个计算公差（tolerance）的函数：

```
double calcEpsilon(); // return tolerance value
```

calcEpsilon 明确的返回一个 double，但是假设你知道，对于你的程序，float 的精度就够用了，而且你很介意 double 和 float 的长度区别。你可以声明一个 float 变量去存储 calcEpsilon 的返回值：

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

但这个很难看出来“我是有意减小函数返回值的精度”，不过，使用显式类型初始化原则，就可以看出来：

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

同样的，如果你想有意的把一个 floating-point 表达式存储成整数值，也可以用这个原则。假设在一个支持随机访问的容器中（如，std::vector,std::deque，或 std::array），你要计算一个元素的下标，并且给定一个 0.0 至 1.0 之间的 double 值，表示元素离容器开始位置的距离（0.5 表示在容器中间），进一步假设，你确定结果是与 int 匹配的，假如容器是 c 和 double 值是 d，那么计算下标的方式如下：

```
int index = d * c.size();
```

但这个看不出你是有意要将 double 转成 int 的，用显示类型初始化原则就显而易见了：

```
auto index = static_cast<int>(d * c.size());
```

需要铭记的要点
● “不可见”代理类型会导致 auto 从初始化表达式推导出来的类型是“错误”的。 ● 显示类型初始化原则强行让 auto 推导出你想要的类型。

第三章转移到 Modern C++

要说那些牛叉的特色（big-name features），C++11 和 C++14 可以列出很多，如，auto、智能指针（smart pointers）、移动语义（move semantics）、lambda、并发（concurrency），每一个都很重要。我专门安排一个章节来讲解。想要成为一个牛叉的 Modern C++程序员，掌握这些要点是必须的，但饭也要一口一口吃。从 C++98 转到 Modern C++，中间会遇到一系列问题，下面的每一个条款都会对某个特定问题做讲解。比如，何时应该用花括号创建对象而不是括号？为什么别名声明比 typedef 更好？constexpr 跟 const 到底有什么不同？const 成员函数与线程安全之间存在什么关系？还有好多好多，下面就逐条来讲解。

条款 7：创建对象时区分()和{}

按照你以往的认知，C++11 的对象初始化语法给人的感觉就是，要么就是选择太多，要么就是乱糟糟的。其实，通常来说，初始化也就是用括号、等号或者花括号：

```
int x(0); // initializer is in parentheses
int y = 0; // initializer follows "="
int z{ 0 }; // initializer is in braces
```

很多情况下，可能用等号和花括号一起：

```
int z = { 0 }; // initializer uses "=" and braces
```

这个条款里面，我就不考虑等号加花括号的语法了，因为 C++把它当做只有花括号来处理。

“乱糟糟”的情况是指，等号初始化经常会误导 C++初学者，认为那是一个赋值操作。对于内置类型（build-in types），如 int，没有区别，但对于用户自定义类型（user-defined types），区分赋值还是初始化就很重要，因为调用的是不用的函数：

```
Widget w1;       // 调用默认构造函数
Widget w2 = w1;  // 不是赋值，调用拷贝构造函数
w1 = w2;         // 赋值，调用 operator=
```

即使有多种初始化语法，C++98 还是有好些情况是没有办法传递想要的初始化的。例如，没有办法在创建 STL 容器时，给定一个数据集（如，1、3、5）。

为了消除多种初始化语法带来的混乱，同时也覆盖到所有的初始化场景，C++11 提供了统一的初始化方式（uniform initialization）：至少概念上来说，可以用在任何地方的初始化语法。语法基于花括号，所以我称它为花括号初始化（braced initialization）。“统一初始化”

是个设想，“花括号初始化”是一个语义实现。

花括号初始化可以让你实现以前无法传递的初始化。使用花括号，初始化一个容器就很简单了：

```
std::vector<int> v{ 1, 3, 5 }; // v's initial content is 1, 3, 5
```

花括号也用来初始化非静态（non-static）数据成员。这个是 C++11 的新功能，同样等号初始化语法也可以，但括号不行：

```
class Widget {
    …
private:
    int x{ 0 }; // fine, x's default value is 0
    int y = 0; // also fine
    int z(0); // error!
};
```

另一方面，不可拷贝的对象（如，std::atomics，参加条款 40）可以用花括号或括号来初始化，但不能用等号：

```
std::atomic<int> ai1{ 0 }; // fine
std::atomic<int> ai2(0); // fine
std::atomic<int> ai3 = 0; // error!
```

这样就很容易理解，为什么花括号初始化是“统一的”。对于 C++提供的三种初始化方式，只有花括号是可以在任何地方使用的。

有意思的是，对于内置（built-in）类型之间的隐形缩小转换（narrowing conversions），花括号是禁止的。如果花括号初始化里表达式的值不能保证可以表示成对象类型，代码就编译报错：

```
double x, y, z;
…
int sum1{ x + y + z };    // error! sum of doubles may not
                          // be expressible as int
```

括号和等号的初始化是不检查缩小转换的，因为那样会破坏以前写的代码（要沿用原来的特性，否则 C++98 的代码就不能很好的切换到 C++11 了——译者注）：

```
int sum2(x + y + z);  // okay (value of expression truncated to an int)
int sum3 = x + y + z; // 同上
```

花括号初始化另一个显著的特色就是，避免了 C++最令人烦恼的语法解析。C++规则的一个副作用就是，任何一个能把解析成声明的东东，都必须理解成声明。这个最令人烦恼的语法解析，大多数让开发者烦恼的地方是，当你是想要用默认构造函数创建对象时，反而被

解析成了一个函数声明。问题的根本原因就是，如果你想用参数调用构造函数，你可以写成下面这样：

```
Widget w1(10); // call Widget ctor with argument 10
```

但如果你想用类似的语法调用无参的构造函数，就变成了声明一个函数而不是对象：

```
Widget w2(); // 最令人烦恼的语法解析（most vexing parse）！
             // 声明了一个名称为 w2 返回 Widget 的函数!
```

函数声明时，形成列表不能用花括号，所以，用花括号默认构造对象，就没得问题：

```
Widget w3{}; // calls Widget ctor with no args
```

说一千道一万，花括号初始化语法可以用在很长宽泛的语境，它阻止了隐式缩小转换，避免了 C++最令人烦恼的语法解析。好的不得了！所以为什么这个条款的标题不叫“优先使用花括号初始化语法”呢？

花括号初始化的缺点是有时有些行为令人惊讶。这种情况主要是源于花括号初始化、std::initializer_list、构造函数重载三者之间的纠缠不清。它们之间的相互作用使得代码看上去是一回事，但时间上又是另外一回事。例如，条款 2 解释到，auto 声明的变量，用花括号初始化的话，推导出来的类型是 std::initializer_list，而用其他方式来声明变量的话，得到的是更加直观的类型。因此，你越中意 auto，你可能对花括号初始化越不来电。

如果不涉及到 std::initializer_list 形参，括号和花括号对构造函数调用一个样：

```
class Widget {
public:
    Widget(int i, bool b);    // 构造函数未声明std::initializer_list参数
    Widget(int i, double d);
    …
};

Widget w1(10, true);          // 调用第一个构造函数
Widget w2{10, true};          // 也调用第一个构造函数
Widget w3(10, 5.0);           // 调用第二个构造函数
Widget w4{10, 5.0};           // 也调用第二个构造函数
```

不过，如果一个或多个构造函数声明了 std::initializer_list 的参数，花括号初始化语法就坚定地优先调用 std::initializer_list 参数的重载构造。只要存在一种解析方式，编译器能用花括号初始化来调用 std::initializer_list 参数的构造函数，编译器就按那种方式执行。如果上面的 Widget 类扩展一个接受 std::initializer_list<long double>参数的构造函数，如下：

```
class Widget {
public:
```

```
    Widget(int i, bool b); // as before
    Widget(int i, double d); // as before
    Widget(std::initializer_list<long double> il); // added
    ...
};
```

Widget 对象 w2 和 w4 将用新的构造函数构造，即使相对来说，其他的构造函数跟实参跟匹配！！

```
Widget w1(10, true); // 使用括号，调用第一个构造函数
Widget w2{10, true}; // 使用花括号，现在调用std::initializer_list构造函数
                     // (10 and true 转换成 long double)
Widget w3(10, 5.0);  // 使用括号，调用第二个构造函数
Widget w4{10, 5.0};  // 使用花括号，现在调用std::initializer_list构造函数
                     // (10 and 5.0 转换成 long double)
```

即使应该调用拷贝和移动构造函数的，也逃不过去：

```
class Widget {
public:
    Widget(int i, bool b);    // as before
    Widget(int i, double d); // as before
    Widget(std::initializer_list<long double> il); // as before
    operator float() const; // convert to float
};

Widget w5(w4);   // 使用括号，调用拷贝构造函数
Widget w6{w4};   // 使用花括号，调用std::initializer_list构造函数
                 // (w4转成float, 然后float 转成long double)
Widget w7(std::move(w4)); // 使用括号，调用移动构造函数
Widget w8{std::move(w4)}; // 使用花括号，调用std::initializer_list构造函数
                          // (原因同w6)
```

编译器坚定地将花括号初始化匹配到带 std::initializer_list 参数的构造函数，即使最匹配的 std::initializer_list 构造函数都不能调用，如下：

```
class Widget {
public:
    Widget(int i, bool b); // as before
    Widget(int i, double d); // as before
    Widget(std::initializer_list<bool> il);    // 元素类型现在是bool,
    ...                                        // 没有隐式转换函数
};

Widget w{10, 5.0}; // error! requires narrowing conversions
```

此处，编译器将忽略前两个构造函数（第二个其实是最匹配实参的），并且尝试调用接

受 std::initializer_list<bool>的构造函数，那样就需要将一个 int（10）和一个 double（5.0）转成 bool，这两种都是缩小转换（bool 不能表示那两个值），而花括号初始化是禁止缩小转换的，所以这个调用无效，代码编译不通过。

只有当没有办法将花括号初始化的实参类型转换成 std::initializer_list 中的类型时，编译器才会回到普通的重载构造函数。例如，如果我们将 std::initializer_list<bool>替换成 std::initializer_list<std::string>，其他构造函数就会调用了，因为没有方法可以将 int 和 bool 转成 std::string：

```
class Widget {
public:
    Widget(int i, bool b); // as before
    Widget(int i, double d); // as before
    // std::initializer_list 元素类型现在是 std::string
    Widget(std::initializer_list<std::string> il);
    … // 不存在隐式类型转换函数
};

Widget w1(10, true); // 使用括号，仍然调用第一个
Widget w2{10, true}; // 使用花括号，现在调用第一个
Widget w3(10, 5.0);  // 使用括号，仍然调用第二个
Widget w4{10, 5.0};  // 使用花括号，现在调用第二个
```

到这儿，花括号初始化和构造函数重载差不多讲完了，但还有一个有趣的边界情况需要强调一下。假设你用一个空的花括号来构造一个支持默认构造函数和 std::initializer_list 构造函数的对象，空的花括号代表什么意思呢？如果是表示“无参数”，应该调用默认构造函数，但如果表示的是“空的 std::initializer_list”，就应该用“空 std::initializer_list”构造。

规则规定是调用默认构造函数。空的花括号代表无参数，而不是“空 std::initializer_list”：

```
class Widget {
public:
    Widget(); // default ctor
    Widget(std::initializer_list<int> il); // std::initializer_list ctor
    …  // no implicit
};     // conversion funcs

Widget w1;  // 调用默认构造函数
Widget w2{}; // 也调用默认构造函数
Widget w3(); // most vexing parse! declares a function!
```

如果你想用“空 std::initializer_list”调用 std::initializer_list 构造函数，你就要在“空花括

号”外面再包一层括号或者花括号，让它变成一个实参：

```
Widget w4({}); // 用空列表调用std::initializer_list构造函数
Widget w5{{}}; // 同上
```

此时此刻，花括号初始化、std::initializer_list 和构造函数重载三者之间晦涩难懂的规则在你脑袋里面绕来绕去，你可能想知道，这些对日常编程能多大影响。你可能想不到，直接受影响的一个类就是 std::vector。std::vector 有个 non-std::initializer_list 构造函数，可以初始化容器的大小和每个初始元素的值，同时也有一个 std::initializer_list 构造函数，可以指定容器中的初始值。如果你创建一个数字类型的 std::vector（如，std::vector<int>），并且传递两个实参，使用括号或是花括号将是天壤之别：

```
std::vector<int> v1(10, 20); // 使用non-std::initializer_list构造函数
                             // 创建10个元素的std::vector,
                             // 所有元素的值都是20
std::vector<int> v2{10, 20}; // 使用std::initializer_list构造函数
                             // 创建2个元素的std::vector,
                             // 元素值分别是10和20
```

让我们回顾一下 std::vector 和括号、花括号、构造函数重载抉择的规则细节。讲两点题外话，首先，作为一个类的作者，你需要知道，如果你的重载构造函数包含一个或多个接受 std::initializer_list 的函数，那么用户使用花括号初始化的话，就只能调用 std::initializer_list 构造，因此，构造函数的最好设计就是，无论用户是用括号还是花括号，构造结果都不受影响；另外一点，从 std::vector 接口设计上吸取教训，避免这样的设计。

一种隐形的情况是，你的类本来是没有 std::initializer_list 构造的，后来添加了一个，那么用户的花括号初始化就由原来的构造变成调用 std::initializer_list 构造了。当然，对于重载来说，这种情况在添加一个重载函数后，都可能发生：从原来的重载调用变成调用新的重载函数。std::initializer_list 构造函数最大的不同就是，它与其他构造函数不是平等竞争关系，而是隐藏了其他构造函数。所以，添加这样的重载函数一定要非常慎重！！

第二个经验教训就是，作为类的用户，你在创建对象时，要仔细选择是用括号还是花括号。大多数开发者选择一种作为默认方式，只有必要的时候才用另外一种。默认用花括号的人们，主要是因为花括号的广泛适用性、阻止了缩小转换、避免了 C++最令人烦恼的语法解析。这些人知道有些情况（比如，创建一个给定大小和初始值的 std::vector）是必须用括号的。相反的，默认用括号的小伙伴们，主要是因为它与 C++98 的传统一致，而且对象创建也不会被 std::initializer_list 带到沟里去，他们知道有时候只有花括号能用（如，创建特定值的容器）。不能一定说谁好谁坏，所以，我的建议是，选择一种作为默认方式即可。

如果你是一个模板作者，用括号还是花括号都是泪，因为，通常来说，你不可能知道应该用哪个。例如，假如你想用任意个数的实参创建一个任意类型的对象，一个可变参数模板如下：

```
template<typename T,      // type of object to create
         typename... Ts>  // types of arguments to use
void doSomeWork(Ts&&... params)
{
    create local T object from params...
    ...
}
```

上面的伪代码可以写成下面两种方式（std::forward 请参考条款 25）：

```
T localObject(std::forward<Ts>(params)...); // using parens
T localObject{std::forward<Ts>(params)...}; // using braces
```

考虑下面的调用：

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

如果 doSomeWork 用括号来创建 localObject，返回值就是一个有 10 个元素的 std::vector，如果用花括号，结果就是 2 个元素的 std::vector，哪个是正确的呢？doSomeWork 的作者不知道，只有调用者才知道。

这也恰恰是 std::make_unique 和 std::make_shared（参考条款 21）面临的问题，这些函数解决问题的方法就是，内部使用括号，并且在接口部分进行说明。

需要铭记的要点
● 花括号初始化是使用最广泛的初始化语法，它阻止了缩小转换，避免了 C++最令人烦恼的语法解析。 ● 进行构造函数重载选择时，花括号初始化只要有可能，就匹配到 std::initializer_list 构造，其他构造函数再匹配都没用。 ● 用两个实参来创建一个 std::vector<numeric type>对象，选择括号还是花括号会导致天壤之别。 ● 在模板中到底是用括号还是花括号来创建对象，这是个挑战。

条款 8：优先使用 nullptr，而非 0 或 NULL

有个约定：字面上 0 是一个 int，而不是指针。如果 C++在一个只能使用指针的语境下发现 0，那么就勉强将 0 理解成空指针，但这是一个无奈之举。C++基本策略是 0 是 int 不是指针。

实际上，NULL 也是同样的。对 NULL 而言，仍有一些细节上的不确定性，因为实现上允许赋予 NULL 一个 int 以外的整数类型（如，long）。这不常见，但这没关系，因为此处的问题不是 NULL 的确切类型而是 0 和 NULL 都不具备指针类型。

在 C++98 中，这意味着，重载指针和整数类型的函数，得到的结果会令人吃惊。传递 0 或者 NULL 给重载函数，永远不会调用指针重载的那个函数：

```
void f(int); // three overloads of f
void f(bool);
void f(void*);
f(0);        // 调用f(int)，而不是f(void*)
f(NULL);     // 有可能编译不过，但调用f(int)。永远不会调用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 被定义为 0L（即 long 类型的 0），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。这个函数调用有个有意思的事情，源代码的字面意思（我在用空指针 NULL 调用 f）和它的真实意思（我在用某种整数调用 f）是矛盾的。这种违背直觉的行为，正是 C++98 程序员原则上避免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，而不用更好的 nullptr。

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 的实际类型是 std::nullptr_t，而且字啊一个神奇般的循环定义下，std::nullptr_t 又定义为 nullptr 类型。std::nullptr_t 可以隐式的转换为所有的原始指针类型，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 void*重载（即指针重载），因为 nullptr 不能被视为任何整数类型：

```
f(nullptr); // calls f(void*) overload
```

使用 nullptr 而不是 0 或者 NULL，可以避免重载抉择上令人吃惊的行为，但是它的优势不仅于此。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) {
    ...
}
```

如果你不知道（或者很难找到）findRecord 的返回值类型，那就不清楚 result 到底是一个指针还是整数类型了。毕竟，用来测试 result 的 0 两种类型都满足。另一方面，你如果看到下面的代码：

```
auto result = findRecord( /* arguments */ );
if (result == nullptr) {
    ...
}
```

明显就没有歧义了： result 一定是个指针类型。

nullptr 用于模板的话，更加光芒四射了。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);       // call these only when
double f2(std::unique_ptr<Widget> upw);    // the appropriate
bool f3(Widget* pw);                       // mutex is locked
```

传递空指针给这些函数，调用如下：

```
std::mutex f1m, f2m, f3m; // mutexes for f1, f2, and f3
using MuxGuard = // C++11 typedef; see Item 9
    std::lock_guard<std::mutex>;
...
{
    MuxGuard g(f1m);     // lock mutex for f1
    auto result = f1(0); // pass 0 as null ptr to f1
}                        // unlock mutex
...
{
    MuxGuard g(f2m);          // lock mutex for f2
    auto result = f2(NULL);   // pass NULL as null ptr to f2
}                             // unlock mutex
...
{
    MuxGuard g(f3m);              // lock mutex for f3
    auto result = f3(nullptr);    // pass nullptr as null ptr to f3
}                                 // unlock mutex
```

挺遗憾的，前两个函数调用中没有使用 nullptr，但是上面的代码是可以工作的，这才是最重要的。然而，代码中调用的重复模式——锁定互斥量，调用函数，解锁互斥量——更不爽。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))）搞得你晕头转向的话，跳到条款 3 看看。在 C++14 中，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); // error!
...
auto result2 = lockAndCall(f2, f2m, NULL); // error!
...

auto result3 = lockAndCall(f3, f3m, nullptr); // fine
```

可以这样写，但是就如注释中指明的，三种情况里面有两种是无法编译通过的。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，这就是实例化调用 lockAndCall 时 ptr 的类型。不幸的是，这意味着在 lockAndCall 中调用 func 时传的是 int，这和 f1 期望接受的参数 std::share_ptr<Widget>不兼容。传入到 lockAndCall 的 0 尝试来表示一个空指针，但是真正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 会导致一个类型错误。使用 0 调用 lockAndCall 会失败，

是因为在模板中，一个 int 类型传给一个参数是 std::share_ptr<Widget>的函数。

对调用 NULL 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，当把 ptr（一个 int 或者类 int 的类型）传给 f2，将产生一个类型错误，因为 f2 期望的是一个 std::unique_ptr<Widget>参数。

相反，使用 nullptr 没有问题。当把 nullptr 传递给 lockAndCall，ptr 的类型推导成 std::nullptr_t 。当把 ptr 传递给 f3，std::nullptr_t 隐形转换成 Widget*，因为 std::nullptr_t 可以隐式转换为任何类型的指针。

对于 0 和 NULL，模板类型推导得出了“错误”类型（即它们的真实类型，而不是为了表示空指针而退化出来的类型），这就是使用 nullptr 而不是 0 或 NULL 的最引人注目的原因。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，显而易见，当你需要用到空指针时，选择 nullptr，跟 0 或者 NULL 说再见。

需要铭记的要点
● 优先使用 nullptr，而不是 0 和 NULL ● 避免在整数和指针类型上重载

条款 9：优先使用别名声明，而非 typedefs

我坚信，大家也应该都同意，使用 STL 容器是个好主意，并且我希望，条款 18 可以说服你使用 std::unique_ptr 也是个好主意，但是我想，我们中间没有人喜欢写像 std::unique_ptr<std::unordered_map<std::string, std::string>>这样的代码。想想看，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，推荐一下 typedef ：

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

但是 typedef 的 C++98 气息太浓。当然，在 C++11 下工作没问题，但 C++11 也提供了别名声明（alias declarations）：

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

既然 typedef 和别名声明干的活是一样的，那么就有理由去猜测，有没有一个强有力的技术原因，来告诉我们，应该优先选择哪个。

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，使用别名声明可以使涉及到函数指针的类型一目了然：

```
// FP is a synonym for a pointer to a function taking an int and
// a const std::string& and returning nothing
typedef void (*FP)(int, const std::string&); // typedef
// same meaning as above
using FP = void (*)(int, const std::string&); // alias declaration
```

当然，上面两种形式都特别容易看明白，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别名声明而不是 typedef 的强有力的理由。

但是，模板就是一个强有力的理由。尤其是别名声明有可能是模板化的（称为别名模板（alias template）），而 typedef 只能说句“臣妾做不到”。在 C++98 中，必须将 typedef 和要表达的类型一起内嵌到模板化的结构体中，而 C++11 程序员用别名模板就很简洁明了。举个栗子，给一个使用个性化的分配器 MyAlloc 的链表定义一个标识符。使用别名模板，这就是小菜一碟：

```
template<typename T>                          // MyAllocList<T>
using MyAllocList = std::list<T, MyAlloc<T>>; // is synonym for
                                              // std::list<T,MyAlloc<T>>
```

```
MyAllocList<Widget> lw;                         // client code
```

用 typedef，那就麻烦多了：

```
template<typename T>                            // MyAllocList<T>::type
struct MyAllocList {                            // is synonym for
    typedef std::list<T, MyAlloc<T>> type;      // std::list<T,MyAlloc<T>>
};

MyAllocList<Widget>::type lw;                   // client code
```

更惨的是，如果想在模板内部中使用“typedef”的类型来创建链表对象，你还要在类型名称前面加上 typename 关键字：

```
template<typename T>
class Widget {                              // Widget<T> contains
private:                                    // a MyAllocList<T>
    typename MyAllocList<T>::type list;     // as a data member
    ...
};
```

此处，MyAllocList<T>::type 引用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之一就是在依赖类型的名称前面必须加上 typename。

如果 MyAllocList 定义成一个别名声明，就不需要使用 typename（笨重的::type 后缀也可以省去）：

```
template<typename T>
using MyAllocList = std::list<T, MyAlloc<T>>; // as before

template<typename T>
class Widget {
private:
    MyAllocList<T> list; // no "typename",
    ...                  // no "::type"
};
```

对你来说，别名模板 MyAllocList<T>看上去跟 MyAllocList<T>::type 一样，依赖于模板参数 T，但你不是编译器，当编译器处理 Widget 模板，并遇到别名模板 MyAllocList<T>时，编译器知道 MyAllocList<T>是一个类型名称，因为 MyAllocList 是一个别名模板：它一定会命名一个类型。因此，MyAllocList<T>是一个非依赖类型（non-dependent type），既不需要也不允许用 typename 关键字。

另一方面，当编译器在 Widget 模板中遇到 MyAllocList<T>::type 时，编译器不能确定它

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，因为人类有可能会写出这样的代码。

例如，一些被误导的灵魂可能会炮制出这样的代码：

```
class Wine { … };

template<>                      // MyAllocList specialization
class MyAllocList<Wine> {       // for when T is Wine
private:
    enum class WineType         // see Item 10 for info on "enum class"
        { White, Red, Rose };
    WineType type;              // in this class, type is a data member!
    …
};
```

正如你看到的，MyAllocList<Wine>::type 并不是一个类型。如果 Widget 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完全依赖于传入的 T 是什么，这也是编译器坚持要求你在类型前面加上 typename 的原因。

如果你曾经做过模板元编程（TMP），你肯定会碰到这样的需求，在模板类型参数基础上创建一个改进的类型。例如，给定一个类型 T，你有可能想去掉 T 的所有 const 或引用限定符，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，那就太遗憾了，因为想要成为一个真正牛叉的 C++程序员，你至少要熟悉 C++这方面的基本概念。你可以看一些 TMP 的例子，包括条款 23 和条款 27 的类型转换。）

C++11 给你提供了工具来完成这类转换的工作，叫类型特性（type traits），就是头文件<type_traits> 中各式各样的模板。这个头文件中有数十个类型特性（type traits），但是并不是所有都执行类型转换。给定一个你想进行类型转换的类型 T，得到的类型结果是 std::transformation<T>::type ，例如：

```
std::remove_const<T>::type          // yields T from const T
std::remove_reference<T>::type      // yields T from T& and T&&
std::add_lvalue_reference<T>::type  // yields T& from 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。总之，我的目的不是指导你 type traits，而是注意，

使用这些类型转换时，总是以::type 结尾。当你应用到模板中的类型参数时（实际代码中会经常用到），你必须在每次使用前冠以 typename，原因就是 C++11 的 type traits 是通过内嵌 typedef 到一个模板化的结构体来实现的。没错，就是通过 typedef 技术来实现的，就是我一直在告诉你不如别名模板的那个技术。

这是一个历史遗留问题，但是我们略过不表。因为标准委员会意识到别名模板更好的时候，已经有点晚了。C++14 沿袭了 C++11 的类型转换，同时也增加了统一格式的别名模板：对于 C++11 中的每个类型转换 std::transformation<T>::type，C++14 都有一个对应的的别名模板 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>                // C++14 equivalent
std::remove_reference<T>::type        // C++11: T&/T&& → T
std::remove_reference_t<T>            // C++14 equivalent
std::add_lvalue_reference<T>::type    // C++11: T → T&
std::add_lvalue_reference_t<T>        // C++14 equivalent
```

C++11 的实现在 C++14 中依然有效，但是我不知道你还有什么理由再用它们。即便你不熟悉 C++14，自己写别名模板也是小儿科。仅仅依赖 C++11 的语言特性，小孩甚至都可以整出来，对吗？如果你碰巧有一份 C++14 标准的电子拷贝，这就更简单了，因为需要做的就是一些复制粘贴。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;
```

看到了吧？简单的不能再简单了。

需要铭记的要点
● typedef 不支持模板化，别名声明支持。 ● 别名模板省去了“::type”后缀，而且，在模板里，typedef 一般要加“typename”前缀 ● C++14 提供了 C++所有类型特征转换的别名模板

条款 10：优先使用 scoped enums，而非 unscoped enums

一般而言，在花括号里面声明的变量名会限制它在括号外不可见。但是这对于 C++98 风格的 enum 中的枚举元素并不成立。枚举元素和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 are
                                  // in same scope as Color
auto white = false;               // error! white already
                                  // declared in this scope
```

事实就是枚举元素泄露到枚举类型所在的作用域中，对于这种类型的 enum 官方称之为 unscoped。（文章中不对 scoped 和 unscoped 作翻译，因为 enum 都是有作用域的，只是说 C++98 的方式作用域限定的不太好，将 unscoped 翻译成无作用域的话，感觉不太贴切——译者注）在 C++11 中对应的 scoped enum 不会造成这种泄露：

```
enum class Color { black, white, red };   // black, white, red
                                          // are scoped to Color
auto white = false;                       // fine, no other
                                          // "white" in scope
Color c = white;                          // error! no enumerator named
                                          // "white" is in this scope
Color c = Color::white;                   // fine
auto c = Color::white;                    // also fine (and in accord
                                          // with Item 5's advice)
```

因为 scoped enum 是通过"enum class"来声明的，它们有时被称作枚举类（enum class）。

scoped enum 可以减少命名空间的污染，就这条理由，这足以让我们选择它而不是 unscoped。除此之外，scoped enum 还有一个令人不可抗拒的优势：它们的枚举元素是强类型的。unscoped enum 会将枚举元素隐式转换为整数类型（并且再从整数类型转换为浮点类型）。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 }; // unscoped enum

std::vector<std::size_t>          // func. returning
    primeFactors(std::size_t x);  // prime factors of x

Color c = red;
...
if (c < 14.5) {                   // compare Color to double (!)
    auto factors =                // compute prime factors
```

```
        primeFactors(c);          // of a Color (!)
    ...
}
```

在 "enum" 后增加一个 "class" ，就可以将 unscoped enum 转换为 scoped enum，变成 scoped enum 之后，那就完全是另一个结果了。在 scoped 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 }; // enum is now scoped

Color c = Color::red;                   // as before, but
...                                     // with scope qualifier
if (c < 14.5) {                         // error! can't compare
                                        // Color and double
    auto factors =                      // error! can't pass Color to
        primeFactors(c);                // function expecting std::size_t
    ...
}
```

如果你就是想将 Color 类型转换成一个其他类型，使用类型强制转换（cast）可以满足你这种荒唐的需求：

```
if (static_cast<double>(c) < 14.5) {  // 奇怪的代码，但是合法
    auto factors =                    // 不可靠，但可以编译通过
        primeFactors(static_cast<std::size_t>(c));
    ...
}
```

相较于 unscoped enum，scoped enum 还有第三个优势，因为 scoped 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; // error!
enum class Color; // fine
```

这是一个误导。在 C++11 中，unscoped 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由编译器决定的潜在类型。对于一个 unscoped enum 如 Color，

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

编译器有可能选择 char 作为潜在的类型，因为只需表示三个值。然而一些枚举类型有很大的取值跨度，如：

```
enum Status { good = 0,
    failed = 1,
```

```
    incomplete = 100,
    corrupt = 200,
    indeterminate = 0xFFFFFFFF
};
```

这里需要表示的值范围从0到0xFFFFFFFF。除非是在一个特殊的机器上(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。

为了更高效的使用内存，编译器通常为枚举类型选择潜在类型的原则是：既可以充分表示枚举元素的取值范围，但又占用内存最小的。在某些情况下，编译器可以进行速度优化而不是内存优化，在那种情况下，编译器可能不会选择占用内存最小的潜在类型，但是编译器肯定是尽可能地优化内存大小。为此，C++98 就只支持枚举类型的定义（所有枚举元素被列出来），而不允许声明。这样可以保证在枚举类型使用之前，编译器已经给每个枚举类型选定了潜在类型。

不能事先声明枚举类型有好些缺点，最显著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
            };
```

可能整个系统都会用到这个枚举体，因此系统中每个依赖它的头文件都要包含它。如果需要引入一个新的状态：

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

就算只有一个子系统——甚至只有一个函数！——用到这个新的枚举元素，有可能整个系统的代码都需要重新编译。这种事情就令人讨厌了。C++11 恰好可以避免这个问题。例如，下面有一个完全有效的 scoped enum 声明，还有一个函数将它作为参数：

```
enum class Status; // forward declaration
void continueProcessing(Status s); // use of fwd-declared enum
```

如果 Status 的定义修改了，包含这个声明的头文件不需要重新编译。而且，如果 Status 修改了（即，增加 audited 枚举元素），但 continueProcessing 的行为也可能不受影响（如，continueProcessing 没有使用 audited），continueProcessing 的实现也不需要重新编译。

但是既然编译器需要在使用枚举体之前知道它的大小，为什么这样的前置声明，C++11 可以，而 C++98 却不行呢？原因很简单，scoped enum 的潜在类型是已知的，unscoped enum 必须指定类型。

scoped enum 默认的潜在类型是 int ：

```
enum class Status; // underlying type is int
```

如果默认类型不符合要求，你还可以指定一个（override）：

```
enum class Status: std::uint32_t; // underlying type for
                                  // Status is std::uint32_t
                                  // (from <cstdint>)
```

无论哪种形式，编译器都知道 scoped enum 中的枚举元素大小。你也可以用同样的方式，给 unscoped enum 指定潜在类型，那样就可以前置声明了：

```
enum Color: std::uint8_t;     // fwd decl for unscoped enum;
                              // underlying type is
                              // std::uint8_t
```

枚举定义的时候，也可以同时指定潜在类型：

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

scoped enum 肯定比 unscoped enum 好得多，它既可以避免命名空间污染，也不允许诡异的隐式类型，但你可能想不到，至少有一种情形，unscoped enum 是更有用的，那就是引用 C++11 的 std::tuples 中的某个字段时。例如，假设我们有一个元组（tuple），元组中保存着姓名、电子邮件地址，以及用户在社交网站的影响力数值：

```
using UserInfo =              // type alias; see Item 9
    std::tuple<std::string,   // name
    std::string,              // email
    std::size_t> ;            // reputation
```

尽管注释已经说明元组的每部分代表什么意思，但是当你在其他源文件里面遇到下面这

样的代码时，那个注释可能就起不到帮助作用了：

```
UserInfo uInfo; // object of tuple type
...
auto val = std::get<1>(uInfo); // get value of field 1
```

作为一个程序员，你有很多事要做。难道你还真的要去记住元组的第一个字段对应的是用户的电子邮件地址吗？我想谁都不愿意。要避免死记这些东西，使用一个 unscoped enum，把名字和域的编号联系在一起来，就万事大吉了：

```
enum UserInfoFields { uiName, uiEmail, uiReputation };
UserInfo uInfo; // as before
...
auto val = std::get<uiEmail>(uInfo); // ah, get value of email field
```

上面代码能正常工作的原因就是，std::get()要求 std::size_t，而 UserInfoFields 正好可以隐式转换成 std::size_t。

如果用 scoped enum 来写类似的代码，就显得有点冗长了：

```
enum class UserInfoFields { uiName, uiEmail, uiReputation };
UserInfo uInfo; // as before
...
auto val =
    std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>(uInfo);
```

我们可以一个模板，接受一个枚举元素参数，返回对应的 std::size_t 类型的值，那样就可以减少这种冗余，但这种做法有点麻烦。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。根据条款 15，这意味着它必须是一个 constexpr 函数。

实际上，它也必须是一个 constexpr 函数模板，因为它要对任何类型的枚举有效。如果要实现这种泛化格式，返回类型也要泛化，不是返回 std::size_t，而是返回枚举体的潜在类型，通过 std::underlying_type 类型转换来获得潜在类型(关于类型转换的信息，参见条款 9)。最后将这个函数声明为 noexcept（参见条款 14），因为我们知道它永远不会产生异常。我们可以写出这样的函数模板 toUType，接受任何的枚举元素，返回这个元素在编译阶段的常数：

```
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，可以将 std::underlying_type<E>::type 替换成 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);
}
```

在 C++14 中，还可以用更加简洁的 auto 返回值类型（参见条款 3）：

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

无论写哪种形式，toUType 允许我们像下面这样访问一个元组的某个域：

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

这样依然比 unscoped enum 要复杂，但是它可以避免命名空间污染和容易忽略的枚举元素类型转换。很多时候，你最好还是多敲击一些额外的代码，避免陷入 unscoped enum 的技术陷阱中。

需要铭记的要点
● C++98 风格的枚举，现在称作为 unscoped enum。 ● scoped enum 的枚举元素只在 enum 内可见，元素只能强制转换成其他类型。 ● scoped enum 和 unscoped enum 都可以指定潜在类型。scoped enum 默认是 int，unscoped enum 没有默认类型。 ● scoped enum 总是能前置声明，unscoped enum 只有指定了潜在类型，才可以前置声明。

条款 11：优先使用 delete 关键字删除函数，而非 private 却又不实现的方式

如果你提供代码给其他开发人员，并且还不想让他们调用某个特定函数，一般情况下，你只要不声明这个函数就可以了。没有函数声明，也就没有函数可以调用。简单，爽！但有时候 C++默认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，即这个成员函数是在需要的时候由 C++自动生成的。条款 17 详细地讨论了这种函数，但是在这里，我们仅仅考虑拷贝构造函数和拷贝赋值运算符（operator=）。这一节大篇幅地讲解了那些在 C++98 中是个普通的实现，但 C++11 中已经废弃了，C++11 有更好的实现。在 C++98 中，如果你想抑制一个成员函数的使用，这个成员函数通常是拷贝构造函数、拷贝赋值运算符，或者两者都有。

C++98 阻止这类函数的方法是将这些函数声明为 private，并且不去定义它们。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直接或间接地继承于这个类。拷贝输入和输出流是没必要的，因为不知道应该采取何种行为。比如，一个表示一系列输入数值的 istream 对象，一些已经读入内存，有些可能后续被读入。如果拷贝一个输入流，是不是应该将已经读入的数据和将来要读入的数据都拷贝一下呢？处理这类问题最简单的方法是不定义这些操作，IO 流的拷贝就是这么做的。

为了使 istream 和 ostream 类不可拷贝，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& );            // not defined
    basic_ios& operator=(const basic_ios&);  // not defined
};
```

将这些函数声明为私有来阻止用户调用，故意不定义它们是因为，如果有代码访问这些函数（通过成员函数或者友好类），在链接的时候会报没有定义的错误。

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”将拷贝构造函数和拷贝赋值函数标记为删除的函数（deleted function）。在 C++11 中 basic_ios 定义成：

```
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不同之处，表面上看，除了时尚一些，没有啥区别，但实际上区别比你想象的要大的多。删除的函数不能通过任何方式调用，即便是成员函数或者友好函数，试图拷贝 basic_ios 对象，也会导致编译失败。这是对 C++98 行为的提升，因为在 C++98 中直到链接时才会诊断出这个错误。

方便起见，删除函数声明为公有，而非私有。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，一些编译器只报告该函数是私有的，而实际上这个函数是不可用的，跟可访问性没有半毛钱关系。牢记这一点，把私有且未定义（private-and-not-defined）的成员函数改为对应的删除函数，因为这个函数变成公有之后，通常可以得到可读性更好的错误信息。

删除函数一个重要的优势是任何函数都可以删除，而 private 这种方式，只适用于成员函数。举个例子，假设我们有个非成员函数，接受一个整数参数，返回是不是幸运数字：

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

C++继承于 C，意味着，其他类型很多可以隐式转换为 int 类型，但是，有些调用虽然可以编译，却没有任何意义：

```
if (isLucky('a')) ···      // is 'a' a lucky number?
if (isLucky(true)) ···     // is "true"?
if (isLucky(3.5)) ···      // should we truncate to 3 before
                           // checking for luckiness?
```

如果幸运数字一定要是一个整数，我们当然希望能到禁止上面那种形式的调用。

实现的一个方法是，将那些想禁止的类型重载函数声明为删除的：

```
bool isLucky(int number);         // original function
bool isLucky(char) = delete;      // reject chars
bool isLucky(bool) = delete;      // reject bools
bool isLucky(double) = delete;    // reject doubles and floats
```

（对 double 重载的注释写到：double 和 float 类型都被拒绝，这可能会令你感到吃惊，但请你回想一下，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，你就就不会感到奇怪了。以 float 类型调用 isLucky 总是调用对应的 double 重载，而不是 int 重载。结果就是将 double 类型的重载删除将会阻止 float 类型的调用编译。）

尽管删除函数不能调用，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删除声明，对 isLucky 不必要的调用会被拒绝：

```
if (isLucky('a')) … // error! call to deleted function
if (isLucky(true)) … // error!
if (isLucky(3.5f)) … // error!
```

还有一个删除函数可以实现，而私有成员函数无法实现的小窍门，就是可以阻止那些应该被禁用的模板实例化。举个例子，假设你需要一个使用内嵌指针的模板（第 4 章建议使用智能指针而不是原始指针）：

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

在指针家族中，有两个特殊的指针。一个是 void*指针，因为没有办法对它们进行解引用、递增或者递减等操作，另一个是 char*指针，因为它们往往表示指向 C 风格的字符串，而不是指向独立字符的指针。这些经常需要特殊处理，在 processPointer 模板这个例子中，假设要拒绝这两个类型的调用。也就是说，不可能以 void*或者 char*调用 processPointer。

这很容易实现，仅仅需要删除那些实例化就行：

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

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

现在，使用 void*或者 char*调用 processPointer 就不合法了，而使用 const void*或者 const char*调用也不合法，因此这些实例化也需要删除：

```
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，然后你就可以愉快的使用其他标准字符类型指针了，包括 std::wchar_t、std::char16_t 和 std::char32_t。

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特例，这种方式行不通，因为成员函数模板特例的访问等级不能与模板主体不一样。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，事实上是无法编译通过的：

```
class Widget {
public:
    …

    template<typename T>
    void processPointer(T* ptr)
    { … }
private:
    template<> // error!
    void processPointer<void>(void*);
};
```

这里的问题是，模板的特例必须要写在命名空间的作用域，而不是类的作用域。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等级。它们可以在类的外面声明为删除的（也在命名空间的作用域）：

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

    …
};

template<>                                          // still
void Widget::processPointer<void>(void*) = delete; // public,but deleted
```

事实上，C++98 中声明函数为私有且未定义，是达不到 C++11 中删除函数的效果的，C++98 的方式做不到那么好，它离开了类是不好使的，在类的内部也不是总是好使，也许编译没问题，但链接的时候可能就报错了。所以还是坚持使用删除函数吧。

需要铭记的要点
● 优先使用删除函数，而不是私有且未定义的函数。 ● 任何函数都可以标记为删除，包括非成员函数和模板实例化。

条款 12：使用 override 关键字声明覆盖的函数

C++中的面向对象编程都是涉及到类、继承和虚函数。其中最基础的部分就是，派生类中的虚函数会覆盖掉基类中对应的虚函数实现。但是令人心痛是，虚函数覆盖很容易跑偏。这部分设计让人感觉，好像墨菲定律（Murphy's Law）只是用来膜拜，不需要遵从。

因为 “覆盖（overriding）”听上去非常像 “重载（overloading）”，但实际上完全没有关系，所以，我要清楚的告诉你，正是有了虚函数覆盖，才可能通过基类的接口来调用派生类的函数：

```
class Base {
public:
    virtual void doWork(); // base class virtual function
    …
};

class Derived: public Base {
public:
    virtual void doWork();    // 覆盖Base::doWork，这儿"virtual"关键字可选
    …
};

std::unique_ptr<Base> upb =       // 创建指向派生类对象的基类指针
    std::make_unique<Derived>(); // see Item 21 for info on
…                                // std::make_unique
upb->doWork();   // 用基类指针调用doWork，实际调用的是派生类的函数
```

如果要实现覆盖，必须满足以下条件：

- 基类的函数必须声明为 virtual。
- 基类和派生类的函数名称必须是完全一样的（析构函数除外）。
- 基类和派生类的函数参数类型必须完全一样。
- 基类和派生类的函数 const 特性必须完全一样。
- 基类和派生类的函数返回值类型和异常声明必须是兼容的。

以上仅仅是 C++98 中要求的部分，C++11 又增加了一条：

- 函数的引用后置修饰符必须完全一样。成员函数的引用后置修饰符是很少关注的 C++11 特性，所以你之前没有听说过也不要惊讶。这些修饰符限定函数只能被左值或者右值对象使用。成员函数不需要声明为 virtual：

```
class Widget {
```

```
public:
    ...
    void doWork() &;  // 只有*this是左值（lvalue）时才能调用
    void doWork() &&; //只有*this是右值（rvalue）时才能调用
};
...
Widget makeWidget(); // factory function (returns rvalue)
Widget w;            // normal object (an lvalue)
...
w.doWork();              // calls Widget::doWork for lvalues
                         // (i.e., Widget::doWork &)
makeWidget().doWork();   // calls Widget::doWork for rvalues
                         // (i.e., Widget::doWork &&)
```

稍后我们会更多介绍带有引用后置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后置修饰符，派生类中对应的那个也必须要有完全一样的引用后置修饰符。如果没有，派生类中声明的那个函数也会存在，但是它不会覆盖基类的函数。

对函数覆盖有这么多要求，意味着，一个小的失误可能会导致天壤之别。在函数覆盖中出现的错误通常还是合法的，但是它的结果却并不是你想要的。所以当你犯了某些错误的时候，你不能指望编译器来通知你。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函数覆盖——没有一个派生类的函数绑定到基类的对应函数上。你能找到每个函数的问题所在吗？即为什么派生类中的函数没有覆盖基类的同名函数。

```
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 const 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（lvalue-qualified），但是在 Derived 中是右值修饰符（rvalue-qualified）
- mf4 在 Base 中没有声明为 virtual

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，但也可能不是这样。经过我的检查，发现在两个编译器上，上边的代码被全盘接受而没有发出任何警告，我是设置了所有警告都输出的。（其他的编译器输出了部分警告信息，但也不全。）

正确声明派生类的覆盖函数是如此的重要，又如此容易出错，因此 C++11 给你提供了一种显式声明 override，表明派生类的函数是要覆盖对应的基类函数的。把这个规则应用到上面的代码，得到下面的派生类代码：

```
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，因为当用这种方式写的时候，编译器会把覆盖相关的所有问题都揭露出来。这也正是你想要的，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（假设目标就是 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"最好，但也不是必须要加
};
```

注意在这个例子中，有一个情况是要声明 mf4 为 Base 类中的虚函数。绝大部分覆盖相关的错误都发生在派生类中，但是也有可能在基类中有不正确的代码。

将派生类中的覆盖声明为 override，不仅仅是让编译器在你想要覆盖而没有覆盖的时候告诉你，还可以帮助你预估一下更改基类里的虚函数签名可能会引起的后果，如果在派生类中都使用了 override ，你可以改一下基类中虚函数的签名，重编一下看看这个举动会造成多少问题（即，有多少派生类无法通过编译），然后决定这个改动是否值得。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，派生类中那些原本应该覆盖基类函数的部分，不会引发编译器的诊断信息。

C++已经有了很多关键字，但 C++又引入了两个上下文关键字（contextual keyword）：override 和 final。这两个关键字只有在特殊语境下才有保留特性。比如 override 只有出现再成员函数声明的结尾时才有，也就是说，如果你以前的代码已经用了 override 这个名字，切换到 C++11 是不用修改的：

```
class Warning { // C++98的类
public:
    ...
    void override(); // C++98 和 C++11 都合法，并且意思一样，都是函数名
    ...
};
```

到这里，override 已经讲完了，但成员函数引用后置修饰符还没讲完，前面我说了后面会来将，那么现在开始讲解。

如果我们想要写一个只接受左值实参的函数，我们会声明一个 non-const 左值引用形参：

```
void doSomething(Widget& w); // 只接受左值（lvalue） Widgets
```

如果我们想要写一个只接受右值实参的函数，我们会声明一个右值引用形参：

```
void doSomething(Widget&& w); // 只接受右值（rvalue） Widgets
```

成员函数引用后置修饰符*this 对象的成员函数调用达到同样的区分左右值效果。这个和成员函数后面的 const 声明类似，const 是表明调用这个函数的*this 对象必须是 const。

成员函数的引用后置修饰符一般不会用到，但有时能提高效率。举个例子，假设我们的 Widget 类有个 std::vector 数据成员，并且提供了一个访问函数直接返回这个成员：

```
class Widget {
    public:
```

```
    using DataType = std::vector<double>; // see Item 9 for
    …                                     // info on "using"
    DataType& data() { return values; }
    …
private:
    DataType values;
};
```

这个是以前经常用到的封装设计，但考虑下面的用户代码：

```
Widget w;
…
auto vals1 = w.data(); // copy w.values into vals1
```

很明显，Widget::data 的返回值类型是一个左值引用（std::vector<double>&），因为左值引用被定义为左值，所以我们用一个左值来初始化 vals1，因此，就像注释里说明的，vals1 从 w.values 拷贝构造。

现在，假设我们有一个创建 Widget 的工厂函数，

```
Widget makeWidget();
```

并且我们想要用 makeWidget 返回的 Widget 中的 std::vector 数据成员来初始化一个变量：

```
auto vals2 = makeWidget().data(); // copy values inside the Widget into
vals2
```

跟前面一样，vals2 也是从 Widget 的数据成员拷贝构造，但是这次的 Widget 是一个由 makeWidget 返回的临时对象，即右值，所以拷贝它的 std::vector 数据成员是浪费时间，最好方式是移动，但是因为 data 返回的是一个左值引用，按 C++的规则，编译器生成的代码是拷贝操作。（有一些怪招可以绕过所谓的规则进行优化，但你最好还是依赖编译器找出一个更优的方式）

要找的方式就是指定好，当用右值 Widget 调用 data 时，返回的也是一个右值。使用引用后置修饰符来重载左值和右值 data 函数，恰好能满足要求：

```
class Widget {
public:
    using DataType = std::vector<double>;
    …
    DataType& data() &              // for lvalue Widgets,
    { return values; }              // return lvalue
    DataType data() &&              // for rvalue Widgets,
    { return std::move(values); }   // return rvalue
    …
```

```
private:
    DataType values;
};
```

注意，data 重载的返回类型是不一样的。左值引用重载返回一个左值引用（即一个左值），而右值引用重载返回的是一个临时变量（即一个右值），这样的话，上面的代码行为就是我们想要的了：

```
auto vals1 = w.data();              // calls lvalue overload for
                                    // Widget::data, copy-constructs vals1
auto vals2 = makeWidget().data(); // calls rvalue overload for
                                    // Widget::data, move-constructs vals2
```

这个肯定爽歪歪，但别高兴的忘了本条款的重点，那就是，任何时候，在派生类要声明一个覆盖基类虚函数的函数时，一定要用 override 关键字。

需要铭记的要点
● 用关键字 override 声明覆盖函数。 ● 成员函数引用后置修饰符可以区别对待左值和右值对象。

条款 13：优先使用 const_iterators 而非 iterators

const_iterator 在 STL 中等价于指向 const 的指针，指向的数值是不能修改的。根据能用 const 尽量用 const 的原则，在你需要用迭代器的时候，如果不需要修改迭代器指向的内容时，你应该使用 const_iterator。

像 C++11 一样，这个说法对 C++98 来说也是正确的，但是在 C++98 中， const_iterator 只能支持部分功能。创建它们并非易事，一旦有一个这样的迭代器，使用方式也会受限。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++"代替"C with class"作为语言名称)，然后在搜到的位置插入数值 1998(这一年第一个 ISO C++标准被采用)，如果 vector 不存在 1983，就在 vector 的末尾插入。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。将代码改为使用 const_iterator 本该是一个微小的改动，但在 C++98 中，却不是那么回事。下面这个改动方式，概念上感觉是可以的，但实际上是不正确的：

```
typedef std::vector<int>::iterator IterT;              // typedefs
typedef std::vector<int>::const_iterator ConstIterT;   //

std::vector<int> values;
…
ConstIterT ci =
    std::find(static_cast<ConstIterT>(values.begin()),     // cast
              static_cast<ConstIterT>(values.end()),       // cast
              1983);
values.insert(static_cast<IterT>(ci), 1998); // 可能无法编译，看下面解释
```

当然，typedef 不是必须的，但这会使得后面的 cast 代码更加容易编写。（如果你想知道为什么使用 typedef 而不是使用条款 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，别名声明是 C++11 的新特性。）

在 std::find 的调用中用到了强制类型转换，是因为 values 是 non-const 的容器，但在 C++98 中并没有简单的办法可以从一个 non-const 容器中得到一个 const_iterator 。强制类型转换并非必要的，因为可以用其他办法得到 const_iterator （比如，可以绑定 values 到

一个 const 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 non-const 容器中得到一个 const_iterator 的过程比较折腾。

一旦使用了 const_iterator ，事情变得更糟，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 中小心翼翼拿到的）又转换成了 iterator ，因为给 insert 传一个 const_iterator 会编译失败。

老实说，我上面展示的代码可能就编译不通过，这是因为并没有从 const_iterator 到 interator 的便捷转换，即使用 static_cast 也不行，甚至最暴力的 reinterpret_cast 也不成。（这不是 C++98 的限制，C++11 也同样如此。 const_iterator 不能简单地转换成 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，不值得花心思。到最后，在任何可能的地方，开发者们都不会使用 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 可以生成 const_iterator ，甚至 non-const 容器也可以，而且用迭代器定位的 STL 成员函数（比如，插入和删除）实际上都是用的 const_iterator 。对于 C++11，修改前面的 C++98 代码，使用 const_iterator 替换 iterator 的改动就微不足道了：

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

现在这个 const_iterator 就非常实用了！

只有一种情况，C++11 对 const_iterator 的支持是有点不足的，就是在编写最大化泛型库代码的时候。这样代码需要考虑，有些容器或者类似于容器的数据结构，提供的是非成员函数 begin 和 end （还有 cbegin, cend,rbegin 等等）。如，内置的数组和一些只有自由函数接口的第三方库。因此，最大化泛型代码只能使用非成员函数，而不是假设存在那样的成员函数。

例如，我们可以将刚才的代码泛化到一个 findAndInsert 模板中，如下：

```
template<typename C, typename V>
void findAndInsert(C& container,           // in container, find
```

```
                    const V& targetVal,     // first occurrence
                    const V& insertVal)     // of targetVal, then
{                                           // insert insertVal
    using std::cbegin;                      // there
    using std::cend;
    auto it = std::find(cbegin(container),     // non-member cbegin
                        cend(container),       // non-member cend
                        targetVal);
    container.insert(it, insertVal);
}
```

这在 C++ 14 中工作得很好，但遗憾的是，在 C++ 11 中不行。由于标准化的疏忽，C++ 11 增加了非成员数的 begin 和 end，但它未能添加 cbegin、rbegin、rend、crbegin 和 crend。C++ 14 弥补了这些。

如果你正在使用的是 C++ 11，你希望编写最大泛型代码，而你正在使用的库都没有提供非成员函数 cbegin 和其他小伙伴的模板，你可以轻松地实现那些函数模板。例如，下面是一个非成员函数 cbegin 的实现：

```
template <class C>
auto cbegin(const C& container)->decltype(std::begin(container))
{
    return std::begin(container); // see explanation below
}
```

你会惊讶地发现非成员函数 cbegin 没有调用成员函数 cbegin，难道你不惊讶吗？我也惊讶，但这个是合乎逻辑的。这个 cbegin 模板接受一个参数，可以是任何一种类似于容器的数据结构类型（C），通过 const 引用（container）访问这个参数。如果 C 是一个常规容器类型(如 std::vector)，container 将是该容器的 const 引用(如，const std::vector<int>&)。在 const 容器上调用非成员函数 begin(由 C++ 11 提供)生成一个 const_iterator，也就是这个模板的返回类型。这样实现的优势是，即使对那些提供了 begin 成员函数（对容器来说，C++11 的非成员函数 begin 调用是这个），却没有提供 cbegin 成员函数的容器也适用，因此，你可以在只支持 begin 成员函数的容器上使用这个非成员函数 cbegin。

如果 C 是内置数组类型，该模板也可以工作。在这种情况下，container 变成对 const 数组的引用。C++ 11 为数据提供了一个专门的非成员函数 begin，返回指向数组第一个元素的指针。const 数组的元素也是 const，因此非成员函数 begin 返回一个指向 const 的指针，而这个指针实际上就是数据的 const_iterator。（要了解模板怎样才能对内置数组进行特化，请参阅条款 1 中关于模板接受数组引用参数时的类型推导）

言归正传，这个条款的关键点是鼓励你只要能用 const_iterator 就用 const_iterator。这条基本原则——只要有意义，就使用 const——在 C++ 11 之前就存在，但是在 C++ 98 中，它对迭代器来说不实用，在 C++ 11 中，就非常实用，C++ 14 还为 C++ 11 一些遗留工作收了尾。

需要铭记的要点
● 优先使用 const_iterator 而不是 iterator。 ● 在最大泛型代码中，相对于它们的成员函数部分，优先使用 begin、end、rbegin 等的非成员函数版本。

条款 14：将不会抛出异常的函数声明为 noexcept

在 C++ 98 中，异常规范是相当难以捉摸的。你往往必须对函数可能发出的所有异常类型进行概述，因此，如果修改了函数的实现，异常规范也可能需要修改。如果更改异常规范，可能会破坏用户代码，因为调用程序可能依赖于原始的异常规范。编译器通常没有帮助你去维护函数实现、异常规范和用户代码之间的一致性。大多数程序员最终认为 C++ 98 异常规范不值得花心思研究使用。

在 C++ 11 的开发过程中，大家普遍认为，真正有意义的信息是，函数是否会触发异常。非黑即白，函数要么可能发出异常，要么保证不会发出异常。这种可能是或者永远不会（maybe-or-never）的二分法构成了 C++ 11 异常规范的基础，从而本质上替代了 C++ 98 的异常规范。(C++ 98 风格的异常规范仍然有效，但不推荐使用。)在 C++ 11 中，绝对的 noexcept 用于修饰保证不会发出异常的函数。

一个函数是否应该这样声明，是一个接口设计要考虑的问题。函数的异常触发行为对用户来说非常关键。调用者可以查询函数的 noexcept 状态，并且查询结果会影响调用代码的异常安全性或效率。因此，函数是否为 noexcept 与成员函数是否为常量同样重要。当你知道一个函数不会触发异常，却没有声明为 noexcept，那就是糟糕的接口设计。

但是，将函数声明为 noexcept 的另一个原始是：它允许编译器生成更好的目标代码。研究一下 C++ 98 和 C++ 11 中，对于不会触发异常的函数的不同表示方法，可以帮助你理解其中的原因。考虑一个函数 f，它承诺调用者永远不会收到异常。这两种表达方式是：

```
int f(int x) throw(); // no exceptions from f: C++98 style
int f(int x) noexcept; // no exceptions from f: C++11 style
```

如果在运行时 f 出现了异常，那么就违反 f 的异常规范。根据 C++ 98 异常规范，调用堆栈解除到 f 的调用者，并且，如果没有做相关的异常处理，程序执行也就终止了。在 C++ 11 异常规范中，运行时行为略有不同：只有在程序执行终止之前才可能解除堆栈。

解除调用堆栈与可能解除调用堆栈之间的差异，对代码生成的影响大得惊人。在 noexcept 函数中，如果有异常从函数中传递出去，优化器不需要将运行时堆栈保持在可接触的状态，也不需要确保在异常离开函数时，按照构造的逆顺序销毁 noexcept 函数中的对象。带有“throw()”异常规范的函数缺乏这种优化灵活性，完全就跟没有“throw()”的函数一样。情况可以这样总结：

```
RetType function(params) noexcept;    // most optimizable
RetType function(params) throw();     // less optimizable
RetType function(params);             // less optimizable
```

这一点就有充足的理由来要求你，在任何时候，只要知道函数不会产生异常，就声明为 noexcept。

对于某些函数，更需要这样做。移动操作就是一个突出的例子。假设你有一个使用 std::vector<Widget>的 C++ 98 代码库。Widget 通过 push_back 不时地添加到 std::vector：

```
std::vector<Widget> vw;
…
Widget w;
… // work with w
vw.push_back(w); // add w to vw
…
```

假设这段代码工作正常，并且你没有兴趣为 C++ 11 修改它。然而，你确实希望利用 C++ 11 的 move 语义，可以在涉及到可移动（move-enabled）类型时，提高遗留代码的性能。因此，你要确保 Widget 支持 move，要么通过自己编写操作，要么确保使用自动生成的操作(参考条款 17)。

当一个新元素添加到 std::vector 时，有可能 std::vector 没有空间了来存放了，即 std::vector 的大小等于容量了。这时，std::vector 就申请一个更大的新内存块，并把元素从旧的内存转移过来。C++98 实现方式时，将每个元素拷贝到新的内存，然后再释放旧内存的对象。这种方式使 push_back 提供了强异常安全保障：如果在拷贝过程中出现异常的话，std::vector 原来的状态是不变的，因为只有所有对象都拷贝到新内存中，原来内存中的对象才会销毁。

C++11 中，一种很自然的优化就是用 move 代替拷贝 std::vector 元素。不幸的是，这样

做的话，push_back 有异常安全保证方面的风险。如果有 n 个元素已经移动完，而在移动第 n+1 个元素发生异常，push_back 操作就无法执行完成，但是原来的 std::vector 已经被修改了：即，n 个元素已经移动完了。要还原回去也许不大可能，因为在还原的过程中，也可能会产生异常。

这是个严重的问题，因为遗留代码的行为依赖于 push_back 的强异常安全保障。因此，C++11 的实现不能偷偷摸摸地将拷贝替换成 move，除非它能知道 move 不会产生异常。那样的话，move 替换拷贝就是安全的，并且可以带来性能提升。

std::vector::push_back 运用了这种“可以的情况下都用 move，必须时才用拷贝”的策略，而且它不是标准库中唯一这样做的函数。其他在 C++ 98 中具有强异常安全保障的函数(例如，std::vector::reserve, std::deque::insert，等等)也有相同的行为。所有这些函数都将由 C++ 98 中调用拷贝操作替换为 C++ 11 中的调用移动操作，前提是移动操作不触发异常。但是函数如何知道移动操作是不会产生异常的呢?答案很明显：它检查操作是否声明为 noexcept。

swap 函数包含另一种情况，在这种情况下，noexcept 是特别可取的。swap 是许多 STL 算法实现的关键组件，它通常也用于拷贝赋值操作。它的广泛使用使 noexcept 优化变得特别有价值。有趣的是，STL 的 swap 是否是 noexcept 是取决于用户定义的 swap 是否是 noexcept。例如，STL 对数组和 std::pair 的 swap 声明如下:

```
template <class T, size_t N>
void swap(T (&a)[N], // see
          T (&b)[N]) noexcept(noexcept(swap(*a, *b))); // below

template <class T1, class T2>
struct pair {
    ...
    void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                                noexcept(swap(second, p.second)));
    ...
};
```

这些函数是有条件的 noexcept：它们是否 noexcept 取决于 noexcept 子句是否 noexcept。例如，给定两个 Widget 数组，只有当数组中的单个元素交换是 noexcept 时，即，Widget 的 swap 是 noexcept 时，交换它们才是 noexcept。因此，Widget 的 swap 函数作者决定了交换 Widget 的数组是否为 noexcept。反过来，这又决定了其他的 swap 操作，如 Widget 数组的数组的 swap，是否是 noexcept。类似地，是否交换两个包含 Widget 的 std::pair 对象是否是 noexcept，取决于 Widget 的 swap 是否是 noexcept。事实上，通常只有在 swap 它的低层级

（lower-level）组成部分是 noexcept 时，swap 高层级（higher-level）数据结构才是 noexcept，这就促使你在任何可能的情况下，将 swap 函数写为 noexcept。

到目前为止，我希望你能对 noexcept 提供的优化机会感到兴奋。唉，我得给你泼泼冷水。优化很重要，但正确性更重要。在本条款的开头，我备注到 noexcept 是函数接口的一部分，所以只有当你愿意长期提交 noexcept 实现时，才应该声明函数为 noexcept。如果你声明了一个函数 noexcept，然后又后悔了，那么你的选择将是凄惨的。你可以从函数的声明中删除 noexcept(即，改变其接口)，因此也就有了破坏用户代码的风险。你可以更改实现，以便异常可以传出，同时保留原来的(现在是错误的)异常规范。如果这样做，当异常从函数传出时，程序将被终止掉。或者你对现有的实现认命，放弃最初更改实现的任何想法。这两个选择都不令人满意。

事实上，大多数函数都是异常无关的（exception-neutral）。这些函数本身不会抛出异常，但是它们调用的函数可能会发出一个异常，此时，异常无关的函数异常通过它向上传递。异常无关的函数不是 noexcept，因为它们可能发出“路过的”异常。因此，大多数函数都缺少 noexcept 指定。

然而，有些函数具有不发出异常的自然实现，对于少数函数(尤其是 move 和 swap)，noexcept 可以带来很大的价值（如性能提升——译者注），应该尽最大的可能，以 noexcept 方式实现。当你可以拍胸脯保证一个函数应该永远不发出异常时，你应该明确地声明它为 noexcept。

请注意，我说过一些函数有自然的 noexcept 实现。扭曲一个函数的实现以便允许一个 noexcept 声明，就像是摇尾乞怜，是本末倒置，是只见树木不见森林。如果一个函数实现明显可能产生异常(例如,调用函数可能抛出)，你有方法可以隐藏这些异常(例如,捕捉所有的异常,代之以状态码或特殊的返回值)，但这样不仅会使你的函数实现复杂化，通常也会使调用侧的代码复杂化。例如，调用者可能必须检查状态码或特殊返回值，这些复杂性带来的运行时成本(如，额外的分支、对指令缓存带来更大压力的超大函数等)，可能超过你希望通过 noexcept 实现的速度提升，而且你还将承担更难以理解和维护的源代码。那将是糟糕的软件工程。

对于某些函数来说，noexcept 非常重要，默认情况下它们就是这样实现。在 C++ 98 中，允许内存释放函数(即 delete 和 delete[])以及析构函数发出异常是很糟糕的风格，在 C++ 11 中，这种做法几乎已经升级为语言规则。默认情况下，所有内存释放函数和所有析构函数(包括用户定义的和编译器生成的)都是隐式 noexcept。因此，没有必要声明它们为 noexcept。

(声明也没什么影响，只是不合常规。)析构函数唯一不是隐式 noexcept 的情况是，类的数据成员(包括继承的成员和包含在其他数据成员中的成员)的类型明确声明其析构函数可能发出异常(例如，声明它为“noexcept(false)”)，这种析构函数并不常见。STL 没有这样的情况，并且如果 STL 引用的对象（如在容器中或传给算法函数）析构抛出异常，程序的行为将是不确定的。

值得注意的是，一些库接口设计人员将宽契约函数与窄契约函数区分开来。宽契约函数没有先决条件，这种函数调用是状态无关的，并且它不会对调用者传递的参数施加任何约束。宽契约函数永远不会表现出不确定的行为。

非宽契约的函数就是窄契约的，对于这些函数，如果违反了先决条件，则结果是不确定的。

如果你正在编写一个宽契约的函数，并且你知道它不会发出异常，那么遵循本条款的建议并声明它为 noexcept 是很容易的。对于窄契约函数来说，情况要复杂得多。例如，假设你编写的函数 f 使用 std::string 参数，并且假设 f 的自然实现永远不会产生异常，那么建议 f 应声明为 noexcept。

现在假设 f 有一个先决条件：它的 std::string 参数的长度不超过 32 个字符。如果用长度大于 32 的 std::string 调用 f，则行为将是不确定的，因为根据定义，违反先决条件会导致不确定的行为。f 没有义务检查这个前提条件，因为函数可以假定它们的前提条件是满足的。(调用者有责任确保这些假设是有效的。)即使有一个先决条件，声明 f noexcept 似乎也是合适的：

```
void f(const std::string& s) noexcept; // precondition: s.length() <= 32
```

但是假设 f 的实现者选择检查是否有违反先决条件的情况。虽然没必要检查，但也不禁止检查，而且检查先决条件可能很有用，如在系统测试期间。调试抛出的异常通常比跟踪不确定行为的原因更容易。但是，前提条件违反了，应该如何报告，以便测试工具或客户端错误处理程序能够检测到它呢？直接的方法是抛出“前提被违反”异常，但如果 f 被声明为 noexcept，那就不可能了，抛出异常将导致程序终止。因此，区分宽契约和窄契约的库设计人员通常只为宽契约函数保留 noexcept 声明。

最后一点，让我详细说明我之前的说法，即编译器通常无法帮助识别函数实现及其异常规范之间的不一致性。考虑以下这个完全合法的代码：

```
void setup(); // functions defined elsewhere
void cleanup();
void doWork() noexcept
```

```
{
    setup();     // set up work to be done
    …            // do the actual work
    cleanup();   // perform cleanup actions
}
```

在这里，doWork 被声明为 noexcept，尽管它调用 non-noexcept 函数 setup 和 cleanup。这看起来很矛盾，但是可能是 setup 和 cleanup 的文档说明它俩永远不会发出异常。他们的 non-noexcept 声明可能有很好的理由。例如，它们可能是用 C 编写的库的一部分，（甚至是从 C 标准库移到 std 命名空间的函数缺少异常规范，如，std::strlen 没有　声明 noexcept。）或者可能是 C++98 库中未使用 C++98 异常规范，但还没被 C++11 改进的部分。

由于有正当理由让 noexcept 函数依赖于没有 noexcept 保证的代码，所以 C++允许这样的代码，而且编译器通常不会发出关于它的警告。

需要铭记的要点
● noexcept 是函数接口的一部分，也就意味着，调用者对它有依赖 ● noexcept 函数比 non-except 函数更可以优化 ● noexcept 对 move 操作、swap、内存释放函数、析构函数特别有价值 ● 大多数函数是异常无关的（exception-neutral），而不是 noexcept

条款 15：尽可能的使用 constexpr

如果要评选 C++ 11 中最易混淆的新词的话，那么非 constexpr 莫属。当用于对象时，它本质上是 const 的一种增强形式，但是当用于函数时，它就是完全不同的含义。花精力理解这个，肯定是值得的，因为当 constexpr 与你想要表达的内容相对应时，你肯定想要使用它。

从概念上讲，constexpr 表示的是一个编译期间已知的常量值。然而，这个概念只表达了一部分，因为当 constexpr 应用于函数时，事情比这要微妙得多。为了不泄露天机，现在我只想说，你不能假定 constexpr 函数的结果是 const，也不能想当然地认为它们的值在编译期间是已知的。也许最有趣的是，这些还都是特色，constexpr 函数不生成 const 或编译期间已知的结果，还是好事呢。

让我们先从 constexpr 对象开始讲解。实际上，如果对象是 const，那么它们的值在编译时就一定是已知的。(从技术上讲，它们的值是在翻译过程中确定的，而翻译不仅包括编译，还包括链接。但是，除非你要为 C++编写编译器或链接器，否则这对你没有任何影响，所以你可以轻松地编写程序，就像编译期间确定了 constexpr 对象的值一样。)

编译期间已知的值是特殊照顾的。例如，它们可能被放置在只读内存中，特别是对于嵌入式系统的开发人员，这可能是一个非常重要的特性。更广泛地讲，编译期间已知的整数常量可以用于 C++中需要整数常量表达式的上下文，这些上下文包括指定数组大小、整型模板参数(包括 std::array 对象的长度)、枚举值、对齐描述符等等。如果你想为这类事情使用一个变量，你当然希望声明它为 constexpr，因为这样编译器将确保它具有编译时值（compile-time value）：

```
int sz;                          // non-constexpr 变量
…
constexpr auto arraySize1 = sz;  // error! sz的值编译期间未知
std::array<int, sz> data1;       // error! 问题同上
constexpr auto arraySize2 = 10;  // fine, 10 is a compile-time constant
std::array<int, arraySize2> data2; // fine, arraySize2 is constexpr
```

注意，const 不提供与 constexpr 相同的保证，因为 const 对象不需要使用编译期间已知值进行初始化：

```
int sz;                          // as before
…
const auto arraySize = sz;       // fine, arraySize is const copy of sz
std::array<int, arraySize> data; // error! arraySize的值编译期间未知
```

简单地说，所有 constexpr 对象都是 const，但不是所有 const 对象都是 constexpr。如果

希望编译器保证变量的值可以在需要编译时常量的上下文中使用，那么需要使用的工具是constexpr，而不是 const。

当涉及到 constexpr 函数时，constexpr 对象的使用场景变得更加有趣。当使用编译时常量调用这些函数时，它们将生成编译时常量。如果用运行时才知道的值调用它们，则会生成运行时值。这听起来好像你不知道它们会怎么做，但这种想法是错误的，正确的看法是：

- 可以在需要编译时常量的上下文中使用 constexpr 函数。如果在这样的上下文中，传递给 constexpr 函数的实参在编译期间已知，那么结果在编译期间就计算好；如果任何一个实参是编译期间未知的，那么代码将被编译不通过。
- 当使用编译期间未知的一个或多个值调用 constexpr 函数时，它就像一个普通函数一样，在运行时计算其结果。这意味着你不需要两个函数来执行相同的操作，一个用于编译时常量，另一个用于其他所有值，constexpr 函数一手搞定。

假设我们需要一个数据结构来保存可以以多种方式运行的实验结果。例如，在实验过程中，照明等级可以是高、低、关闭，风扇转速和温度也有类似情况，等等。如果有 n 个环境条件与实验相关，每个条件有三个状态，那么它们的组合个数就是 3n。因此，存储所有条件组合的实验结果需要一个具有足够空间容纳 3n 个值的数据结构。假设每个结果是一个 int，并且在编译期间 n 是已知的(或可以计算的)，那么 std::array 可能是一个合理的数据结构选择。但是我们需要一个方法在编译期间计算 3n。C++标准库提供了我们所需要的数学功能函数 std::pow，但是，对于我们的目的，它有两个问题。首先，std::pow 处理的是浮点类型，而我们需要一个整型结果；其次，std::pow 不是 constexpr，因此，我们不能使用它来指定 std::array 的大小。

幸运的是，我们可以写出我们需要的 pow。稍后我将展示如何实现这一点，先让我们看看如何声明和使用：

```
constexpr                       // pow是一个永远不会抛出异常的constexpr函数
int pow(int base, int exp) noexcept
{
    ...                                     // impl is below
}

constexpr auto numConds = 5;
std::array<int, pow(3, numConds)> results; // results具有3^numConds个元素
```

回想一下，pow 前面的 constexpr 并没有说 pow 返回一个 const 值，而是说如果 base 和 exp 是编译时常量，那么 pow 的结果可以用作编译时常量。如果 base 和 exp 有一个不是编

译时常量，pow 的结果将在运行时计算。这意味着，pow 不仅可以用来在编译时计算 std::array 大小，还可以在运行时调用，比如：

```
auto base = readFromDB("base");        // get these values
auto exp = readFromDB("exponent");     // at runtime
auto baseToExp = pow(base, exp);       // call pow function at runtime
```

因为当使用编译时值调用 constexpr 函数时，函数必须能够返回编译时结果，所以对它们的实现作了限制。C++ 11 和 C++ 14 的限制是不同的。

在 C++ 11 中，constexpr 函数只能包含一条 return 语句。但可以使用两种技巧将 constexpr 函数的表达式扩展到超出你想象的范围。首先，有条件的“?:"运算符可以用来代替 if-else 语句，其次，递归可以用来代替循环。因此，pow 可以这样实现：

```
constexpr int pow(int base, int exp) noexcept
{
    return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

这是可行的，但是很难想象除了铁杆的函数式程序员之外，还有谁会认为它很漂亮。在 C++ 14 中，对 constexpr 函数的限制要宽松得多，因此可以实现以下功能：

```
constexpr int pow(int base, int exp) noexcept // C++14
{
    auto result = 1;
    for (int i = 0; i < exp; ++i) result *= base;
    return result;
}
```

constexpr 函数仅限于接受和返回字面类型（literal type），实际上就是可以在编译期间确定值的类型。在 C++ 11 中，除了 void 以外的所有内置类型都是合格的，但是用户定义的类型也可能是字面类型，因为构造函数和其他成员函数可能都是 constexpr：

```
class Point {
public:
    constexpr Point(double xVal = 0, double yVal = 0) noexcept
    : x(xVal), y(yVal)
    {}
    constexpr double xValue() const noexcept { return x; }
    constexpr double yValue() const noexcept { return y; }
    void setX(double newX) noexcept { x = newX; }
    void setY(double newY) noexcept { y = newY; }
private:
    double x, y;
};
```

在这里，Point 构造函数可以声明为 constexpr，因为如果传递的参数的编译时已知的，那么数据成员就是编译时已知的，Point 也就可以初始化成 constexpr：

```
constexpr Point p1(9.4, 27.7);    // fine, 编译期间"运行" constexpr构造函数
constexpr Point p2(28.8, 5.3);    // also fine
```

类似地， xValue 和 yValue 的 getter 也可以是 constexpr，因为如果在编译期间调用具有已知值的 Point 对象(例如 constexpr)时调用它们，可以在编译过程中知道数据成员 x 和 y 的值。这样就可以编写调用 Point 的 getter 的 constexpr 函数，并使用这些函数的结果初始化 constexpr 对象：

```
constexpr
Point midpoint(const Point& p1, const Point& p2) noexcept
{
    return { (p1.xValue() + p2.xValue()) / 2,      // call constexpr
             (p1.yValue() + p2.yValue()) / 2 };    // member funcs
}

constexpr auto mid = midpoint(p1, p2);     // init constexpr
                                           // object w/result of
                                           // constexpr function
```

这太爽了！这意味着，尽管 mid 对象的初始化涉及到对构造函数、getter 和非成员函数的调用，但它可以创建在只读内存中！这样就可以在模板的参数或指定枚举值的表达式中使用 mid.xValue() * 10 这样的表达式。这意味着，在编译期间完成的工作和在运行时完成的工作之间，传统上相当严格的界限开始模糊，一些传统上在运行时完成的计算可以迁移到编译时。参与迁移的代码越多，软件运行的速度就越快。(不过，编译可能需要更长的时间。)

在 C++ 11 中，有两个限制阻止了 Point 的成员函数 setX 和 setY 声明为 constexpr。首先，它们要修改所操作的对象，而 C++ 11 的 constexpr 成员函数都是隐式 const；其次，它们有 void 返回类型，而 void 在 C++ 11 中不是字面类型。这两个限制在 C++ 14 中都被解除了，所以在 C++ 14 中，甚至 Point 的 setter 都可以是 constexpr：

```
class Point {
public:
    …
    constexpr void setX(double newX) noexcept // C++14
    { x = newX; }
    constexpr void setY(double newY) noexcept // C++14
    { y = newY; }
    …

};
```

这样就可以写如下函数：

```
// return reflection of p with respect to the origin (C++14)
constexpr Point reflection(const Point& p) noexcept
{
    Point result; // create non-const Point
    result.setX(-p.xValue()); // set its x and y values
    result.setY(-p.yValue());
    return result; // return copy of it
}
```

用户代码可以这样写：

```
constexpr Point p1(9.4, 27.7);    // as above
constexpr Point p2(28.8, 5.3);
constexpr auto mid = midpoint(p1, p2);
constexpr auto reflectedMid =     // reflectedMid's value is(19.1, 16.5)
    reflection(mid);              // and known during compilation
```

本条款的建议是尽可能使用 constexpr，到此为止，我希望你已经整明白了，与非 constexpr 对象和函数相比，constexpr 对象和函数都可以在更广泛的上下文中使用。只要可能，通过使用 constexpr，可以最大限度地扩大对象和函数的使用情境。

需要注意的是，constexpr 是对象或函数接口的一部分，constexpr 表明“我可以在 C++需要常量表达式的上下文中使用”，如果你声明一个对象或函数为 constexpr，用户就可以用在那样的上下文中。如果你后来认为使用 constexpr 是错误的，并将其删除，那么可能会导致大量用户代码无法编译。(往函数中添加 I/O 用来调试或性能调优，可能会导致这样的问题，因为在 constexpr 函数中通常不允许 I/O 语句)。在“尽可能使用 constexpr”中，“尽可能”的含义是你愿意承诺对 constexpr 对象和函数的 constexpr 长期不变。

需要铭记的要点
● constexpr 对象是 const 的，并且是用编译时已知值进行初始化。 ● 当用编译时已知值的实参调用 constexpr 函数时，生成的是编译时结果。 ● constexpr 对象和函数比 non-constexpr 函数的应用上下文更广泛。 ● constexpr 是对象和函数接口的一部分。

条款 16：确保 const 成员函数线程安全

如果我们在数学领域工作，我们可能会发现如果有一个表示多项式的类会很方便。在这个类中，有一个计算多项式的根（即多项式取值为 0 时的值）的函数可能会有用。这样的函数不会修改多项式，所以很自然地声明它为 const：

```
class Polynomial {
public:
    using RootsType =         // data structure holding values
        std::vector<double>; // where polynomial evals to zero
    …                         // (see Item 9 for info on "using")
    RootsType roots() const;
    …

};
```

计算多项式的根是很费时的，所以如果不是必要的话，我们就不想做。如果我们必须这样做，我们肯定不想做多于一次。因此，如果需要计算，我们将缓存多项式的根，而我们实现的 roots 函数就是返回缓存的值。下面是一个基本实现：

```
class Polynomial {
public:
    using RootsType = std::vector<double>;
    RootsType roots() const
    {
        if (!rootsAreValid) {     // if cache not valid
            …                     // compute roots,
                                  // store them in rootVals
            rootsAreValid = true;
        }
        return rootVals;
    }

private:
    mutable bool rootsAreValid{ false };  // see Item 7 for info
    mutable RootsType rootVals{};         // on initializers
};
```

从概念上讲，roots 函数不会改变它所操作的多项式对象，但是作为其缓存活动的一部分，它可能需要修改 rootVals 和 rootsAreValid。这是使用 mutable 关键字的一个经典用例，这就是为什么数据成员声明包含 mutable。

现在假设两个线程同时调用多项式对象的根：

```
Polynomial p;
…
/*----- Thread 1 ----- */  /*------- Thread 2 ------- */
auto rootsOfP = p.roots(); auto valsGivingZero = p.roots();
```

这个用户代码非常合理。roots 是一个 const 成员函数，这意味着它表示一个读操作。让多个线程在没有同步的情况下执行读操作是安全的。至少理论上应该是这样的。但在本例中，却不是这样的，因为在 roots 中，这些线程中的一个或两个可能试图修改数据成员 rootsAreValid 和 rootVals。这意味着，此处有不同的现成读写同一块内存却没有同步处理，这就是数据竞争的定义。这段代码存在不确定行为。

问题就是 roots 声明为 const，但它不是线程安全的。const 声明在 C++11 中与在 C++98 中一样正确(获取多项式的根不会改变多项式的值)，因此需要纠正的是缺乏线程安全性。

解决这个问题最简单的办法就是使用互斥量：

```
class Polynomial {
public:
    using RootsType = std::vector<double>;
    RootsType roots() const
    {
        std::lock_guard<std::mutex> g(m); // lock mutex
        if (!rootsAreValid) {             // if cache not valid
            …                             // compute/store roots
            rootsAreValid = true;
        }
        return rootVals;
    } // unlock mutex
private:
    mutable std::mutex m;
    mutable bool rootsAreValid{ false };
    mutable RootsType rootVals{};
};
```

std::mutex m 声明为 mutable，因为锁定和解锁它都是非 const 成员函数，而在 roots(const 成员函数)中，m 被视为 const 对象。

值得注意的是，因为 std::mutex 是一种只能移动的（move-only）类型(即，只能移动不能拷贝)，在多项式中加入 m 的一个副作用就是这个多项式不能拷贝，但仍然可以移动。

在某些情况下，互斥量是没有必要用的。例如，如果你所做的只是统计一个成员函数的调用次数，用 std::atomic 计数器(参考条款 40)通常是占用资源更好的方式。（是否占用资源少，这个也依赖于实际运行的硬件和 STL 中 mutex 的实现）此处可以用 std::atomic 来统计

调用次数。

```
class Point { // 2D point
public:
    …
    double distanceFromOrigin() const noexcept     // see Item 14
    {                                              // for noexcept
        ++callCount;                               // atomic increment
        return std::sqrt((x * x) + (y * y));
    }
private:
    mutable std::atomic<unsigned> callCount{ 0 };
    double x, y;
};
```

跟 std::mutex 一样，std::atomic 也是 move-only 类型，所以因为 callCount 成员变量，Point 也是 move-only。

因为对 std:: atomic 变量的操作通常比 mutex 的请求和释放更节省资源，你可能会过度依赖 std::atomic。例如，在缓存一个计算很耗资源的 int 的类中，你可能会用一对 std::atomic 变量，而不是互斥：

```
class Widget {
public:
    …
    int magicValue() const
    {
        if (cacheValid) return cachedValue;
        else {
            auto val1 = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cachedValue = val1 + val2;    // uh oh, part 1
            cacheValid = true;            // uh oh, part 2
            return cachedValue;
        }
    }
private:
    mutable std::atomic<bool> cacheValid{ false };
    mutable std::atomic<int> cachedValue;
};
```

代码可以工作，但有时难以像设想的那样工作，考虑如下情况：

- 一个线程调用 Widget::magicValue，看到 cacheValid 是 false，就执行了那两个耗资源的计算，然后将结果赋值给 cachedValue。

- 碰巧在那个时间点，第二个线程调用 Widget::magicValue，也看到 cacheValid 是 false，因此也执行了两个计算。（这所谓的“第二个线程”也可能是很多个类型情况的线程）

这种行为与缓存的设计目标是相悖的。颠倒 cachedValue 和 CacheValid 的赋值顺序，可以消除这个问题，但结果更糟糕：

```
class Widget {
public:
    ...
    int magicValue() const
    {
        if (cacheValid) return cachedValue;
        else {
            auto val1 = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cacheValid = true;                  // uh oh, part 1
            return cachedValue = val1 + val2;   // uh oh, part 2
        }
    }
    ...
};
```

想象一下，cachedValue 是 false，那么：

- 一个线程调用了 Widget::magicValue，并且执行到将 cachedValue 设置为 true；
- 在那个时间点，第二个线程调用 Widget::magicValue，并且检查 cache 是否有效。刚好发现是 true，线程就返回了 cachedValue，即使第一个线程还没有为 cachedValue 赋值，因此，返回值就是不正确的。

这儿有个经验教训。对于需要同步的单个变量或内存位置，使用 std::atomic 是合适的，但是一旦有两个或以上的变量或内存位置，你应该使用互斥锁。对于 Widget::magicValue，看上去应该是这样的：

```
class Widget {
public:
    ...
    int magicValue() const
    {
        std::lock_guard<std::mutex> guard(m); // lock m
        if (cacheValid) return cachedValue;
        else {
            auto val1 = expensiveComputation1();
            auto val2 = expensiveComputation2();
            cachedValue = val1 + val2;
```

```
            cacheValid = true;
            return cachedValue;
        }
    } // unlock m
    ...
private:
    mutable std::mutex m;
    mutable int cachedValue;          // no longer atomic
    mutable bool cacheValid{ false }; // no longer atomic
};
```

现在，可以肯定本条款的建议了，因为多个线程可以同时执行 const 成员函数。如果你写的 const 函数不是那样的情况，也就是你可以保证永远不会有多个线程执行该成员函数，那么函数的线程安全性就无关紧要。例如，专为单线程使用而设计的类成员函数，是否考虑线程安全是不重要的。在这种情况下，你可以避免使用 mutex 和 std::atomic 带来的开销以及因为它们而导致类变成 move-only。然而，这种线程无关的场景越来越少见，越来越珍贵。可以肯定的是，const 成员函数将是并发执行的主题，这也就是你为什么要保证你的 const 成员函数是线程安全的。

需要铭记的要点
● 除非你能肯定不存在并发的上下文，否则一定要保证 const 成员函数是线程安全的。 ● std::atomic 的性能比 mutex 好，但只适合于操作一个变量或内存地址的情况。

条款 17：理解特殊成员函数的生成

在官方的 C++说法中，特殊的成员函数就是是那些 C++愿意自动生成的函数。C++ 98 有四个这样的函数：默认构造函数、析构函数、拷贝构造函数和拷贝赋值操作符。这些函数只有在需要时才会生成，即，代码用到了这些函数，而类中又没有明确声明。一个默认的构造函数只有在该类没有声明任何构造函数时才生成。（这样可以防止在你已经指定构造函数需要参数的情况下，编译器还为你创建默认构造函数。）自动生成的特殊成员函数默认是 public 和 inline，并且是非虚的，除非是派生类的析构函数，并且继承的基类是虚析构函数。那种情况下，派生类的析构函数也是虚函数。

虽然你们可能已经知道这些事了，但时代变了，C++中特殊成员函数的自动生成规则也随之改变了。了解这些新规则非常重要，因为很少有东西，能像知道编译器何时默默地将成员函数插入类一样，对高效的 C++编程如此重要。

在 C++ 11 中，特殊成员函数俱乐部还有两个成员:move 构造函数和 move 赋值操作符。它们的签名是：

```
class Widget {
public:
    ...
    Widget(Widget&& rhs);              // move构造函数
    Widget& operator=(Widget&& rhs); // move赋值操作
    ...

};
```

它们的生成和行为的规则与拷贝类似，只有在需要的时候才自动生成，而一旦自动生成，它们对非静态数据成员执行的是“逐一移动（memberwise moves）”。这意味着 move 构造函数从参数 rhs 移动构造每个非静态数据成员，而 move 赋值操作符就是 move 赋值每个非静态数据成员，如果有基类的话，同样是如此操作。

现在，当我提到移动操作，即移动构造或移动赋值一个数据成员或基类，不能保证实际上发生的就是 move。实际上，“成员逐一移动”更像是逐一移动请求，因为有些类型可能不是 move-enable(即对 move 操作没有特别支持，例如，大多数 C++ 98 遗留类)将通过它们的拷贝操作“移动”。成员逐一“移动”的核心就是对要移动的对象执行 std::move，并根据结果，在函数重载解析期间，确定到底是执行移动还是拷贝。条款 23 详细介绍了这一过程。对于本条款，请简单地记住，成员逐一移动由支持移动操作的数据成员和基类的移动操作组成，如果某些数据成员或基类不支持移动操作，就执行拷贝操作。

与拷贝操作的情况一样，如果你已经声明了 move 操作，编译器就不会自动生成。然而，它们自动生成的确切条件与拷贝操作略有不同。

两个拷贝操作是相互独立的：声明一个拷贝操作并不会阻止编译器自动生成另一个。如果你声明了一个拷贝构造函数，但没有声明拷贝赋值操作，然后编写了需要拷贝赋值的代码，编译器将自动生成拷贝赋值操作。反之也如此。这在 C++ 98 中是正确的，现在 C++ 11 依然是这样。

两个移动操作不是相互独立的。如果声明了其中任何一个，就会阻止编译器自动生成另一个。其基本原理是，如果声明一个 move 构造函数，那也就表明移动构造实现上应该跟编译器自动生成的有所不同。而且，如果编译器默认的移动构造操作有可能出错的话，那么默认的移动赋值操作也可能会出错。因此，声明一个移动构造函数可以防止移动赋值操作自动生成，反之也如此。

进一步地讲，任何一个类，如果显式声明了拷贝操作，移动操作就不会自动生成。理由就是，声明了一个拷贝操作（构造或赋值）也就表明成员逐一拷贝的方式不适合这个类，而编译器认为，如果成员逐一拷贝不合适，那么移动操作的逐一成员移动可能也不合适。

反之亦如是。声明了移动操作(构造或赋值)也会导致编译器禁用默认的拷贝操作。(拷贝操作是通过 delete 关键字来禁用的—请参见条款 11)。毕竟，如果成员逐一移动不合适的话，没有理由认为成员逐一拷贝就是合适的拷贝方式。这听起来可能会破坏 C++ 98 代码，因为启用默认拷贝操作的条件，C++ 11 比 C++ 98 的约束更多，但事实并非如此。C++ 98 代码是没有移动操作的，因为在 C++ 98 中没有“移动”对象这样的东东。只有一种情况，C++98 的遗留类才会拥有用户声明的移动操作，就是要用到 move 语义特性，并且自动生成的规则不满足的时候，类就必须修改添加用户声明的移动操作。

也许你听说过一条叫做“三原则（Rule of Three）”的指导方针。三原则指出，如果要声明了拷贝构造函数、拷贝赋值运算符或析构函数中任何一个，你就应该把这三个都声明了。这个是根据观察得出来的，也就是说，如果需要自定义拷贝操作，几乎总是因为类需要执行某种资源管理，这也就暗示着：(1)无论在一个拷贝操作中执行的是怎样的资源管理，另一个拷贝操作肯定也要那么干，(2)类析构函数也会参与资源管理(通常是释放资源)。要管理的经典资源就是内存，这就是为什么所有那些管理内存的 STL 类（如，STL 容器执行的是动态内存管理）都声明“三巨头”：两个拷贝操作和一个析构函数。

“三规则”的一个重要推论是，用户声明的析构函数表明简单的成员逐一拷贝可能不太适合这个类的拷贝操作。这进而表明，如果类声明了析构函数，则拷贝操作可能就不应该自

动生成，因为它们可能是不正确的。在 C++ 98 时代，这条原则没有得到充分的重视，所以在 C++ 98 中，用户声明析构函数对编译器自动生成拷贝操作没有影响。C++ 11 仍然保持这样，但这仅仅是因为限制了自动生成拷贝操作的条件会破坏太多遗留代码。

然而，“三原则”背后的推理仍然有效，而且结合声明拷贝操作将排除隐式生成移动操作，推论出 C++ 11 在用户声明析构函数的情况下，不会再自动生成移动操作。

所以，移动操作只有满足下面三个条件时，才会自动生成：

（1） 没有声明任何拷贝操作。

（2） 没有声明任何移动操作。

（3） 没有声明析构函数。

在某种程度上，类似的规则可以扩展到拷贝操作，因为 C++ 11 不赞成为已经声明某个拷贝操作或析构函数的类再自动生成拷贝操作。这意味着，如果你有代码依赖某个类自动生成的拷贝操作，而这个类已经声明了另一个拷贝操作或析构函数，你应该考虑升级这些类以消除依赖性。如果编译器生成的函数的行为是正确的(即，如果逐一拷贝非静态数据成员就是你想要的)，你的升级作很容易，因为 C++的“=default”可以让你显式地指明采用编译器生成：

```
class Widget {
public:
    …
    ~Widget();                                  // 用户声明的析构函数
    …
    Widget(const Widget&) = default;            // 默认构造函数，行为OK
    Widget&
        operator=(const Widget&) = default;     // 默认拷贝赋值，行为OK
    …

};
```

这种方法在多态基类中通常很有用，多态就是定义接口类，派生类对象可以通过它进行操作。多态的基类通常有虚拟析构函数，因为如果没有的话，一些操作(例如，通过基类指针或引用对派生类对象执行 delete 或 typeid 操作)会产生未定义或误导的结果。除非类继承了虚析构函数，而使析构函数成为虚析构函数的唯一方法就是显式声明。通常，默认实现就是正确的，那么用“= default”就是很便捷的方式。但是，用户声明的析构函数会阻止自动生成移动操作，因此如果要支持可移动性，“= default”通常还要用在移动操作上。同时，声明移动操作将禁用拷贝操作，因此如果也需要可拷贝性，还要在拷贝操作上多加几个“= default”：

```
class Base {
```

```
public:
    virtual ~Base() = default;    // make dtor virtual
    Base(Base&&) = default;       // support moving
    Base& operator=(Base&&) = default;
    Base(const Base&) = default; // support copying
    Base& operator=(const Base&) = default;
    …
};
```

事实上，即使有一个编译器愿意生成拷贝和移动操作，并且生成的函数行为也是你想要的，你可能也会选择自己声明并使用“= default”的策略。这当然要多写代码，但会让你的意图更清晰，而且会帮助避开一些相当微妙的 bug。例如，假设你有一个表示字符串表（string table）的类，即，该数据结构允许通过整数 ID 快速查找字符串值：

```
class StringTable {
public:
    StringTable() {}
    …  // 有插入、删除、查找等函数,但没有拷贝、移动、析构等函数
private:
    std::map<int, std::string> values;
};
```

假设该类没有声明拷贝操作、移动操作和析构函数，如果用到的话，编译器将自动生成，这当然是非常方便的。

但是假设以后某个时候，它决定在默认构造和析构函数加日志。添加功能很简单：

```
class StringTable {
public:
    StringTable()
    { makeLogEntry("Creating StringTable object"); }   // added
    ~StringTable()                                     // also
    { makeLogEntry("Destroying StringTable object"); } // added
    …                                  // other funcs as before
private:
    std::map<int, std::string> values; // as before
};
```

这看起来是合理的，但是声明析构函数具有潜在的重要影响：阻止了移动操作的生成。不过，类的拷贝操作不受影响。因此，代码可能会编译、运行，并通过了功能测试。这包括测试它的 move 功能，因为即使这个类不再支持移动，移动它的请求也会被编译和运行。如本条款前面所述，这种情况下执行的拷贝。这意味着“移动”StringTable 对象实际上变成了拷

贝，也就是拷贝底层那些 std::map<int, std::string>对象。而拷贝 std::map<int, std::string>可能比移动慢几个数量级。因此，向类中添加析构函数的简单操作可能会引入显著的性能问题！如果拷贝和移动操作明确使用“= default”定义，问题就不会出现。

现在，我没完没了的唠叨了 C++ 11 管理拷贝和移动操作的原则，你可能想知道我什么时候会将注意力转向其他两个特殊成员函数，默认构造函数和析构函数。那就现在讲吧，但也就那么几句话，因为两个成员函数几乎没啥变化：C++ 11 中的规则与 C++ 98 中的规则几乎相同。

因此，总结下来，C++11 管理这些特殊成员函数的规则是：

1) **默认构造函数**：跟 C++98 规则一样，只有用户未声明任何构造函数的类才会生成。
2) **析构函数**：与 C++98 规则基本相同，唯一的区别是析构函数默认情况下 noexcept(见条款 14)。就像跟 C++ 98 中一样，只有当基类析构函数是 virtual，派生类析构函数才是 virtual。
3) **拷贝构造函数**：与 C++ 98 运行时行为相同：逐一拷贝构造非静态数据成员。只有用户未声明拷贝构造函数的类才会生成。如果类声明了移动操作，则删除拷贝构造。在用户声明了拷贝赋值运算符或析构函数的类中，不赞成让编译器自动生成拷贝构造。
4) **拷贝赋值运算符**：与 C++ 98 运行时行为相同：逐一拷贝赋值非静态数据成员。只有用户未声明拷贝赋值运算符的类才会生成。如果类声明了移动操作，则删除拷贝赋值运算符。在用户声明了拷贝构造或析构函数的类中，不赞成让编译器自动生成拷贝赋值运算符。
5) **移动构造和移动赋值运算符**：每个都是逐一移动非静态数据成员。只有用户未声明拷贝操作、移动操作或析构函数的类才会生成。

注意，规则中没有提到成员函数模板会阻止编译器生成特殊的成员函数。也就是说，如果 Widget 是这样的，

```
class Widget {
    ...
    template<typename T>              // construct Widget
    Widget(const T& rhs);             // from anything
    template<typename T>              // assign Widget
    Widget& operator=(const T& rhs); // from anything
    ...
};
```

编译器仍然会为 Widget 生成拷贝和移动操作(假设管理这些函数生成的通常件已经满足),即使这些函数可以实例化成拷贝构造函数和拷贝赋值运算符的样子。(如果 T 是 Widget，情况就会是这样。)在所有的可能性中，这可能会让你觉得这是一个几乎不值得承认的边界情况，但是有一个我提到它的原因。条款 26 表明它可以产生严重后果。

需要铭记的要点
● 特殊成员函数就是那些编译器可能自动生成的函数：包括默认构造函数、析构函数、拷贝操作和移动操作。 ● 只有没有显式声明移动操作、拷贝操作和析构函数的那些类，移动操作才会生成。 ● 只有没有显式声明拷贝构造的类，拷贝构造才会生成，并且如果声明了任何一个移动操作，它就标记为 delete。只有没有显式声明拷贝赋值运算符的类，拷贝赋值运算符才会生成，并且如果声明了任何一个移动操作，它也标记为 delete。在显式声明了析构函数的类中，拷贝操作不赞成自动生成。 ● 成员函数模板不会阻止特殊成员函数的生成。

第四章智能指针

诗人和作曲家都会有一些关于爱的作品，有时候也会有一些关于计数的，偶尔会有关于两者的。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."，受这些诗句启发，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对象还是一个数组；
2. 从它的声明看不出，即如果指针拥有(owns)它指向的内容，你用完之后是否应该销毁；
3. 当你确定要销毁的时候，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. 当你终于整明白要使用 delete 来销毁的时候，根据第 1 条，你又不知道该使用 delete 还是 delete[]，因为一旦使用错误，结果可能是不确定的；
5. 假设你确定指针指向的内容是啥，也确定该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，多执行哪怕一次都会产生不确定的行为；
6. 如果一个指针已经悬挂，即指针指向的内存对象已经不存在，现在没有办法知道。当一个指针指向的对象被销毁了，该指针就变成了野指针。

当然，原始指针是强大的工具，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。

智能指针是解决这些问题的一种方法。智能指针是原生指针的再包装，它们表现起来很像原生指针，同时也规避了原生指针的许多陷阱。因此，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。

C++ 11 中有四个智能指针：std::auto_ptr、std::unique_ptr、std::shared_ptr 和 std:: weak_ptr。所有这些设计都是为帮助管理动态分配对象的生命周期，即，确保对象在适当的时间以适当的方式(包括异常情况)销毁，从而避免资源泄漏。

std::auto_ptr 是 C++ 98 遗留下来的，C++11 中不推荐使用。C++98 尝试用 std::auto_ptr 来标准化后来 C++11 中 std::unique_ptr 的行为。为了达到目标，move 语义是不可少的，但 C++ 98 当时还没有，所以做了个妥协方案，std::auto_ptr 用拷贝操作来模拟移动。这导致了

令人惊讶的代码(如拷贝一个 std::auto_ptr 会将其设置为 null！)，并且存在令人沮丧的使用限制(例如，不能在容器中存储 std:: auto_ptr)。

std::unique_ptr 可以做 std::auto_ptr 所做的一切事情，甚至更多。它的效率是一样的，也没有扭曲拷贝的含义。它各方面都比 std::auto_ptr 好。std::auto_ptr 唯一合理的使用场景就是需要用 C++ 98 编译器来编译代码。除非存在那个约束，否则你应该将 std::auto_ptr 替换为 std::unique_ptr，并且永远不回头。

智能指针的 API 有显著区别，唯一相同的功能就是默认构造。因为这些 API 的全面介绍满大街都是，所以我将把讨论的重点放在 API 介绍上缺少的知识，例如，值得注意的使用场景，运行性能分析，等等。掌握这些信息，你也就不仅仅局限于会用智能指针，而是会高效地运用。

条款 18：使用 std::unique_ptr 管理独占资源

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(包括解引用)，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 表达的是独占(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(源指针置为 null)。不允许拷贝一个 std::unique_ptr，因为假如真的可以拷贝 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销毁那块资源。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 会销毁它所拥有的资源。默认情况下， std::unique_ptr 会使用 delete 来释放它所包装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。

```
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>          // return std::unique_ptr
std::unique_ptr<Investment>       // to an object created
makeInvestment(Ts&&... params);   // 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 =                // pInvestment is of type
        makeInvestment( arguments );  // std::unique_ptr<Investment>
    ...
}
```

它们也可以在拥有权转移的场景中使用，例如当工厂函数返回的 std::unique_ptr 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，管理的资源也因此得到释放。

默认情况下，析构会使用 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)：当资源释放时，传入的特定析构方法(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 makeInvestment 创建的对象不应该直接 delete，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)        // custom
                {                                  // deleter
                    makeLogEntry(pInvestment);     // (a lambda
                    delete pInvestment;            // expression)
                };

template<typename... Ts>                           // revised
std::unique_ptr<Investment, decltype(delInvmt)>    // return type
makeInvestment(Ts&&... params)
{
```

```
    std::unique_ptr<Investment, decltype(delInvmt)> // ptr to be
        pInv(nullptr, delInvmt);                    // 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;
}
```

稍后，我将解释这是如何工作的，但是首先考虑一下，如果你是一个调用者，事情会是怎样的。假设你将 makeInvestment 调用的结果存储在一个 auto 变量中，你可以无需关注所用资源的释放是特殊处理的。事实上，你的确是超级幸福的，因为使用了 std::unique_ptr 就意味着你不需要关心资源何时应该销毁，更不要说确保程序的每条路径上销毁只发生一次。std::unique_ptr 负责所有这些事情。站在用户的角度来看，makeInvestment 的接口设计是很好的。

一旦你理解了下面这些，你会明白，它的实现也是相当不错的：

1) delInvmt 是 makeInvestment 返回对象的自定义 deleter。所有自定义删除函数接受一个指向要销毁对象的原始指针，然后做必要的事情来销毁那个对象。在这种情况下，就是调用 makeLogEntry 然后执行 delete。使用 lambda 表达式创建 delInvmt 很方便，但是，正如我们很快将看到的，它也比写一个常规函数更高效。

2) 当使用自定义 deleter 时，必须将其类型指定为 std::unique_ptr 的第二个类型实参。在这个例子中就是 delInvmt 的类型，这也是为什么 makeInvestment 的返回类型是 std::unique_ptr<Investment, decltype(delInvmt)>。(decltype 见条款 3)。

3) makeInvestment 的基本策略是创建一个空的 std::unique_ptr，使它指向适当类型的对象，然后返回它。为将自定义删除器 delInvmt 与 pInv 关联，我们将其作为构造函数的第二个实参。

4) 试图将一个原生指针(例如，new 出来的)赋值给 std::unique_ptr 是编译不通过的，因为它将构成从原生指针到智能指针的隐式转换。这种隐式转换可能有问题，所以

C++ 11 智能指针禁止这种转换。这就是为什么使用 reset 让 pInv 拥有 new 出来的对象。

5) 随着每次使用 new，我们使用 std::forward 来完美转发传递给 makeInvestment 的实参 (见条款 25)。这样调用者提供的所有信息就都传递给了构造函数。

6) 自定义删除器接受类型为 Investment*的参数。不管 makeInvestment 中创建的对象的实际类型是啥（Stock、Bond 或者 RealEstate），它最终都是一个 Investment*对象在 lambda 表达式中被删除。这意味着我们是要通过基类删除派生类对象指针。要实现这一点，基类 Investment 必须有一个虚析构函数：

```
class Investment {
public:
    …                          // essential
    virtual ~Investment();     // design
    …                          // component!
};
```

C++ 14 支持函数返回类型推导(见条款 3)，意味着 makeInvestment 的实现可以更简单、封装更好：

```
template<typename... Ts>
auto makeInvestment(Ts&&... params) // C++14
{
    auto delInvmt = [](Investment* pInvestment)    // lambda表达式放
                    {                              // makeInvestment里面
                        makeLogEntry(pInvestment);
                        delete pInvestment;
                    };
    std::unique_ptr<Investment, decltype(delInvmt)>    // as
        pInv(nullptr, delInvmt);                       // before
    if ( … )                                           // as before
    {
        pInv.reset(new Stock(std::forward<Ts>(params)...));
    }
    else if ( … )                                      // as before
    {
        pInv.reset(new Bond(std::forward<Ts>(params)...));
    }
    else if ( … )                                      // as before
    {
        pInv.reset(new RealEstate(std::forward<Ts>(params)...));
    }
    return pInv;                                       // as before
```

```
}
```

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 的大小增长一个到两个字节。因为 deleters 是函数对象，大小的改变就取决于函数对象要存储多少状态。无状态的函数对象(如，没有 captures 的 lambda 表达式) 不会导致额外的大小开销。这就意味着当一个自定义的 deleter 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 是第一优选：

```
auto delInvmt1 = [](Investment* pInvestment)        // 自定义deleter是
                 {                                  // 无状态lambda
                     makeLogEntry(pInvestment);
                     delete pInvestment;
                 };

template<typename... Ts>                            // 返回类型大小等于
std::unique_ptr<Investment, decltype(delInvmt1)>    // Investment*指针大小
makeInvestment(Ts&&... args);

void delInvmt2(Investment* pInvestment)             // 自定义deleter是函数
{
    makeLogEntry(pInvestment);
    delete pInvestment;
}

template<typename... Ts>                            // 返回类型大小等于
std::unique_ptr<Investment,                         // Investment*指针大小
                void (*)(Investment*)>              // 至少加上函数指针大小！
makeInvestment(Ts&&... params);
```

带有过多状态的函数对象 deleter 会显著增加 std::unique_ptr 的大小。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，请考虑改变你的设计。

工厂函数不是 std::unique_ptr 的唯一常见用例。它们更流行的是作为实现 Pimpl 用法的机制。具体参考条款 22。

std::unique_ptr 有两种格式，一种是独立对象(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。它的 API 设计可以匹配你的任何使用形式。例如，单对象格式中没有索引运算符(operator [])，而数组格式则没有解引用运算符(operator*和 operator->)。

std::unique_ptr 的数组格式对你来说可能是华而不实的东东，因为相对于原生数据，

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 来返回一个指向堆数组的原生指针，并且接管所有权。

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高效的转化为 std::shared_ptr：

```
std::shared_ptr<Investment> sp = // converts std::unique_ptr
    makeInvestment( arguments ); // 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一个关键点。因为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 ，工厂函数给调用者提供了有效的智能指针，同时也不阻止他将 std::unique_ptr 转成其他灵活的智能指针类型(std::shared_ptr ，继续看 Item 19)。

需要铭记的要点
● std::unique_ptr 是一个开销小、速度快、move-only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，也可以自定义deleter。但有状态的deleter和函数指针作为自定义deleter会增大 std::unique_ptr 的存储开销。 ● 很容易将一个 std::unique_ptr 转化为 std::shared_ptr。

条款 19：使用 std::shared_ptr 管理共享资源

使用垃圾回收机制的程序员指责并且嘲笑 C++程序员防止内存泄露的做法。“你们太原始了!”他们嘲笑道。“你们有没有看过 1960 年 Lisp 语言的备忘录？应该由机器管理资源的生命周期，而不是人类。”C++程序员开始翻白眼了："你妹的指的是那个备忘录吗？说唯一的资源是内存，而且资源回收的时机是不确定性的，谢谢，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的确像是用原始工具来做一个记忆性的内存回路。为什么鱼和熊掌不能兼得呢？做一个像垃圾回收那样的自动化系统，然后还可以运用到任何资源，并且具有像析构函数可预测的回收时机。

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 访问的对象被那些具备共享所有权(shared ownership)的智能指针管理。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，引用计数与资源关联的一个值，它记录有多少个 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（通常是这样——后面再解释），析构函数会自动递减这个计数，而拷贝赋值运算符两者都做(比如，sp1 和 sp2 都是 std::shared_ptr 类型，它们指向了不同的对象，赋值操作 sp1=sp2 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 在递减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对象。

引用计数的存在对性能会产生部分影响：

1) **std::shared_ptr 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引用计数的原生指针。
2) **引用计数的内存必须被动态分配**。概念上来说，引用计数与被指向的对象关联，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。)条款 21 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。

3) **引用计数的递增或者递减必须是原子操作**，因为在多线程环境下，会同时存在多个读写。例如，在一个线程中，一个指向资源的 std::shared_ptr 即将析构(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，为什么我们不在构造 std::sharedptr 时总是增加引用计数呢？

Move 构造函数就是那个特例。从一个 std::shared_ptr 移动构造 std::shared_ptr 会使得源 std::shared_ptr 指向 null，这就意味着新的 std::shared_ptr 取代老的 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。移动构造要比拷贝构造，因为拷贝需要修改引用计数，而移动却不需要。赋值与构造的情况一样，总结来说就是，移动构造要比拷贝构造快，移动赋值要比拷贝赋值快。

像 std::unique_ptr (条款 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。然而，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)  // custom deleter
                  {               // (as in Item 18)
                      makeLogEntry(pw);
                      delete pw;
                  };

std::unique_ptr<                  // deleter类型是指针类型的组成部分
    Widget, decltype(loggingDel)
    > upw(new Widget, loggingDel);
```

```
std::shared_ptr<Widget>          // deleter类型不是指针类型的组成部分
    spw(new Widget, loggingDel);
```

std::shared_prt 的设计更加的弹性一些，考虑到两个 std::shared_ptr，分别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) { … };  // custom deleters,
auto customDeleter2 = [](Widget *pw) { … };  // each with a
                                             // different type
std::shared_ptr<Widget> pw1(new Widget, customDeleter1);
std::shared_ptr<Widget> pw2(new Widget, customDeleter2);
```

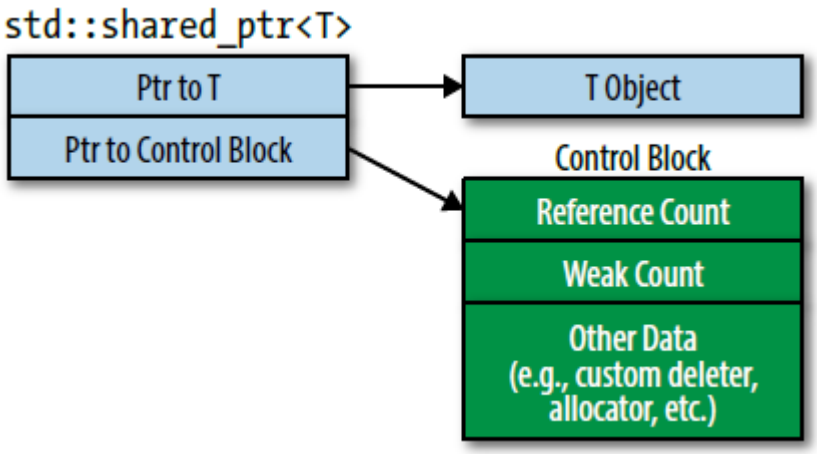
因为 pw1 和 pw2 属于相同类型，所以它们可以放置到属于同一个类型的容器中去：

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

它们之间可以相互赋值，也都可以传递给参数类型为 std::shared_ptr<Widget>的函数。所有的这些特性，自定义 deleter 类型不同的 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 另一个不同点是，为 std::shared_ptr 指定自定义的 deleter 不会改变 std::shared_ptr 的大小。不管 deleter 是啥样的，std::shared_ptr 始终是两个指针大小。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，而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，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 。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。该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(在指定的情况下)。如果指定了一个自定义的 allocator，它的拷贝也会被包含在其中。控制块也可能包含其他的额外数据，比如条款 21 所说的，第二个叫作 weak count 的引用计数，在本条款中我们先略过它。我们可以想象出 std::shared_ptr<T> 的内存布局如下所示：



一个对象的控制块由创建第一个 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：

1) std::make_shared (请看条款 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。

2) 当从一个独占性的指针(例如，std::unique_ptr 或者 std::auto_ptr)构建 std::shared_ptr 时，控制块被相应的被创建。因为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置为 null)。

3) 当用原生指针构造 std::shared_ptr 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并指向必要的控制块。

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(每个引用计数一次)。这就意味着下面的代码着实糟糕透顶：

```
auto pw = new Widget;                          // pw is raw ptr
...
```

```
std::shared_ptr<Widget> spw1(pw, loggingDel); // create control block
                                              // for *pw
...
std::shared_ptr<Widget> spw2(pw, loggingDel); // create 2nd
                                              // control block for *pw!
```

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：优先使用智能指针而不是原生指针。但是先不管这么多，创建 pw 的那行代码确实不太建议，但是至少它不会导致不确定的程序行为.

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(因此有个引用计数)。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)。因此*pw 拥有两个引用计数。每一个最终都会变成 0，最终会引起两次对 *pw 的析构行为。第二次析构导致未定义行为了。

关于 std::shared_ptr 的使用，这里至少有两点教训。首先，避免给 std::shared_ptr 构造函数传递原生指针。通常的做法是使用 std::make_shared(请看条款 21)。但是在上面的例子中，我们使用了自定义的 deleter，没法调用 std::make_shared。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写成下面这样：

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

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)。这样无论如何都不会有问题：

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

令人惊讶的时，使用原生指针作为 std::shared_ptr 构造参数，有可能产生多个控制块。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 有一个执行处理过程的成员函数：

```
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 of
}                                           // 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，请看条款 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，这就完蛋了，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头昏脑涨，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，它还有一个标准名字，虽然也像 std::enable_shared_from_this 一样古怪，名字叫 The Curiously Recurring Template Pattern(CRTP)。

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 ，当你在成员函数中要创建一个指向 this 对象的 std::shared_ptr，可以在其中调用 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)，尽管 shared_from_this 通常会抛出异常，但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:
    // 工厂函数将参数完美转发至private 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，是因为我们想要理解 std::shared_ptr 的性能开销。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作，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，请看条款 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)需要一到两个原子操作，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 或许是更好的选择。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。

反过来可就不行了。如果你把一个资源的生命周期管理交给了 std::shared_ptr ，后面没有办法改变了。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。也是与 std::unique_ptr 的另一个不同点， std::shared_ptr 的 API 专为单个对象设计。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；其次，std::shared_ptr 支持子类到基类的转换，这个对单个对象很有意义，但应用到数据组，就像是在类型系统上打洞。（基于这个原因，std::unique_ptr<T[]> API 禁止这样的转换。）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)，给数组声明一个智能指针通常是不当设计。

需要铭记的要点
● std::shared_ptr 为任意资源的共享式管理提供了自动垃圾回收式的便利。 ● std::shared_ptr 是 std::unique_ptr 的两倍大，除了控制块，还有引用计数的原子操作带来的开销。 ● 资源的默认析构操作是 delete，但也支持自定义的 deleter。deleter 的类型不会影响 std::shared_ptr 的类型。 ● 避免从原生指针类型变量创建 std::shared_ptr。

条款 20：对可能悬挂（dangle）的 std::shared_ptr-like 指针使用 std::weak_ptr

说起来有些矛盾，如果有这样一种智能指针，表现的像 std::shared_ptr ，但又不参与资源的共享式管理，那样可能会比较方便。换句话说，一个类似于 std::shared_ptr 但不影响引用计数的指针。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，你甚至会更加怀疑。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 的一种附加类型。

这种关系与生俱来，std::weak_ptr 通常由 std::shared_ptr 创建，它们指向相同的对象，但是 std::weak_ptr 不会影响到它所指向对象的引用计数：

```
auto spw =                          // spw创建后，引用计数是1
    std::make_shared<Widget>();
…
std::weak_ptr<Widget> wpw(spw);     // wpw 指向跟spw同样的Widget，
                                    // 引用计数仍然是1
…
spw = nullptr;                      // 引用计数变成0，Widget被销毁，wpw悬挂
```

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()) … // if wpw doesn't point to an object…
```

我们经常要做的是：查看 std::weak_ptr 是否已经过期，如果没有过期的话，即没有悬挂，就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，也就没办法写这样的代码。即使有办法，但将检查和解引用分开的写法，也会引入一个竞争：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。就是通过 std::weak_ptr 来创建 std::shared_ptr 来实现。以 std::weak_ptr 为参数创建 std::shared_ptr ，共有两种方式，当 std::weak_ptr 已经

过期，产生的结果也不一样。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(); // if wpw's expired,
                                           // spw1 is null
auto spw2 = wpw.lock();                    // same as above, but uses 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);    // if wpw's expired,
                                      // throw std::bad_weak_ptr
```

你可能还是疑惑， std::weak_ptr 到底有啥用。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。根据条款 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工厂返回的对象，这个对象会被销毁，对应的缓存对象会悬挂，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 loadWidget 缓存版本的一个应急（quick-and-dirty）实现：

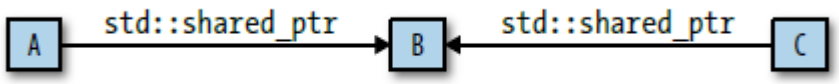
```
std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
{
    static std::unordered_map<WidgetID,
    std::weak_ptr<const Widget>> cache;
    auto objPtr = cache[id].lock();   // objPtr is std::shared_ptr
                                      // to cached object (or null
                                      // if object's not in cache)
    if (!objPtr) {                    // if not in cache,
```

```
            objPtr = loadWidget(id);      // load it
            cache[id] = objPtr;           // cache it
        }
        return objPtr;
    }
```

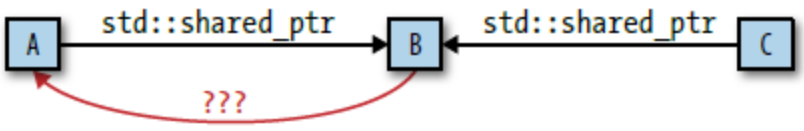
这个实现利用了 C++11 的 hash 表容器(std::unordered_map)，尽管它没有提供所需的 WidgetID 哈希算法以及等式比较函数。

fastLoadWidget 的实现忽略了一个情况，就是缓存可能积压那些已经过期的 std::weak_ptr (对应的 Widget 不再使用，已经被销毁了)。所以它的实现还可以再改进，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：观察者模式。该模式的主要组成部分是：状态可能会发生变化的主题，以及当状态变化时需要得到通知的观察者。在大多数实现中，每一个主题都包含一个数据成员来存储所有观察者的指针，这就使得主题很容易发送出状态变化的通知。主题没有兴趣去控制观察者的生命周期(即观察者何时被析构)。但是，它们必须知道，如果一个观察者析构了，主题就不能尝试去访问它了。一个合理的设计是：每一个主题包含一个容器来存储指向各个观察者的 std::weak_ptr，这样就可以在访问之前，先检查一下指针是否处于悬挂状态。

下面讲最后一个 std::weak_ptr 实用的例子，有这样一个数据结构，包含 A、B 和 C 三个对象。A 和 C 共享 B 的所有权，它们都包含了一个 std::shared_ptr 指向 B：



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？



有三种选择：

1) **原生指针**。这种方式下，A 如果析构了，但是 C 继续指向 B，B 中指向 A 的指针现在处于悬挂状态。而 B 对此毫不知情，所以 B 有可能不小心解引用那个野指针，这样会产生未定义的行为。

2) **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，并且 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 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 都会内存泄露，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。

3) **std::weak_ptr** 。这个方式避免了以上两个问题。如果 A 析构了，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 环，这样的需求不是那么普遍。在层级比较严格的数据结构中，比如说树，子节点一般只被父节点拥有，当父节点析构时，子节点也应该会被析构。因此，父节点指向子节点的链接最好使用 std::unique_ptr。因为子节点的生命周期不应该比父节点长，子节点指向父节点的链接完全可以使用原生指针。因此也不会出现子节点解引用一个指向父节点的野指针。

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。

从效率的角度来看， std::weak_ptr 和 std::shared_ptr 的情况基本相同，二者对象大小相同，它们使用同样的控制块(参考条款 19)，并且诸如构造、析构以及赋值操作都涉及到引用计数的原子操作。这可能让你大吃一惊，因为我在本条款开始的时候说，std::weak_ptr 不参与引用计数。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 操作这个引用计数。欲知详情，请继续阅读条款 21。

需要铭记的要点
● 对可能悬挂（dangle）的 std::shared_ptr-like 指针使用 std::weak_ptr。 ● std::weak_ptr 的潜在应用场景包括缓存、观察者列表以及防止std::shared_ptr环。

条款 21：优先使用 std::make_unique 和 std::make_share，而非直接使用 new

我们先给 std::make_unique 以及 std::make_shared 提供一个公平的竞争环境。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 C++ 14 才加入标准库。如果你在使用 C++11。不要怕，因为你自己可以很容易地写一个基本版的 std::make_unique ，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 只是将它的参数完美转发到对象的构造函数，由 new 出来的原生指针构造一个 std::unique_ptr ，并将之返回。这种函数格式不支持数组和自定义 deleter(参考条款 18)，但是它说明只需稍加努力，便可自己创造出所需的 make_unique。请记住不要把你自己实现的版本放在命名空间 std 下面，因为假如日后升级到 C++ 14 的标准库，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 一样。

在是否使用 make 函数来创建智能指针这件事上，即使是最普通的比较，也说明了为什么使用 make 函数是更可取的。考虑以下代码：

```
auto upw1(std::make_unique<Widget>());     // with make func
std::unique_ptr<Widget> upw2(new Widget); // without make func

auto spw1(std::make_shared<Widget>());     // with make func
std::shared_ptr<Widget> spw2(new Widget); // without make func
```

我已经高亮显示了必要的差别：使用 new 需要重复写一遍要创建的类型（即 Widget），而使用 make 函数则不需要。重复敲类型违背了软件工程中的一项基本原则：应当避免代码重复。源代码重复会增加编译次数，导致目标代码臃肿，通常产生出的 code base 都难以维护。它经常会导致产生不一致的代码，一个 code base 中的不一致代码会引发 bug。并且，

敲某段代码两遍会比敲一遍更费事，而且谁不想少敲代码呢？

第二个偏向 make 函数的原因是为了保证异常安全（exception safety）。设想我们有一个函数来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，但是条款 41 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();
```

并且我们调用 processWidget 时，使用 new 而不是 std::make_shared ：

```
processWidget(std::shared_ptr<Widget>(new Widget),      // 可能有内存泄露！
              computePriority());
```

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，资源会自动释放。如果每个人在每个地方都是用的 std::shared_ptr ，怎么会发生内存泄露呢？

这个问题的答案和编译器将源代码翻译为目标代码有关系。在运行时，在函数调用前，函数的实参必须推算出来，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> 的构造函数必须执行。
- computePriority 必须执行。

并没有任何要求说编译器必须按上面的顺序来生成目标代码。"new Widget"必须要在调用 std::shared_ptr 构造函数之前执行，因为 new 的结果作为该构造函数的一个参数，但 computePriority 可能在这些调用之前、之后，甚至是两者之间执行。这样的话，编译器可能按如下操作的顺序生成目标代码：

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

如果生成这样的代码，并且在运行时，computePriority 产生出了一个异常，那么第 1 步

中动态分配的 Widget 可能会产生泄漏，因为它永远不会存储到第 3 步的 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>(), // 没有潜在的内存泄露
              computePriority());
```

在运行时，std::make_shared 或者 computePriority 都有可能先调用。如果先调用 std::make_shared，动态分配的 Widget 安全的存储在 std::shared_ptr 中(在调用 computePriority 之前)。如果 computePriority 产生了异常， std::shared_ptr 的析构函数会负责释放 Widget。如果先调用 computePriority 并且产生一个异常，则不会再调用 std::make_shared，因此不会动态分配 Widget，也就不存在泄露问题。

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，也是同样的情况，应该使用 std::make_unique 代替直接使用 new，这样都是为了异常安全(exception-safe)考虑。

和直接使用 new 相比，使用 std::make_shared 的一个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 的情况：

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

看上去代码只需一次内存分配，但实际上执行了两次。条款 19 提过每个 std::shared_ptr 都指向了一个包含对象引用计数的控制块，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，就需要为 Widget 分配一次内存，然后为控制块再次分配内存。

如果使用的是 std::make_shared，

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

一次分配就足够了。那是因为 std::make_shared 分配了一整块空间，来同时存储 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内存分配调用，并且加快了代码的执行速度，因为只分配一次内存。此外，使用 std::make_shared 避免了在控制块中添加一些额外信息，潜在的减少了程序所需的总内存。

上文的 std::make_shared 效率分析同样适用 std::allocate_shared，所以 std::make_shared 的性能优点，std::allocate_shared 函数也同样具备。

上面对于优先使用 make 函数，而不是直接用 new 的论点，每一个都有理有据。尽管有软件工程、异常安全和性能这些优点，不过，本条款的指导原则是优先使用 make 函数，并不是要我们只使用它们。这是因为有一些情况下，不能或者不应该使用 make 函数。

例如，make 函数都不支持自定义 deleter(请看条款 18 和条款 19)。但是 std::unique_ptr 和 std::shared_ptr 都有构造函数支持。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：

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

make 函数就没法做到这样。

make 函数的第二个限制来自于它们实现的语义细节。条款 7 解释了当创建了一个对象，该对象重载了是否以 std::initializer_list 为参数的两种构造函数，使用花括号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，它到底应该使用括号还是花括号呢？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 是包含值都是 20 的 10 个元素，还是包含 10 和 20 两个值呢？或者说结果是不确定的？

好消息是，结果是确定的：两个调用都产生了同样的 std::vector，拥有 10 个元素，每个元素的值都是 20。这就意味着在 make 函数中，完美转发使用的是括号而非花括号。坏消息是，如果你想要使用花括号来构造指向的对象，你必须直接使用 new，如果要用 make 函数，就需要完美转发花括号初始化，但是，正如条款 30 所说的那样，花括号初始化是没有办法完美转发的。不过，条款 30 也描述了一种变通方案，使用 auto 类型推导从花括号初始化(参考条款 2)创建出一个 std::initializer_list 对象，然后将其传递给 make 函数：

```
// create std::initializer_list
auto initList = { 10, 20 };
// create std::vector using std::initializer_list ctor
auto spv = std::make_shared<std::vector<int>>(initList);
```

对于 std::unique_ptr，只有自定义 deleter 和花括号初始化这两个场景，make 函数不适用。但对于 std::shared_ptr 来说，问题可不止两个了。这两个都是边界情况，但确实有些程序员会面临这种边界情况，你也有可能会碰到。

有些类会定义它们自己的 new 和 delete 运算符。这些函数暗示了为这种类型的对象不

适合用全局的内存分配和回收方法。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 经常设计为：只是处理大小就是 sizeof(Widget)的内存块的分配和回收。而 std::shared_ptr 支持的自定义分配(通过 std::allocate_shared)以及回收(通过自定义 deleter)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，使用 make 确实不合适。

相对于直接使用 new，使用 std::make_shared 的内存大小及性能优势源自于：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，但是，它所占用的内存直到控制块被销毁才会释放，因为被动态分配的内存块同时包含了这两个。

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，记录了多少个 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)。但为了本条款的目的，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(参考条款 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，否则就未过期。

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。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 的析构与最后一个 std::weak_ptr 析构之间的间隔时间很长时，该对象析构与它所占内存的回收之间也会产生间隔：

```
class ReallyBigType { … };
auto pBigObj =                          // 通过std::make_shared
    std::make_shared<ReallyBigType>();// 创建一个非常大的对象
…                                       // 创建一堆std::shared_ptr
                                        // 和std::weak_ptr来使用这个对象
…                                       // 最终std::shared_ptr都销毁了，
                                        // 但std::weak_ptr还有
```

```
…                                   // 这段时间，前面分配的内存还占用着
…                                   // 最终std::weak_ptr全部销毁了，
                                    // 控制块和对象的内存也就释放了
```

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。（因为使用了 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——译者注）

```
class ReallyBigType { … };  // as before

std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);
                            // 通过new创建一个非常大的对象
…                           // 同前面一样，创建一堆std::shared_ptr
                            // 和std::weak_ptr来使用这个对象
…                           // 最终std::shared_ptr都销毁了，
                            // 但std::weak_ptr还有，而对象的内存已经回收
…                           // 这段时间，只有控制块的内存还在
…                           // 最终std::weak_ptr全部销毁了，控制块内存回收
```

当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，只要保证你在一条语句中，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，这也会防止编译器在 new 的使用和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，对于我们之前检查的那个异常不安全的 processWidget 函数，我们做个微小的修改。这次，我们指定一个自定的 deleter：

```
void processWidget(std::shared_ptr<Widget> spw,    // as before
                   int priority);
void cusDel(Widget *ptr);                          // custom deleter
```

这里有一个异常不安全的调用方式：

```
processWidget(                                     // as before,
    std::shared_ptr<Widget>(new Widget, cusDel),   // 潜在的内存泄露！
    computePriority()
);
```

回想一下：如果在"new Widget"之后，但在 std::shared_ptr 构造之前调用 computerPriority，并且 computePriority 抛出了一个异常，那么动态分配的 Widget 会泄露。

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 的构造单独放到一条语句中，然后用得到的 std::shared_ptr 来调用 processWidget，这就是技术的本质，尽管过会儿你还会看到，我们可以对此稍加改进来提升性能。

```
std::shared_ptr<Widget> spw(new Widget, cusDel);
processWidget(spw, computePriority()); // 正确但不是最优的，下面再讲
```

这个确实可行，因为即使构造函数抛出异常，std::shared_ptr 也已经接管了原生指针的所有权。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 is rvalue
    computePriority()
);
```

而在异常安全的调用中，我们传递了一个左值：

```
processWidget(spw, computePriority()); // arg is lvalue
```

因为 processWidget 的 std::shared_ptr 参数按值传递，从右值构造只需要一个 move，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递增，而 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，若想获得和异常非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(参考条款 23)：

```
processWidget(std::move(spw), computePriority()); // 高效且异常安全
```

是不是很有趣，值得一看，但是这种情况不是很常见，因为你也很少有理由不去使用 make 函数。如果不是非要用其他方式不可，我还是推荐你尽量使用 make 函数。

需要铭记的要点
● 与直接使用 new 相比，使用 make 函数减少了代码重复，提升了异常安全度，并且，对于 std::make_shared 以及 std::allocate_shared 来说，生成的目标代码更小、更快。 ● 不适合使用 make 函数的场景包括需要指定自定义 deleter 和要求传递花括号初始化。 ● 对于 std::shared_ptr 来说，make 函数不适合的场景还包含：(1)带有自定义内存管理的类；(2) 系统内存紧俏，对象非常大，并且 std::weak_ptr 比对应的 std::shared_ptr 存活时间长。

条款 22：当使用 Pimpl 方式时，在实现文件中定义特殊成员函数

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，也就意味着，使用 Widget 的用户必须那些头文件。这些头文件增加了用户的编译时间，并且使得用户依赖这些头文件。如果一个头文件改变了，用户就必须重新编译。虽然标准的头文件不怎么发生变化，但 Gadget 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 的用户就可以不包含这些头文件了。这加快了编译速度，并且 Widget 的用户也不受到影响。

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，动态分配和回收一个存储原始类中数据成员的对象(本例中的*pImpl)。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 {            // definition of Widget::Impl
    std::string name;            // with data members formerly
    std::vector<double> data;    // in Widget
    Gadget g1, g2, g3;
};

Widget::Widget()                 // allocate data members for
: pImpl(new Impl)                // this Widget object
{}
Widget::~Widget()                // destroy data members for
{ delete pImpl; }                // this object
```

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。不过，这些依赖从 widget.h 转移到了 widget.cpp(只对 Widget 的实现者可见)中，我已经高亮了分配和回收 Impl 对象的代码。因为需要在 Widget 析构时，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！本章的内容是围绕着“智能指针优于原生指针”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(参考条款 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。头文件的内容变成这样：

```
class Widget { // in "widget.h"
public:
    Widget();
    ...

private:
    struct Impl;
```

```
    std::unique_ptr<Impl> pImpl; // use smart pointer
};
```

实现文件变成这样：

```
#include "widget.h" // in "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl { // as before
    std::string name;
    std::vector<double> data;
    Gadget g1, g2, g3;
};

Widget::Widget()                    // 参考Item 21, create std::unique_ptr
: pImpl(std::make_unique<Impl>()) // 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。

这个代码可以编译，但一个很平凡的用法，就编译不过了：

```
#include "widget.h"
Widget w; // error!
```

错误信息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，让代码工作起来也很简单。我们首先需要理解为啥会出错。

这问题是由于生成 w 析构时的目标代码报错。在此时，析构函数被调用，而在定义使用 std::unique_ptr 的 Widget 类中 ，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(参考条款 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，因为与编译器生成的其他特殊成员函数一样，Widget 的析构函数也是 inline 的。出错指向 w 被创建的那一行，因为该行创建了 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，在生成销毁 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，但不要定义：

```
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 { // as before, definition of
    std::string name; // Widget::Impl
    std::vector<double> data;
    Gadget g1, g2, g3;
};

Widget::Widget()                        // as before
: pImpl(std::make_unique<Impl>())
{}
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 做法的类是天然支持 move 语义的，因为编译器生成的 move 操作正是我们想要的：对底层的 std::unique_ptr 上执行 move 操作。就像条款 17 解释的那样，Widget 声明了析构函数，所以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了，你可能会像下面那样实现：

```
class Widget {        // still in
public:               // "widget.h"
    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，问题的本质也一样。对于编译器生成的 move 赋值运算符，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，情况不一样，它的问题在于编译器通常是在 move 构造函数内抛出异常的事件中，生成析构 pImpl 的代码，而对 pImpl 析构需要 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：

```
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;
};
```

```
#include <string>                       // as before,
…                                       // in "widget.cpp"

struct Widget::Impl { … };              // as before

Widget::Widget()                        // as before
: pImpl(std::make_unique<Impl>())
{}
Widget::~Widget() = default;            // as before
Widget::Widget(Widget&& rhs) = default;             // definitions
Widget& Widget::operator=(Widget&& rhs) = default;
```

Pimpl 做法是一种减少 class 的实现和用户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，那么 Widget 支持拷贝操作就很有意义。我们必须要自己手写这些拷贝函数，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。(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：

```
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;
};

#include "widget.h"                             // as before,
…                                               // in "widget.cpp"

struct 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 结构体。相对于逐个字段拷贝，我们利用了编译器为 Impl 生成的拷贝操作，这些操作会自动拷贝每个字段。因此 Widget 拷贝的实现就是调用编译器生成的 Widget::Impl 拷贝操作。在拷贝构造函数中，我们依然遵循了条款 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，用的是 std::unique_ptr 智能指针，因为对象内(如 Widget 内)的 pImpl 指针拥有实现对象(如，Widget::Impl 对象)的独占所有权。但是，很有意思的是，如果我们用 std::shared_ptr 替代 std::unique_ptr，我们会发现本条款的建议不再适用了。我们没必要在 Widget.h 中声明析构函数，并且，没有了用户自己声明的析构函数，编译器会很乐意生成我们想让它生成的 move 操作。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<Impl> pImpl; // std::shared_ptr
                                 // instead of std::unique_ptr
};
```

并且，引用 widget.h 的用户代码如下，

```
Widget w1;
auto w2(std::move(w1)); // move-construct 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 都得到销毁(这使得指向 Widget::Impl 的对象被销毁)。

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 deleter 的方式不同。对于 std::unique_ptr，deleter 的类型是智能指针的一部分，这就使得编译器能够生成更小的运行时数据结构以及更快的运行时代码。更高效率的结果是要求当编译器生成特殊函数时，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。虽然会造成大一点的运行时数据结构和稍微慢一些的代码。但是在编译器生成特殊函数时，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 两者之间，其实并没有一个真正的权衡。因为 Widget 和 Widget::Impl 之间是独占关系，std::unique_ptr 是很合适的。然而，在一些其他的场景中，存在共享式关系， 那时，std::shared_ptr 才是一个合适的选择，也就没必要像 std::unique_ptr 那样定义那些特殊函数了。

需要铭记的要点
● Pimpl 做法是通过减少用户与实现之间的编译依赖来减少编译次数。 ● 对于 std::unique_ptr 类型的 pImpl 指针，要在头文件声明特殊成员函数，然后在实现文件中实现，即使编译器生成的默认函数实现就行，也必须要手动写一遍。 ● 以上两点建议只适用于 std::unique_ptr，对 std::shared_ptr 不适用。

第五章右值引用、Move 语义和完美转发

当你第一次学习 move 语义和完美转发时，它们看起来非常直观：

- **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函数，这些目标函数接受的实参和 forwarding 函数接受的实现完全一样。

对于这两个完全不同的特性，右值引用就是粘合两者的胶水。它作为底层的语言机制，使得 move 语义和完美转发成为可能。

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低时，也没有你想象的那么低，它也并不总是在 move 有效的上下文中被调用；“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，是真正的直截了当。

在本章的所有条款中，你必须要牢记一点，函数的参数永远是一个左值，即使它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。（确定表达式是否是左值的一个简单方式是，看你能否拿到它的地址）

条款 23：理解 std::move 和 std::forward

依据它们不做什么来理解 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，它们什么都不做。它们不产生可执行代码，一个字节的代码也不产生。

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 无条件的将它的实参转换成一个右值，而 std::forward 只有当特定的条件满足时，才会这样转换。这就是本质。本条款的解释会引出一组新问题，但是，基本上，就是那么一些问题。

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>                            // in namespace std
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);
}
```

我为你高亮了两处代码，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，应该是一个通用引用。请看条款 24。)，并且返回同一个对象的引用。

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，正如条款 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(请看条款 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：将它的实参转换成了右值(rvalue)。

说一句题外话，在 C++14 中，std::move 可以实现的更优雅。受益于函数返回类型推导(请看条款 3)和标准库别名模板 std::remove_reference_t (请看条款 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 除了将它的实参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，右值是适合移动的合格候选者，所以对一个对象应用 std::move 就是告诉编译器，该对象很适合执行移动操作，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，右值并不总是适合 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。依据条款 41，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，将 text 改为 const：

```
class Annotation {
public:
    explicit Annotation(const std::string text)
    …

};
```

为了避免拷贝 text 到对象成员变量带来的拷贝代价。你继续依据 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;
};
```

这个代码可以编译、链接、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之处是，它不是将 text 移动到 value，而是拷贝到 value。text 确实被 std::move 转化成了 rvalue，但是 text 被声明为 const std::string。所以在转换之前，text 是一个 const std::string 类型的左值。转换的结果是一个 const std::string 右值，但是自始至终，const 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 构造函数的成员初始化列表， std::move(text) 的结果是 const std::string 右值，这个右值不能传递给 std::string 的 move 构造函数，因为 move 构造函数接收的是非 const 的 std::string 右值引用。不过，因为 lvalue-reference-to-const 的参数类型允许绑定 const 右值，所以右值可以传递给拷贝构造函数。因此即使 text 被转换成了右值，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 对象传递给像移动构造这样会改变此对象的函数。

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 对象的移动请求通常会悄悄地转成拷贝。其次，std::move 不仅是不移动任何东东，它也不保证要转换的对象一定是适合移动的，执行 std::move 后唯一能确定的就是，得到一个右值。

std::forward 的情况和 std::move 类似，但是和 std::move 无条件地将它的实参转换成右值不同，std::forward 在特定的条件下才会执行转换。std::forward 是一个有条件的转换。为了理解它何时转换何时不转换，我们来回想一下 std::forward 的典型应用场景。最常见的场景是：一个函数模板接受一个通用引用参数，并将它传递给另外一个函数：

```
void process(const Widget& lvalArg);  // process lvalues
```

```
void process(Widget&& rvalArg);        // process rvalues

template<typename T>                   // template that passes
void logAndProcess(T&& param)          // param to process
{
    auto now =                         // get current time
        std::chrono::system_clock::now();
    makeLogEntry("Calling 'process'", now);
    process(std::forward<T>(param));
}
```

考虑对 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 传递给了函数 process。process 对左值和右值做了重载。当我们用左值调用 logAndProcess 时，我们自然地期望，转发给 process 的也是左值，当我们用右值来调用 logAndProcess 时，我们希望调用 process 的右值重载版本。

但是就像所有的函数参数一样，param 本身是左值。因此 logAndProcess 内，每一次调用 process 都会是调用左值重载版本。为了防止这种情况，我们需要一个机制，当且仅当 logAndProcess 的实参是右值时，param 才会转换成右值。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换：只有实参是右值时，才将参数转化为右值。

你可能想知道 std::forward 怎么知道实参是否是右值。比如，在上面的代码中，std::forward 怎么知道 param 被一个左值还是右值初始化？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，请参考条款 28。

std::move 和 std::forward 都可以归之为转换，唯一的不同之处是，std::move 总是在执行转换，而 std::forward 是在某些条件满足时才转。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转换代码，但是我希望我们在此达成共识：那样做非常恶心。

std::move 的魅力在于：方便，减少错误概率，而且非常简洁。举个栗子，有这样的一个类，我们想要跟踪，它的 move 构造函数调用了多少次，我们只需要一个 static 的计数器，在每次 move 构造时递增。假设还有一个 std::string 类型的非静态数据成员，下面是一个实

现移动构造(即使用 std::move)的常见例子：

```
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)
    : s(std::forward<std::string>(rhs.s)) //非常规，不可取的实现方式
    { ++moveCtorCalls; }
    ...

};
```

请注意：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)。其次，我们传递给 std::forward 是非引用类型，因为这是惯例，通过这个才能解码出实参是右值（请看条款 28）。综上，这就意味着 std::move 比 std::forward 少敲代码，也免去了传递类型参数的麻烦，同时也消除了传递错误类型的可能性(比如说，传一个 std::string&，会导致数据成员 s 被拷贝构造，而不是想要的移动构造)。

更重要的是，std::move 的使用表明了对右值的无条件的转换，而 std::forward 只对绑定了右值的引用进行转换。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，就是将一个对象传递给另外一个函数，同时保留此对象的左值或右值特性。所以我们需要这两个不同的函数来区分这两个操作。

需要铭记的要点
● std::move 执行一个无条件右值转换，本身不会 move 任何东东。 ● std::forward 只有实参是右值时，才执行右值转换。 ● std::move 和 std::forward 在运行时啥都不做。

条款 24：区分通用引用和右值引用

你应该听说过，真相可以让人感到自由，但是在恰当的场景下，适当的谎言也可以。本条款就是这样的一个“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&&。因此下面的假设看起来合理：如果你在代码中看到了"T&&"，那么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&&”有两个不同的含义。一个当然是右值引用，这样的引用表现起来和你预期一致：只绑定右值，它们基本含义就是表示对象可以移动。

“T&&”的另外一个含义是：要么是右值引用，要么是左值引用。这样的引用在代码中看起来像右值引用（即"T&&"），但是实际行为可能是左值引用（即"T&"）。它们的双重特性允许他们既可以绑定右值也可以绑定左值。此外，它们还可以绑定 const 或者 non-const，volatile 或者 non-volatile，甚至是 const volatile 对象。它们几乎可以绑定任何东西。这么牛叉的引用，我它们起个名字：通用引用（universal references）。

两种上下文中会出现通用引用。最普遍的一种是函数模板参数，就像上面代码那样：

```
template<typename T>
void f(T&& param); // param is a universal reference
```

第二种是 auto 声明，上面代码也写到了，

```
auto&& var2 = var1; // var2 is a universal reference
```

这两种上下文的共同点是：都存在类型推导。在模板 f 中，参数 param 的类型是推导出来的，在 var2 的声明中，var2 的类型也是推导出来的。和接下来的例子(也是上面例子的代码)对比，我们会发现，下面栗子是不存在类型推导的。所以，如果你看到"T&&"，却没有看到类型推导，那么你看到的就是一个右值引用：

```
void f(Widget&& param);       // 没有类型推导，param是右值引用
Widget&& var1 = Widget();     //没有类型推导，var1是右值引用
```

因为通用引用是引用，所以必须初始化。通用引用的初始化决定了它表示的是右值引用

还是左值引用。如果初始化是右值，那么通用引用对应的是右值引用；如果初始化是左值，那么通用引用对应的就是左值引用。对于身为函数参数的通用引用，初始化在调用侧提供：

```
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)
```

引用要成为通用引用，类型推导是必须的，但是还不够，引用声明的格式也很严格，声明必须正确，必须精确的"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 会被推导（除非调用者显式的指明类型，这种边界情况我们不予考虑）。但 param 声明的格式不是 T&&，而是 std::vector<T>&&。根据规则得出，它不是通用引用，而是右值引用。如果你传一个左值给 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
```

即使一个简单的 const 限定符，也可以把一个引用成为通用引用的可能抹杀：

```
template<typename T>
void f(const T&& param); // param is an rvalue reference
```

如果你在一个模板内部，看到了一个函数参数是 T&&类型，你可能就会假设它就是一个通用引用。但是你不能这样假设，因为在模板内部可不保证一定有类型推导。看下 std::vector 的 push_back 成员函数：

```
template<class T, class Allocator = allocator<T>> // from C++
class vector {                                    // Standards
public:
    void push_back(T&& x);

    ...

};
```

push_back 的参数确实是通用引用的格式 T&&，但却没有类型推导。因为 push_back 依赖于 vector 的实例化，而实例化的类型就完全决定了 push_back 的函数声明。也就是说，

```
std::vector<Widget> v;
```

将 vector 模板实例化如下：

```
class vector<Widget, allocator<Widget>> {
public:
    void push_back(Widget&& x); // rvalue reference
    ...
};
```

如你所见，push_back 没有用到类型推导。所以 vector<T>的 push_back（有两个重载的 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 函数确实用到了类型推导：

```
template<class T, class Allocator = allocator<T>> // still from
class vector {                                    // C++
public:                                           // Standards
    template <class... Args>
    void emplace_back(Args&&... args);
    ...
};
```

此处，类型 Args 独立于 vector 的类型参数 T ，所以每次调用 emplace_back 的时候，Args 就要推导一次。（实际上， Args 是一个参数包，并不是一个类型参数，但是为了讨论的方便，我们姑且当它是一个类型参数）。

我之前说通用引用的格式必须是 T&&，emplace_back 的类型参数名字命名为 Args，但这不影响 args 是一个通用引用，管它叫做 T 还是叫做 Args 呢，没啥区别。举个例子，下面的模板接受的参数就是通用引用，一是因为格式是"type&&"，二是因为 param 的类型会被推导(再一次提一下，除了调用者显式地指明了类型的那种边角情况)。

```
template<typename MyTemplateType>      // param is a
void someFunc(MyTemplateType&& param); // universal reference
```

我之前提到过 auto 变量可以是通用引用，更准确的说，声明为 auto&&的变量就是通用引用，因为存在类型推导并且格式为"T&&"。auto 通用引用并不像函数模板参数那么常见，在 C++ 11 中时不时会出现，到 C++ 14，出现的更加频繁，因为 C++ 14 中的 lambda 表达式允许声明 auto&&形参。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)(           // invoke func
            std::forward<decltype(params)>(params)... // on params
```

```
        );
    stop timer and record elapsed time;
};
```

如果你对于"std::forward<decltype(blah blah blah)>"的反应是“这是什么鬼!?”，那就意味着你还没看过条款 33，不要为此担心，本条款的重点是 auto&&形参。 func 是一个可以绑定到任何可调用对象的通用引用，可以是左值，也可以右值，args 是 0 个或多个通用引用，可以绑定到任意数量的任意类型对象。所以，多亏了 auto 通用引用，timeFuncInvocation 可以统计几乎任何函数的执行时间(any 和 pretty much any 的区别，请看条款 30)。

请记住，本条款作为通用引用的基础，其实是一个谎言，呃，或者说是一种"抽象(abstraction)"。底层的真相其实是条款 28 讲到的引用折叠（reference collapsing）。但是该事实并不会降低它的用途。右值引用和通用引用会帮助你更准确地阅读源码(“我现在看到的 T&&是只绑定右值，还是可以绑定任何值呢?”)，和同事沟通时避免歧义(“我这儿用的是通用引用，不是右值引用…”)，它也会使得你理解条款 25 和条款 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)跟爱因斯坦的广义相对论(truth)一样有用且更容易应用，通用引用的概念也跟引用折叠一样可取。

需要铭记的要点
● 如果一个函数模板形参具备 T&&格式且 T 类型需要推导，或者一个对象声明为 auto&&，那么这个形参或对象就是一个通用引用。 ● 如果类型声明的格式不完全是 type&&，或者不存在类型推导，那么 type&&表示的是一个右值引用。 ● 通用引用如果用右值初始化，就是右值引用，如果用左值初始化，就是左值引用。

条款 25：对右值引用使用 std::move，对通用引用使用 std::forward

右值引用仅绑定到可移动的对象。如果你有右值引用参数，你知道它绑定的对象可能被移走：

```
class Widget {
    Widget(Widget&& rhs); // rhs definitely refers to an
    …                     // object eligible for moving
};
```

在这种情况下，你希望给其他函数传递对象时，允许那些函数利用对象的右值。方法就是将绑定到这些对象的参数转换为右值。正如条款 23 所解释的那样，std::move 不仅仅是能干这个活，也是为此创造的：

```
class Widget {
public:
    Widget(Widget&& rhs)          // rhs is rvalue reference
    : name(std::move(rhs.name)),
    p(std::move(rhs.p))
    { … }
    …
private:
    std::string name;
    std::shared_ptr<SomeDataStructure> p;
};
```

另一方面，通用引用(见条款 24)可能绑定到可移动的对象。通用引用只有在以右值初始化时，才应该转换为右值。条款 23 说明这正是 std::forward 的用途：

```
class Widget {
public:
    template<typename T>
    void setName(T&& newName)             // newName is
    { name = std::forward<T>(newName); }  // universal reference
    …
};
```

简而言之，在转发给其他函数时，右值引用应该无条件地转换为右值(通过 std::move)，因为它们总是绑定到右值，而通用引用应该有条件地转换为右值(通过 std::forward)，因为它们只是有时绑定到右值。

条款 23 提到，对右值引用使用 std::forward 也是可以的，但源代码冗长、易出错并且不符合语言习惯，因此，应该避免对右值引用使用 std::forward。更糟糕的想法是对通用引用使用 std::move，因为存在意外修改左值(如局部变量)的问题：

```
class Widget {
public:
    template<typename T>
    void setName(T&& newName)          // universal reference
    { name = std::move(newName); }     // compiles, but is
    ...                                // bad, bad, bad!
private:
    std::string name;
    std::shared_ptr<SomeDataStructure> p;
};

std::string getWidgetName();       // factory function

Widget w;
auto n = getWidgetName();          // n is local variable
w.setName(n);                      // moves n into w!
...                                // n's value now unknown
```

此处，局部变量 n 传递给 w.setName，调用者会假设这个调用对 n 是只读操作，这个情有可原，但因为 setName 在内部使用 std::move 无条件地将其引用参数转换为一个右值， n 的值将被移到 w.name 中，从 setName 返回后，n 的值就变成未知的了，这种行为可能会让调用者感到绝望甚者发飙。

你可能会 setName 的参数不应该声明为通用引用，这种引用不能是 const(条款 24)，但是 setName 肯定不应该修改它的参数。你可能会指出，如果 setName 简单地重载 const 左值和右值，可能就避免了这个问题。像这样：

```
class Widget {
public:
    void setName(const std::string& newName)  // set from
    { name = newName; }                        // const lvalue
    void setName(std::string&& newName)        // set from
    { name = std::move(newName); }             // rvalue
    ...

};
```

这当然是可行的，但也有缺点。首先，要编写和维护更多的源码(两个函数而不是一个模板)；其次，效率可能会降低。例如，考虑 setName 的这种用法：

```
w.setName("Adela Novak");
```

对于通用引用版本的 setName，字符串“Adela Novak”将会直接传递给 std::string 的赋值运算符。因此，w.name 数据成员将直接从字符串赋值，不会出现临时 std::string 对象。不过，对于 setName 的重载版本，先要创建一个临时 std::string 对象，然后再 move 到 w 的数据成员，这个调用需要执行一次 std::string 构造(创建临时对象)，一次 std::string 移动赋值操作（将 newName 移到 w.name），以及一次 std::string 析构（销毁临时对象）。这样肯定比只执行一次 std::string 赋值要代价大得多。额外的代价可能因实现而有所不同，而这个代码是否值得担心也因应用和库而有所不同，但事实是，用左值引用和右值引用这一对重载来代替通用引用模板，在某些情况下，可能会带来额外的运行时开销。如果我们再往开了讲，Widget 的数据成员可能是任意的类型，性能差距可能会更大。

然而，重载左值和右值最严重的问题不是源代码的体谅或习惯，也不是代码的运行时性能，而是设计的可扩展性差。setName 只接受一个参数，因此只要重载两个就行了，但是对于更多参数的函数，每个参数都可以是左值或右值，重载的数量呈几何级增长：n 个参数需要 2n 个重载。这还不是最糟糕的。一些函数，实际就是函数模板，参数个数是无限制的。典型代表是 std::make_shared，还有 C++14 的 std::make_unique(参考条款 21)。看看最常用的声明：

```
template<class T, class... Args>              // from C++11
shared_ptr<T> make_shared(Args&&... args);    // Standard
template<class T, class... Args>              // from C++14
unique_ptr<T> make_unique(Args&&... args);    // Standard
```

对于这样的函数，无法重载左值和右值，只能选择通用引用。在这些函数中，传递参数给其他函数时，必须要用 std::forward。

嗯，通常，最终，但一开始不一定，在某些情况下，你会想要在一个函数中多次使用绑定到右值引用或通用引用的对象，你要确保在其他地方用完之前，这些对象不会被移走。这种情况下，你需要在最终使用引用的时候，才应用 std::move (对于右值)或 std::forward(对于通用引用)。例如：

```
template<typename T>                  // text is
void setSignText(T&& text)            // univ. reference
{
    sign.setText(text);               // use text, but
                                      // don't modify it
    auto now =                        // get current time
        std::chrono::system_clock::now();
```

```
    signHistory.add(now,
                    std::forward<T>(text)); // conditionally cast
}                                           // text to rvalue
```

这里，我们要确保 text 的值不会被 sign.setText 改变，因为我们调用 signhistory.add 时还有用到。所以仅在最后使用通用引用时才使用 std::forward。

对于 std::move，也是同样的思路，即最后使用右值引用时才应用 std::move，但是需要注意的是，在极少数情况下，你会希望使用 std::move_if_noexcept 而不是 std::move。要想知道什么时候和为什么，请阅读条款 14。

如果在一个按值返回的函数中，要返回一个绑定到右值引用或通用引用的对象，你应该会想到应用 std::move 或 std::forward。要了解原因，请考虑一个 operator+函数，用来将两个矩阵相加，已知左边的矩阵是一个右值(因此可以用来保存两个矩阵的和)：

```
Matrix                          // by-value return
operator+(Matrix&& lhs, const Matrix& rhs)
{
    lhs += rhs;
    return std::move(lhs);      // move lhs into
}                               // return value
```

return 语句中，通过将 lhs 转换为右值，lhs 将被移动到函数的返回值位置。如果省略了对 std::move 的调用，

```
Matrix                  // as above
operator+(Matrix&& lhs, const Matrix& rhs)
{
    lhs += rhs;
    return lhs;         // copy lhs into
}                       // return value
```

实际上，因为 lhs 是左值，这将迫使编译器将其复制到返回值的位置。假设矩阵类型支持 move 构造，即比拷贝构造更高效，使用 std::move 将生成更高效的代码。

如果 Matrix 不支持移动，则将其转换为一个右值也不会造成伤害，因为右值将被 Matrix 的拷贝构造函数简单地拷贝(参见条款 23)。如果 Matrix 后来修改为支持移动，operator+将在下一次编译的时候自动受益。既然如此，对于按值返回的函数，对要返回的右值引用使用 std::move，就没什么损失了(可能还会得到很多)。

对于通用引用和 std::forward 来说，情况也是类似的。考虑一个函数模板 reduceAndCopy，接受一个未约简的 Fraction 对象，先约简，然后返回其副本。如果原始对象是右值，它的值应该被移动到返回值中(从而避免了拷贝的代价)，但如果原始值是左值，则必须创建实际的

副本。因此：

```
template<typename T>
Fraction                            // by-value return
reduceAndCopy(T&& frac)             // universal reference param
{
    frac.reduce();
    return std::forward<T>(frac); // move rvalue into return
}                                    // value, copy lvalue
```

如果省略了 std::forward，frac 就会无条件的拷贝到返回值。

有些程序员如获至宝，并且还延用到一些不适用的场景中。“既然对右值引用参数使用 std::move 可以将拷贝变成移动”他们推理道，“我也可以在返回局部变量时，采用同样方式优化。”换句话说，他们认为，给定一个按值返回局部变量的函数，像下面这样，

```
Widget makeWidget()  // "Copying" version of makeWidget
{
    Widget w;        // local variable
    …                // configure w
    return w;        // "copy" w into return value
}
```

他们可以“优化”一下，将“拷贝”变成移动：

```
Widget makeWidget()          // Moving version of makeWidget
{
    Widget w;
    …
    return std::move(w);     // move w into return value
}                            // (don't do this!)
```

我用了引号，也就提醒你了，这条推理是有缺陷的。为什么呢？

因为标准化委员会早就意识到了这种优化。人们很久以前就认识到，makeWidget 的“拷贝”版本，通过在函数返回值的内存中构造局部变量 w，就可以避免拷贝。这称为返回值优化(return value optimization，RVO)，它在 C++标准发布时就得到了明确支持。

说出这种优化反倒是一件麻烦事，因为你只想在不影响软件行为的地方，省去那些拷贝。剖析一下标准中这个合法的 (可以说是有害的)优化，它表达的是，如果满足(1)局部对象的类型与返回值类型相同(2)返回的就是那个局部对象，编译器可能省去按值返回局部对象时的拷贝或移动。（备注：这种局部对象包括大多数的局部变量，如 makeWidget 中的 w，以及 return 语句中的临时对象。函数参数不属于这种。有些人对 RVO 还区分命名和未命名（即临时的）局部对象，未命名对象叫 RVO，命名对象就叫 NRVO (named return value optimization)）。

带着这些，再看一下 makeWidget 的“拷贝”版本：

```
Widget makeWidget() // "Copying" version of makeWidget
{
    Widget w;
    ...
    return w;        // "copy" w into return value
}
```

这儿上面的两个条件都满足，每个正常的 C++编译器都会使用 RVO 来避免拷贝 w。这意味着 makeWidget 的“拷贝”版本实际上不会拷贝任何东西。

makeWidget 的移动版本做的就像它的名字说的那样(假设 Widget 提供了移动构造函数)：将 w 的内容移动到 makeWidget 返回值的位置。但是为什么编译器不使用 RVO 来消除移动，依然在函数返回值的内存中构造 w 呢？答案很简单：他们不能。条件(2)规定当且仅当返回的是一个局部对象时，才执行 RVO，但移动版本 makeWidget 没有返回局部对象。再看一下它的 return 语句：

```
return std::move(w);
```

这里返回的不是本地对象 w，而是对 w 的引用，即 std::move(w)的结果。返回对本地对象的引用不满足 RVO 条件，所以编译器必须将 w 移动到函数返回值的位置。开发人员试图通过应用 std::move 来帮助编译器优化，实际上反而限制了优化！

但 RVO 的确是一种优化。编译器没有要求一定会省去拷贝和移动操作，即使允许那样做。也许你会怀疑，并且担心编译器会用拷贝操作惩罚你，仅仅因为它们可以做到。或者，也许你研究足够深入，会认识到存在某些情况，RVO 是很难实现的，例如，当一个函数中有不同的控制路径返回不同的局部变量。（编译器必须要在返回值内存中构造局部变量，但编译器怎么知道要构造哪个合适呢？）如果是这样，你可能更愿意执行移动而不是拷贝。也就是说，你可能还是会想到使用 std::move，因为那样就不存在拷贝了。

那种情况下，使用 std::move 依然是一个糟糕的想法。标准中 RVO 部分还提到，如果 RVO 条件满足，但编译器选择不省去拷贝，返回的对象必须是右值。实现上，标准要求，当 RVO 满足条件时，要么就选择省去拷贝，要么就隐性使用 std::move。所以在“拷贝”版本的 makeWidget 中，

```
Widget makeWidget() // as before
{
    Widget w;
    ...
```

```
    return w;
}
```

编译器要么就省去对 w 的拷贝，要么就把函数当做下面这样处理：

```
Widget makeWidget()
{
    Widget w;
    …
    return std::move(w); // treat w as rvalue, because
}                        // no copy elision was performed
```

函数传值参数的情况也类似。它们不适用于返回值省去拷贝的那种策略（因为参数的内存已经分配好，译者注），如果返回它们，编译器必须按右值处理。因此，如果源代码是这样的，

```
Widget makeWidget(Widget w)  // 传值参数，类型与返回值类型一样
{
    …
    return w;
}
```

编译器必须按以下方式处理：

```
Widget makeWidget(Widget w)
{
    …
    return std::move(w); // treat w as rvalue
}
```

这就意味着，如果你自己用了 std::move，不仅帮不了编译器，反而肯定阻止了优化。有些是应该对局部变量使用 std::move 的（即当给函数传递变量，并且你也知道这个变量以后用不着了），但是 return 语句中，因为 RVO，坚决不能使用 std::move，同样返回传值的参数也不能用 std::move。

需要铭记的要点
● 在最后一次使用的时候，对右值引用使用 std::move，对通用引用使用 std::forward。 ● 按值返回的函数，返回右值引用或通用引用时，也同样要分别应用 std::move 和 std::forward。 ● 在符合 RVO 的局部对象，坚决不能使用 std::move 或 std::forward。

条款 26：避免对通用引用重载

假设你需要写一个函数，接受一个 name 参数，记录当前日期和时间，并将 name 添加到一个全局的数据结构中。你可能写成下面这样：

```
std::multiset<std::string> names;     // global data structure

void logAndAdd(const std::string& name)
{
    auto now =                        // get current time
        std::chrono::system_clock::now();
    log(now, "logAndAdd");            // make log entry
    names.emplace(name);              // add name to global data structure;
}                                     // see Item 42 for info on emplace
```

也不能说这个代码不合理，但是这个代码不够高效。考虑下面三种调用：

```
std::string petName("Darla");
logAndAdd(petName);                         // pass lvalue std::string
logAndAdd(std::string("Persephone"));       // pass rvalue std::string
logAndAdd("Patty Dog");                     // pass string literal
```

第一个调用，logAndAdd 的形参 name 绑定到变量 petName，在 logAndAdd 内部，name 最终传给 names.emplace。因为 name 是左值，所以是拷贝进 names。没有任何方法可以避免那个拷贝，因为传进 logAndAdd 的是左值 petName。

第二个调用，形参 name 绑定到右值，即从"Persephone"显式创建的临时 std::string 对象，name 本身是左值，所以它还是拷贝进 names。但是我可以发现，原则上来说，它的值可以移动到 names。这个调用中，我们执行了一次拷贝，但实际上我们应该只要执行一次移动就可以了。

第三个调用，形参 name 还是绑定到右值，但这次是从"Patty Dog"隐式创建的临时 std::string 对象。跟第二个调用一样，name 也是拷贝进 names，但在这个调用中，原始实参是字符串。如果那个字符串直接传给 emplace，那么可以根本不用创建临时 std::string 对象。相反，emplace 直接在 std::multiset 内部用字符串创建 std::string 对象。那么在第三个调用，我们已经执行了一次拷贝，实际上也没有理由再去执行哪怕一次移动，更不要说拷贝了。

通过重写 logAndAdd，接受一个通用引用（参考条款 24），并且根据条款 25，std::forward 这个引用到 emplace，我们就可以消除第二个和第三个调用低效情况。如下所示：

```
template<typename T>
void logAndAdd(T&& name)
```

```
{
    auto now = std::chrono::system_clock::now();
    log(now, "logAndAdd");
    names.emplace(std::forward<T>(name));
}
std::string petName("Darla");          // as before
logAndAdd(petName);                    // as before, 拷贝左值到multiset
logAndAdd(std::string("Persephone")); // 移动右值而不是拷贝
logAndAdd("Patty Dog");                // 在multiset创建std::string
                                       // 而不是拷贝std::string
```

万岁，效率最优了！！

故事就此结束，我们可以就此打住，自豪地退休吗？但我还没告诉你，用户并不总是直接访问 logAndAdd 所需要的 names。有些用户只有一个索引，logAndAdd 使用索引查找表中对应的 name。要支持这样的用户，logAndAdd 要像下面这样重载：

```
std::string nameFromIdx(int idx); // return name corresponding to idx
void logAndAdd(int idx) // new overload
{
    auto now = std::chrono::system_clock::now();
    log(now, "logAndAdd");
    names.emplace(nameFromIdx(idx));
}
```

对这两个重载调用的抉择与预期一致：

```
std::string petName("Darla");          // as before
logAndAdd(petName);                    // as before, these
logAndAdd(std::string("Persephone")); // calls all invoke
logAndAdd("Patty Dog");                // the T&& overload
logAndAdd(22);                         // calls int overload
```

事实上，只有在你期望不太高的情况下，抉择才会达到预期的效果。假设用户用 short 存储索引并将其传递给 logAndAdd：

```
short nameIdx;
…                   // give nameIdx a value
logAndAdd(nameIdx);  // error!
```

最后一行的注释并不是很清晰，所以让我解释一下这里发生了什么。

有两个 logAndAdd 重载。接受通用引用的那个重载可以推导 T 为 short，因此可以精确匹配。带有 int 参数的重载只有在类型提升时才能与 short 匹配。根据正常的重载抉择规则，精确匹配优于需要类型提升的匹配，因此会调用通用引用重载。

在那个重载中，形参 name 绑定到传入的 short。然后，name 被 std::forward 到

names(std::multiset<std::string>)的成员函数 emplace，该函数依次将其转发到 std::string 构造函数。但 std::string 没有接受 short 的构造函数，所以 std::string 构造函数调用失败。都是因为通用引用比 int 更匹配 short 实参。

接受通用引用的函数是 C++中最贪吃的函数。它们实例化可以为几乎所有类型的实参创建出精确匹配。(条款 30 描述了一些类型的实参，情况与此不同。)这就是为什么将重载和通用引用结合在一起几乎总是一个坏主意：通用引用重载所匹配的参数类型比编写重载的开发人员通常期望的要多得多。

一个容易踩坑的情况是编写一个完美转发构造函数。对 logAndAdd 示例做个小修改来演示这个问题。与其编写一个接受 std::string 或用来查找 std::string 的索引的自由函数，不如想象一个 Person 类，它的构造函数做同样的事情：

```
class Person {
public:
    template<typename T>
    explicit Person(T&& n)              // perfect forwarding ctor;
    : name(std::forward<T>(n)) {}       // initializes data member

    explicit Person(int idx)            // int ctor
    : name(nameFromIdx(idx)) {}
    …
private:
    std::string name;
};
```

与 logAndAdd 的情况一样，传递一个非 int 类型的整数类型 (例如 std::size_t、short、long 等等)将调用通用引用构造函数重载，而不是 int 重载，并且将导致编译失败。然而，这里的问题更糟糕，因为 Person 还有更多的重载是眼睛看不到的。条款 17 提到，在适当条件下，C++将生成拷贝和移动构造函数，即使类包含一个模板化的构造函数，可以实例化它来生成拷贝或移动构造函数的签名。如果 Person 的拷贝和移动构造函数生成了， Person 实际上就会像下面这样：

```
class Person {
public:
    template<typename T>                // perfect forwarding ctor
    explicit Person(T&& n)
    : name(std::forward<T>(n)) {}


    explicit Person(int idx);           // int ctor
```

```
    Person(const Person& rhs);          // copy ctor (compiler-generated)
    Person(Person&& rhs);               // move ctor (compiler-generated)
    ...
};
```

只有当你花了大量时间在编译器和编译器-编写器上时，才会对这些有直觉，否则你已经忘记这个东东了：

```
Person p("Nancy");
auto cloneOfP(p); // create new Person from p; this won't compile!
```

在这里，我们试图从另一个 Person 创建一个 Person，看上去是拷贝构造的最明显的例子。(p 是一个 lvalue，所以我们可以排除任何关于通过移动操作完成“拷贝”的想法。)但是这段代码不会调用拷贝构造函数。它将调用完美转发构造函数。然后该函数将尝试使用 Person 对象(p)初始化 Person 的 std::string 数据成员。std::string 没有接受 Person 的构造函数，你的编译器会愤怒地举手，可能会用冗长而难以理解的错误消息来惩罚你，以表达他们的不满。

你可能会想，“为什么会调用完全转发构造函数而不是拷贝构造函数？我们正在用另一个 Person 初始化一个 Person 呀！”我们确实是想这样，但是编译器要坚决执行 C++规则，而这里的相关规则就是对重载函数的调用抉择规则。

编译器的理由如下。cloneOfP 使用 non-const 左值(p)来初始化，这意味着模板构造函数可以实例化成接受类型为 Person 的 non-const 左值。实例化之后，Person 类看起来是这样的：

```
class Person {
public:
    explicit Person(Person& n)              // instantiated from
    : name(std::forward<Person&>(n)) {}     // perfect-forwarding template
    explicit Person(int idx);               // as before
    Person(const Person& rhs);              // copy ctor(compiler-generated)
    ...
};
```

在语句，

```
auto cloneOfP(p);
```

可以将 p 传递给拷贝构造函数或实例化模板。调用拷贝构造函数需要向 p 添加 const 以匹配拷贝构造函数的参数类型，但是调用实例化的模板不需要这样的添加。因此，从模板生成的重载是更好的匹配，所以编译器按它们设计去做：生成对更好匹配函数的调用。因此，“拷贝”non-const 左值 Person 由完全转发构造函数处理，而不是由拷贝构造函数处理。

如果我们稍微改变一下例子，将拷贝的对象改为const，我们听到的是完全不同的调子：

```
const Person cp("Nancy"); // object is now const
auto cloneOfP(cp);        // calls copy constructor!
```

因为要拷贝的对象现在是 const，所以它与拷贝构造函数的参数完全匹配。模板化的构造函数可以实例化为相同签名，

```
class Person {
public:
    explicit Person(const Person& n); // instantiated from template
    Person(const Person& rhs);        // copy ctor (compiler-generated)
    ...
};
```

但这无关紧要，因为 C++中的重载抉择规则之一是在模板实例化和非模板函数(即，“普通”函数)都是同样好的匹配时，普通函数是首选。因此，拷贝构造函数(普通函数)胜过具有相同签名的实例化模板。

(如果你想知道为什么编译器可以实例化一个模板化的构造函数来获得拷贝构造函数可能具有的签名，请回看条款 17)。

当继承掺和进来时，完美转发构造函数和编译器生成的拷贝/移动操作之间的相互作用会更加波折。特别是，派生类拷贝和移动操作的传统实现表现得令人非常惊讶。这里，看看下面的情况：

```
class SpecialPerson: public Person {
public:
    SpecialPerson(const SpecialPerson& rhs)    // copy ctor; calls
    : Person(rhs)                              // base class
    { ... }                                    // forwarding ctor!

    SpecialPerson(SpecialPerson&& rhs)         // move ctor; calls
    : Person(std::move(rhs))                   // base class
    { ... }                                    // forwarding ctor!
};
```

正如注释所指出的，派生类 copy 和 move 构造函数不调用基类的 copy 和 move 构造函数，它们调用基类的完美转发构造函数！要理解原因，请注意派生类函数使用 SpecialPerson 类型的实参传递给基类，然后在类 Person 中，通过模板实例化和重载抉择得出构造函数。最终，代码将无法编译，因为 std::string 没有构造函数接受 SpecialPerson 类型。

我希望现在我已经说服你，如果可能的话，应该避免重载通用引用参数。但是，既然重

载通用引用是一个坏主意，那么如果你需要一个函数来转发大多数参数类型，也需要以一种特殊的方式处理某些参数类型，那么该怎么办呢？有很多方法来解决，我花了整整一个条款来介绍它们，就是条款 27，下一个条款。继续读，你会整明白的。

需要铭记的要点
● 重载通用引用几乎总是会导致通用引用的调用比预期的多。 ● 完美转发构造函数特别有问题，因为对于 non-const 左值，它们通常比拷贝构造更匹配，并且阻止派生类调用基类的拷贝和移动构造函数。

条款 27：熟悉对通用引用重载的可选方式

条款 26 解释了对通用引用的重载会导致各种各样的问题，包括独立函数和成员函数(特别是构造函数)。然而，它也给出了这样的重载可能有用的例子。要是它能像我们希望的那样就好了！本项目探索了实现所需行为的方法，既可以通过一些设计，这些设计避免对通用引用进行重载，也可以通过以某些方式来使用重载，这些方式会约束它们可以匹配的实参类型。

下面的讨论以条款 26 中介绍的例子为基础。如果你最近还没读过那个条款，你最好是先回顾一下。

放弃重载（Abandon overloading）

条款 26 中的第一个示例 logAndAdd 是许多函数的代表，这些函数可以通过简单地为可能的重载使用不同的名称，从而避免在通用引用上重载的缺点。例如，两个 logAndAdd 重载可以分为 logAndAddName 和 logAndAddNameIdx。可惜，这种方法不适用于我们考虑的第二个示例，即 Person 构造函数，因为构造函数名称是由语言固定的。此外，谁愿意放弃重载呢?

传 const 左值引用（Pass by const T&）

另一种方法是恢复到 C++ 98，并将传通用引用(pass-by-universal-reference)替换为传 const 左值引用(pass-by-lvalue-reference-to-const)。事实上，这是条款 26 考虑的第一个方法。缺点是这个设计没有我们希望的那么有效。我们现在对通用引用和重载之间的相互作用熟知之后，牺牲一些效率从而使事情简单化，可能比最初看起来更有吸引力。

传值（Pass by value）

在不增加复杂性的情况下，通常允许你改变行为的一种方法是，将传引用替换为传值。该设计遵循条款 41 的建议，当你知道要拷贝对象时，考虑按值传递对象，因此我将按照该条款详细讨论是怎么工作的以及效率如何。这里，我将展示如何在 Person 示例中使用该技术：

```
class Person {
public:
    explicit Person(std::string n)     // replaces T&& ctor; see
    : name(std::move(n)) {}            // Item 41 for use of std::move

    explicit Person(int idx)           // as before
    : name(nameFromIdx(idx)) {}
    …
private:
    std::string name;
};
```

因为 std::string 没有构造函数只接受整数，所以，所有以 int 和 int-like 实参(例如，std::size_t, short, long)构造 Person 的情况，都会传递给 int 重载。类似地，std::string 类型的所有参数(以及可以从中创建 std::string 的内容，例如“Ruth”之类的字符串)都传递给接受 std::string 的构造函数。因此，调用者不会感到意外。我想，你可能会争辩说，有些人可能会使用 0 或 NULL 来指示空指针，那样会调用 int 重载，但是这些人应该参考条款 8，并反复阅读，直到他们想明白，使用 0 或 NULL 标识空指针是种退步。

使用标签分派（Use Tag dispatch）

无论是传 lvalue-reference-to-const，还是传值，都不能提供对完美转发的支持。如果使用通用引用的动机是完美转发，我们就必须使用通用引用，没有别的选择了。然而，我们不想放弃重载。所以如果我们既不放弃重载也不放弃通用引用，我们如何避免对通用引用的重载呢？

其实没那么难。对重载函数的调用通过查看所有重载的所有形参以及调用侧的所有实参来解决，然后，考虑所有形参/实参组合，选择总体匹配最好的函数。通用引用形参通常可以精确匹配任何实参，但如果通用引用只是形参列表的一部分，同时还包含其他非通用引用的形参，这些非通用引用的形参如果匹配不上的话，也就不会选择此类重载。这就是标签分派方法背后的基础，举个栗子，会更容易理解。

我们在 logAndAdd 上应用标签分派，下面是刚才的示例代码，省得你再翻回去看：

```
std::multiset<std::string> names; // global data structure

template<typename T>      // make log entry and add
void logAndAdd(T&& name)  // name to data structure
```

```
{
    auto now = std::chrono::system_clock::now();
    log(now, "logAndAdd");
    names.emplace(std::forward<T>(name));
}
```

这个函数本身工作得很好，但是如果我们引入一个重载，它使用一个 int 来根据索引查找对象，我们就会碰到条款 26 提到的麻烦。本条款的目标是避免这种情况。我们将重新实现 logAndAdd，不是添加重载，而是委托给另外两个函数，一个用于整数值，另一个用于其他所有类型。logAndAdd 本身将接受所有参数类型，包括整型和非整型。

完成实际工作的两个函数将命名为 logAndAddImpl，即，我们将使用重载。其中一个函数将接受通用引用，所以我们会同时具备重载和通用引用。但是每个函数还将接受第二个参数，这个参数指示传递的参数是否为整数。第二个参数将防止我们掉进条款 26 中描述的泥潭，因为我们将通过第二个参数来决定选择哪个重载。

是的，我知道了，“Blah, blah, blah。别废话了，给我看看代码！”没问题。下面的一个几乎正确的 logAndAdd 升级版：

```
template<typename T>
void logAndAdd(T&& name)
{
    logAndAddImpl(std::forward<T>(name),
                  std::is_integral<T>()); // not quite correct
}
```

这个函数将它的参数转发给 logAndAddImpl，但它也传递一个参数，指出该参数的类型(T)是否为整型。至少，这是它期望实现的。对于右值整数实参，它也的确能做到。但是，正如条款 28 解释的，如果将左值实参传给通用引用 name，则 T 推导出的类型将是左值引用，因此，如果将 int 类型的左值传给 logAndAdd，T 被推导为 int&。这不是整数类型，因为引用不是整数类型。这意味着 std::is_integral<T>对于任何左值实参都是 false，即使该实参确实表示了一个整数值。

认识到这个问题就等于解决了它，因为标准 C++库有一个类型特性(见条款 9)，std::remove_reference，可以完成我的要求：从类型中删除任何引用限定符。因此，logAndAdd 的正确写法是：

```
template<typename T>
void logAndAdd(T&& name)
{
    logAndAddImpl(
```

```
            std::forward<T>(name),
            std::is_integral<typename std::remove_reference<T>::type>()
        );
    }
```

这就行了。(在 C++ 14 中，你可以使用 std::remove_reference_t<T>替换高亮显示的文本，从而少敲一些代码。详细信息请参见条款 9。)

这样，我们就可以将注意力转移到被调用的函数 logAndAddImpl 上。有两种重载，第一种仅适用于非整数类型(即，std::is_integral<typename std::remove_ref erence<T>::type>是 false 的类型)：

```
template<typename T>                                // non-integral
void logAndAddImpl(T&& name, std::false_type)       // argument:
{                                                   // add it to
    auto now = std::chrono::system_clock::now();    // global data
    log(now, "logAndAdd");                          // structure
    names.emplace(std::forward<T>(name));
}
```

一旦你理解了高亮参数背后的机制，这个代码就一目了然。从原理上讲，就是 logAndAdd 向 logAndAddImpl 传递一个 boolean，表明传给 logAndAdd 的实参是否是一个整数类型，但 true 和 false 是运行时值，我们需要使用重载解析(这是在编译时完成)来选择正确的 logAndAddImpl 重载。这意味着我们需要一个对应 true 的类型和另一个对应 false 的类型。这种需求非常普遍，标准库提供了 std::true_type 和 std::false_type。logAndAdd 传递给 logAndAddImpl 的实参是一个类型的对象，T 如果是整数，该类型就继承自 std::true_type，而 T 如果不是整数，该类型就继承自 std::false_type。最终结果是，只有当 T 不是整数类型时，上面的 logAndAddImpl 重载才是 logAndAdd 中调用的可选项。

第二个重载覆盖相反的情况：T 是整数类型。在这种情况下，logAndAddImpl 只需找到与传入索引对应的名称，并将该名称传回 logAndAdd：

```
std::string nameFromIdx(int idx);               // as in Item 26
void logAndAddImpl(int idx, std::true_type)     // integral
{                                               // argument: look
    logAndAdd(nameFromIdx(idx));                // up name and
}                                               // call logAndAdd
                                                // with it
```

通过再调用 logAndAdd，我们避免了同时在两个 logAndAddImpl 重载中写日志代码。

在这个设计中，std::true_type 和 std::false_type 类型是“标签”，它们的唯一目的就是强

制重载解析按照我们想要的方式进行。注意我们甚至没有给这些参数命名。它们在运行时没有任何作用，事实上，我们希望编译器能够认识到标签参数是不使用的，并将它们从程序的执行镜像中优化出来。(一些编译器会这么做，至少在某些时候是这样的。) 对 logAndAdd 中重载实现函数的调用，通过创建适当的标签对象，将工作“分派”到正确的重载。因此，此设计的名称为：标签分派。它是模板元编程的标准构建块，你越去深入了解现代 C++库的源码，你就越会经常遇到它。

对于我们的目的，标签分派的重要之处不在于它如何工作，而在于它如何允许我们，在避免条款 26 问题的情况下，组合使用通用引用和重载。分派函数 logAndAdd 接受一个不受约束的通用引用参数，但是这个函数没有重载，实现函数 logAndAddImpl 是重载的，其中一个函数接受一个通用引用参数，但是对这些函数的调用的解析不仅取决于通用引用参数，还取决于标签参数，并且标签值的设计保证可行匹配的重载不超过一个。因此，是标签决定了调用哪个重载，通用引用总是为其参数生成精确匹配，这一点无关紧要。

约束接受通用引用的模板

标记分派的关键是存在一个作为用户 API 的单一函数(未重载)。这个函数将需要完成的工作分派给实现函数。创建一个未重载的分派函数通常很容易，但是对于条款 26 的第二个问题，即 Person 类的完美转发构造函数，是一个例外。编译器本身可能会生成拷贝和移动构造函数，因此，即使只编写一个构造函数并在其中使用标签分派，一些构造也可能调用编译器生成的函数，而这些函数会绕过标签分派。

事实上，真正的问题不是上面这种绕过分派的情况，而是并不能总是调用正确。实际上，你总是希望类的拷贝构造函数处理拷贝该类型左值的请求，但是，正如条款 26 所演示的，如果提供了一个接受通用引用的构造函数，那么在拷贝非 const 左值时，就会调用这个通用引用构造函数，而不是拷贝构造函数。该条款还解释了，当基类声明一个完美转发构造函数时，通常会在派生类以传统方式实现其拷贝和移动构造函数时，调用该构造函数，尽管正确的行为是调用基类的拷贝和移动构造函数。

对于这样的情况，当一个重载的函数接受一个通用引用时，它比你想要的更贪婪，但又不够贪婪，不能充当单个分派函数时，标签分派不是你要使用的实现方式。你需要一种不同的技术，这种技术可以让你了解通用引用所在的函数模板允许使用的条件。我的朋友，你需要的是 std::enable_if。

enable_if 提供了一种方法，可以强制编译器表现出来好像不存在特定模板一样。这样的模板可以说是禁用的。默认情况下，所有模板都是启用的，但是使用 std::enable_if 的模板只有在满足 std::enable_if 指定的条件时才启用。在我们的例子中，我们希望只在传递的类型不是 Person 时启用完美构造构造函数。如果传递的类型是 Person，则需要禁用完美转发构造函数(即让编译器忽略它)，这样就会调用类的拷贝或移动构造函数来处理，这正是我们想要的，用另一个 Person 对象来初始化一个 Person 对象。

表达这一想法的方式并不特别困难，但是它的语法令人讨厌，特别是如果你以前从未见过它，所以，我将帮助你逐步理解它。关于 std::enable_if 的条件部分有一些样板，我们将从那些样板开始。下面是针对完美转发构造函数的声明，仅显示 std::enable_if 需要使用的部分。我只显示这个构造函数的声明，因为 std::enable_if 的使用对函数的实现没有影响，函数实现还是与条款 26 的一样。

```
class Person {
public:
    template<typename T,
             typename = typename std::enable_if<condition>::type>
    explicit Person(T&& n);
    ...
};
```

为了准确地理解高亮显示的文本干了什么，很遗憾，我建议你咨询其他来源，因为细节需要花不少时间来解释，而且这本书中篇幅不够。(在你研究的过程中，仔细研究 “SFINAE” 以及 std::enable_if，因为 SFINAE 是使 std::enable_if 工作的技术。)在这里，我想重点讨论控制是否启用此构造函数的条件表达式。

我们要指定的条件是，T 不是 Person，即，只有在 T 是 Person 以外的类型时，才应启用模板化构造函数。还好有个可以判断两个类型是否相同的类型特性(std::is_same)，看起来我们需要的条件就是!std::is_same<Person, T>::value。(注意这个短语开头的”!”，表明我们是希望 Person 与 T 不一样。)这与我们的需求非常接近，但并不完全正确，因为正如条款 28 所解释的，以左值初始化的通用引用，推导的类型总是左值引用。也就意味着，像下面这段代码，

```
Person p("Nancy");
auto cloneOfP(p); // initialize from lvalue
```

通用构造函数中的 T 类型将被推导为 Person&。显然，Person 和 Person&的类型不相

同，std::is_same 的结果是：std::is_same<Person, person&>::value 为 false。

如果我们更准确地思考，当我们说只有当 T 不是 Person 时才应该启用 Person 的模板化构造函数时，我们会意识到，当我们查看 T 时，我们想要忽略它

- 是否是一个引用。为了确定是否应该启用通用引用构造函数，Person、Person&和 Person&&的类型都与 Person 相同。
- 是否是 const 或 volatile。在我们看来，const Person 和 volatile Person 以及 const volatile Person 都跟 Person 相同。

这意味着在检查类型是否与 Person 相同之前，我们需要一种方法来从 T 中剥离任何引用、const 和 volatile。再一次，标准库为我们提供了相应的类型特性，就是 std::decay。std::decay<T>::type 与 T 相同，删除了引用和 cv 限定符(即，const 或 volatile 限定符)。(这里我是在胡说八道，因为 std::decay，顾名思义，也会将数组和函数类型转换为指针（见条款 1），但对于此处的讨论，std::decay 的行为正如我所描述的。)那么，我们想要控制构造函数是否启用的条件，就是

```
!std::is_same<Person, typename std::decay<T>::type>::value
```

即，忽略任何引用或 cv 限定符之后，Person 与 T 不是同一类型。(正如条款 9 所说，std::decay 前的“typename”是必需的，因为 std::decay<T>::type 依赖于模板参数 T。)

将此条件插入到上面的 std::enable_if 样板中，并对结果进行格式化，以便更容易地看到各个部分是如何组合在一起的，从而为 Person 的完美转发构造函数生成如下声明：

```
class Person {
public:
    template<
        typename T,
        typename = typename std::enable_if<
                        !std::is_same<Person,
                                      typename std::decay<T>::type
                                     >::value
                    >::type
    >
    explicit Person(T&& n);
    …

};
```

如果你以前从来没有见过这样的事情，感谢上帝。我把这个设计留到最后是有原因的。当你可以使用其他机制来避免混合通用引用和重载时(你几乎总是可以)，你应该采用那些机制。不过，一旦你习惯了函数语法和尖括号的扩展，就不会那么糟糕了。此外，这会给你带

来你一直想要的行为。根据上面的声明，从另一个 Person 构造 Person，无论是左值、右值、const 或 non-const、volatile 或 non-volatile 中的哪种，永远都不会调用通用引用构造。

成功了，对吗？我们做到了!!

嗯，还没有。先别忙着庆祝。条款 26 仍有一个未解决的问题，还在继续困扰着我们。我们得继续处理。

假设从 Person 派生的类以传统方式实现拷贝和移动操作：

```
class SpecialPerson: public Person {
public:
    SpecialPerson(const SpecialPerson& rhs)    // copy ctor; calls
    : Person(rhs)                              // base class
    { … }                                      // forwarding ctor!

    SpecialPerson(SpecialPerson&& rhs)         // move ctor; calls
    : Person(std::move(rhs))                   // base class
    { … }                                      // forwarding ctor!
    …

};
```

这与我在条款 26 中显示的代码相同，包括注释，唉，这些注释仍然是正确的。当我们拷贝或移动一个 SpecialPerson 对象时，我们希望使用基类的拷贝和移动构造函数来拷贝或移动它的基类部分，但在这些函数中，我们把 SpecialPerson 对象传递给基类的构造函数，因为 SpecialPerson 与 Person 不同(即使应用 std::decay，也还是不同)，基类的通用引用构造函数启用，它实例化来精确匹配 SpecialPerson 实参。这种精确匹配比拷贝和移动构造函数更好，所以对于我们现在的代码，拷贝和移动 SpecialPerson 对象将使用 Person 完美转发构造函数来拷贝或移动它们的基类部分！这就重复了条款 26 的问题。

派生类只是遵循实现派生类拷贝和移动构造函数的标准规则，所以解决这个问题的方法是在基类中，特别是在控制是否启用 Person 通用引用构造函数的条件。我们现在意识到，我们不希望为 Person 以外的任何参数类型启用模板化构造函数，我们希望为 Person 或 Person 派生类以外的任何参数类型启用它。好讨厌的继承！

如果你听说在标准类型特性中有一个特性判断一种类型是否派生自另一种类型，你不应该感到惊讶。它叫做 std::is_base_of。如果 T2 从 T1 派生，那么 std::is_base_of<T1, T2>::value 为 true。所有类型都可以认为是从自身派生出来的，因此 std::is_base_of<T, T>::value 为 true。这就很方便了，因为我们希望修改启用 Person 完美转发构造函数的条件为：类型 T，在删除引用和 cv 限定符之后，既不是 Person 也不是 Person 派生类。使用

std::is_base_of 代替 std::is_same 就可以得到我们需要的：

```
class Person {
public:
    template<
        typename T,
        typename = typename std::enable_if<
                        !std::is_base_of<Person,
                                         typename std::decay<T>::type
                                        >::value
                    >::type
    >
    explicit Person(T&& n);
    …

};
```

现在我们终于完成了。上面就是我们用 C++ 11 编写的代码。如果我们使用 C++ 14，这段代码仍然可以工作，但是我们可以为 std::enable_if 和 std::decay 使用别名模板，以摆脱“typename”和“::type”这两个累赘，从而生成更合适的代码：

```
class Person { // C++14
public:
  template<
    typename T,
    typename = std::enable_if_t<                  // less code here
                !std::is_base_of<Person,
                                 std::decay_t<T>  // and here
                                >::value
               >                                  // and here
  >
  explicit Person(T&& n);

  …

};
```

好吧，我承认：我撒谎了。我们仍然没做完。但我们正在逐步接近。

我们已经了解了如何使用 std::enable_if 选择性地禁用 Person 的通用引用构造函数，以便可以通过类的拷贝和移动构造函数来处理我们期望的实参类型，但是我们还没有看到如何来区分整型和非整型参数。毕竟，这是我们最初的目标，构造函数模糊性问题只是我们在这个过程中遇到的一些附带问题。

我们需要做的就是(1)添加一个 Person 构造函数重载来处理整型实参，(2)进一步约束模板化构造函数，以便禁用此类实参。将这些结合起来，就完成了：

```
class Person {
public:
    template<
      typename T,
      typename = std::enable_if_t<
          !std::is_base_of<Person, std::decay_t<T>>::value
          &&
          !std::is_integral<std::remove_reference_t<T>>::value
      >
    >
    explicit Person(T&& n)          // ctor for std::strings and
    : name(std::forward<T>(n))      // args convertible to
    { … }                           // std::strings

    explicit Person(int idx)        // ctor for integral args
    : name(nameFromIdx(idx))
    { … }
    …                               // copy and move ctors, etc.
private:
    std::string name;
};
```

瞧！多么漂亮！好吧，对于那些具有模板元编程癖好的人来说，这种方法的优点可能最为明显，但事实上还包括，这种方法不仅完成了工作，而且还以独特的沉着冷静完成了任务。因为它使用了完美转发，提供了最大的效率，而且控制了通用引用和重载的组合，而不是禁止这种组合，所以这种技术可以应用于无法避免重载的情况(例如构造函数)。

有得有失（Trade-offs）

本条款中考虑的前三种技术(放弃重载、传 const T&和传值)为函数的每个参数都指定类型。后两种技术（标记分派和约束模板的可使用性）使用完美转发，因此不为参数指定类型。这个基本决定（指定类型还是不指定）会产生一些后果。

一般来说，完美转发更有效，因为它避免仅为了符合参数声明的类型而创建临时对象。Person 构造函数这个例子中，在 Person 内部，完美转发允许将类似于"Nancy"这样的字符串转发给 std::string 构造函数，而不使用完美转发就必须创建一个临时 std::string 对象。

但是完美转发也有缺点。一种是某些类型实参不能完美转发，即使它们可以传递给具有特定类型的函数。条款 30 探讨了这些完美转发的失败场景。

第二个问题是，当用户传递无效参数时，错误消息的可理解性。例如，假设创建 Person

对象的用户传递由 char16_t (C++ 11 中引入的一种类型，用于表示 16 位字符)组成的字符串，而不是 char(std::string 由 char 组成)：

```
Person p(u"Konrad Zuse"); // "Konrad Zuse" consists of
                          // characters of type const char16_t
```

根据本条款的前三种方法，编译器看到可用的构造函数是接受 int 或 std::string，就会生成一条或多或少简单的错误消息，说明无法将 const char16_t[12]转换成 int 或 std::string。

然而，使用基于完美转发的方法，const char16_t 数组可以毫无怨言地绑定到构造函数的参数，并被转发到 Person 的 std::string 数据成员的构造函数，只有到此处，才会发现传入的类型(const char16_t 数组)和所需的类型(std::string 构造函数可以接受的任何类型)不匹配。由此产生的错误消息可能会令人印象深刻，我使用的一个编译器，错误信息超过 160 行。

在这个例子中，通用引用只转发一次(从 Person 的构造函数到 std:: string 构造函数)，但系统越复杂，就越有可能通过好几层通用引用转发，最后到达一个调用，确定参数类型是否可以接受。转发通用引用的次数越多，出错时的错误消息就越令人困惑。许多开发人员发现，仅这个问题就有足够理由考虑，那些最关心性能的接口，选择通用引用参数时应该有所保留。

在 Person 例子中，我们知道转发函数的通用引用参数应该是对 std::string 的初始化，所以我们可以使用 static_assert 来验证它是否可以起那个作用。is_constructible 这个类型特性执行编译时检查，以确定是否可以从一个不同类型(或类型集)的对象(或对象集)构造另一个类型的对象，因此断言很容易编写：

```
class Person {
public:
  template<                                          // as before
    typename T,
    typename = std::enable_if_t<
      !std::is_base_of<Person, std::decay_t<T>>::value
      &&
      !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n)
  : name(std::forward<T>(n))
  {
    // assert that a std::string can be created from a T object
    static_assert(
      std::is_constructible<std::string, T>::value,
```

```
            "Parameter n can't be used to construct a std::string"
        );
        ...             // the usual ctor work goes here
    }
    ...                 // remainder of Person class (as before)
};
```

如果用户代码试图从某个类型创建 Person，而该类型又不能用于构造 std::string，则会生成指定的错误消息。不幸的是，在本例中，static_assert 位于构造函数的主体中，但是作为成员初始化列表一部分的转发代码位于其前面。基于我使用的编译器，得到的结果是，static_assert 产生的漂亮的、可读的消息，只在通常的错误消息(多达 160+错误消息)之后，就立马出现了。

需要铭记的要点
● 通用引用和重载组合的可选方法包括：区分函数名，传 lvalue-reference- to-const，传值，以及使用标签分派。 ● 通过 std::enable_if 约束模板，允许同时使用通用引用和重载，但是它控制了编译器可以使用通用引用重载的条件。 ● 通用引用参数一般具有效率优势，但它们也通常具有可用性劣势。

条款 28：理解引用折叠

条款 23 提到，当实参传递给模板函数时，推导出的模板形参类型包含了实参是左值还是右值这一信息。条款没有提到只有通用引用参数才是这种情况，但它有理由去忽略这一情况，因为直到条款 24 才引入通用引用。综合起来看，对于下面的模板，

```
template<typename T>
void func(T&& param);
```

模板参数 T 推导出的类型会包含传给 param 的实参是左值还是右值这一信息。

这个机制很简单。当实参是左值时，T 被推导为左值引用，当传右值时，T 被推导为非引用类型。(注意不对称性：左值编码为左值引用，但是右值编码为非引用)。因此：

```
Widget widgetFactory();       // 返回右值
Widget w;                     // 左值

func(w);                      // 用左值调用，T推导为Widget&
func(widgetFactory());        // 用右值调用，T推导为Widget
```

在连个 func 调用中，传的都是 Widget，不过因为一个是左值，一个是右值，所以模板参数 T 推导出来的类型是不同的。这个，也是我们即将看到的，是什么决定了通用引用是成为右值引用还是左值引用，它也是 std::forward 工作的底层机制。

在我们进一步了解 std::forward 和通用引用之前，我们必须知道，在 C++里面，引用的引用是非法的。如果你想那样声明，编译器绝不答应：

```
int x;
…
auto& & rx = x; // error! can't declare reference to reference
```

但是考虑一下，如果给通用引用传个左值，会发生什么呢：

```
template<typename T>
void func(T&& param);     // as before

func(w);                  // invoke func with lvalue; T deduced as Widget&
```

如果我们用 T 推导的类型，即 Widget&，来实例化模板，结果如下：

```
void func(Widget& && param);
```

这是引用的引用!! 并且编译器肯定不允许。从条款 24 我们知道，因为通用引用 param 用左值来初始化，所以 param 的类型会是左值引用，编译器是如何将 T 的推导类型代入模板，从而得到下面最终的函数签名？

```
void func(Widget& param);
```

答案是引用折叠（reference collapsing）。是的，你自己的确是不允许声明引用的引用，但编译器可以在特定的上下文中生成，模板实例化就是其中之一。当编译器生成引用的引用时，就会触发引用折叠。

因为有两类引用（lvalue 和 rvalue），所以，存在四种引用-引用组合（lvalue to lvalue, lvalue to rvalue, rvalue to lvalue,和 rvalue to rvalue）。如果在允许的上下文（如模板实例化）中出现了引用的引用，引用折叠按如下规则处理：

如果两个引用中任何一个是 lvalue 引用，那么结果就是一个 lvalue 引用。否则(即，两个都是右值引用)结果就是一个 rvalue 引用。

在我们上面的例子中，将推导出的类型 Widget&代入模板函数 func，得到类型是左值引用的右值引用，而根据引用折叠规则，结果就是左值引用。

引用折叠是 std::forward 工作的核心部分。就像条款 25 解释的，std::forward 要应用到通用引用，所以一个常用的用例就像下面这样：

```
template<typename T>
void f(T&& fParam)
{
    ...                                     // do some work

    someFunc(std::forward<T>(fParam));      // forward fParam to someFunc
}
```

因为 fParam 是一个通用引用，所以类型参数 T 将包含实参到底是左值还是右值这个信息。std::forward 的任务就是将 fParam（一个左值）转换为一个右值，当且仅当实参是一个右值，即 T 是非引用类型。

下面是 std::forward 的实现：

```
template<typename T>                                // in
T&& forward(typename                                // namespace
            remove_reference<T>::type& param)       // std
{
    return static_cast<T&&>(param);
}
```

这并不完全符合标准(我省略了一些接口细节)，但是对于理解 std::forward 的行为，差异是无关紧要的。

假设传给 f 的实参是一个 Widget 左值，T 将推导为 Widget&，而 std::forward 将实例化为 std::forward<Widget&>，将 Widget&插入 std::forward 的实现，结果如下：

```
Widget& && forward(typename
                   remove_reference<Widget&>::type& param)
{ return static_cast<Widget& &&>(param); }
```

类型特性 std::remove_reference<Widget&>::type 得到结果是 Widget（见条款 9），所以 std::forward 变成：

```
Widget& && forward(Widget& param)
{ return static_cast<Widget& &&>(param); }
```

引用折叠作用于返回类型和 static_cast，最终的 std::forward 版本如下：

```
Widget& forward(Widget& param)                // still in
{ return static_cast<Widget&>(param); }       // namespace std
```

正如你所见，当给 f 传递一个左值实参时，std::forward 实例化为接受并返回一个左值引用。std::forward 内部的 static_cast 不做任何事情，因为 param 的类型本身就已经是 Widget&，再转换成 Widget&就没有影响。因此，传给 std::forward 的左值实参就返回一个左值引用。根据定义，左值引用本身是左值，所以，给 std::forward 传左值，返回的也是左值，就跟预期的一样。

现在，假设传给 f 的是一个 Widget 右值。在这个例子中，T 将推导为 Widget，因此，f 内部的 std::forward 将变成 std::forward<Widget>。std::forward 的实现就变成下面这样：

```
Widget&& forward(typename
                 remove_reference<Widget>::type& param)
{ return static_cast<Widget&&>(param); }
```

对非引用类型 Widget 应用 remove_reference，得到结果还是 Widget，所以 std::forward 的实现变成这样：

```
Widget&& forward(Widget& param)
{ return static_cast<Widget&&>(param); }
```

此处没有引用的引用，所以不存在引用折叠，并且这就是最终实例化的版本。

从函数返回的右值引用被定义为右值，所以，在这个例子中，std::forward 将 f 的参数 fParam（一个左值）转换成一个右值。最终的结果就是，传给 f 的右值实参将会按预期的那样，以右值转发给 someFunc。

在 C++14 中，std::remove_reference_t 可以使 std::forward 的实现更简洁：

```
template<typename T>                          // C++14; still in
T&& forward(remove_reference_t<T>& param) // namespace std
{
    return static_cast<T&&>(param);
```

```
}
```

引用折叠在四种上下文中可能发生。第一种也是最普遍的，就是模板实例化；第二种是 auto 变量的类型生成，情况跟模板基本相同，因为 auto 类型推导与模板类型推导基本相同（见条款 2）。再看一下本条款早期的例子：

```
template<typename T>
void func(T&& param);

Widget widgetFactory();  // function returning rvalue

Widget w;                // a variable (an lvalue)
func(w);                 // call func with lvalue; T deduced to be Widget&
func(widgetFactory());   // call func with rvalue; T deduced to be Widget
```

auto 可以模仿一下，如声明

```
auto&& w1 = w;
```

用左值初始化 w1，因此推导出 auto 类型为 Widget&，代入到声明中，就生成引用的引用：

```
Widget& && w1 = w;
```

引用折叠以后，就变成：

```
Widget& w1 = w;
```

因此，w1 就是一个左值引用。

另一方面，下面这个声明

```
auto&& w2 = widgetFactory();
```

用右值初始化 w2，推导出 auto 类型为 Widget，代入之后，得到：

```
Widget&& w2 = widgetFactory();
```

此处没有引用的引用，所以我们得到最终结果，w2 是一个右值引用。

现在我们才算真正理解条款 24 引入的通用引用。通用引用不是一个新的引用类型，它实际上是一个右值引用，但它的上下文要满足下面两个条件：

- **类型推导区分出左值和右值**。T 类型的左值推导出类型为 T&，而右值推导出类型就是 T。
- **产生引用折叠**。

通用引用这个概念非常有用，因为它让你不必去识别引用折叠上下文，然后在心里为左值和右值推导不同类型，并将推导出的类型代入到上下文中，再应用引用折叠规则算出最终

结果。

前面我提到有四种上下文，现在已经讲了模板实例化和 auto 类型生成这两种。第三种是 typedef 和别名声明（见条款 9）的生成与使用。如果在创建或评估 typedef 期间，出现了引用的引用，那么引用折叠就会发生。取个例子，假设我们有一个 Widget 类模板，内嵌一个右值引用类型的 typedef：

```
template<typename T>
class Widget {
public:
    typedef T&& RvalueRefToT;
    ...
};
```

并且假设我们用一个左值引用来实例化：

```
Widget<int&> w;
```

将 int&代入进模板，我们得到下面的 typedef：

```
typedef int& && RvalueRefToT;
```

引用折叠将其简化为：

```
typedef int& RvalueRefToT;
```

这个会导致 typedef 出来的类型不是我们想象的那样：当 Widget 用左值引用初始化时，RvalueRefToT 变成了左值引用的 typedef。

最后一种产生引用折叠的上下文是 decltype 的使用。如果在分析牵扯到 decltype 的类型期间，出现了引用的引用，也会产生引用折叠。（decltype 的信息参考条款 3）

需要铭记的要点
● 产生引用折叠的上下文有四种：模板实例化、auto 类型生成、typedef 和别名声明的生成与使用、以及 decltype。 ● 但编译器在引用折叠上下文中生成一个引用的引用时，结果会变成单个引用。如果任何一个原始引用是左值引用，那么结果就是左值引用，否则就是右值引用。 ● 通用引用是特殊上下文的右值引用，这个上下文满足两个条件：类型推导区分出左值和右值，并且发生引用折叠。

条款 29：假定 move 操作未提供、不廉价、不可使用

移动语义可以说是 C++ 11 的首要特性。“现在是移动容器跟拷贝指针一样廉价！”你可能会听到，“拷贝临时对象现在是如此的高效，避免这种拷贝的编码就等于过早的优化！（拷贝临时对象会通过移动语义在编译时优化，可以大胆使用临时拷贝——译者注）”这样情绪很容易理解。移动语义确实是一个重要的特性。它不只是允许编译器用相对廉价的移动来代替昂贵的拷贝操作，实际上也要求编译器这样做(当满足合适的条件时)。

对于 C++ 98 的 code base，使用符合 C++ 11 的编译器重新编译，突然的，你的软件运行更快了。

Move 语义可以真正做到这一点，这赋予了这个特性一个传奇光环。然而，传说通常是夸张的结果，本条款的目的就是要让你的期望有据可依。

让我们从那些不支持移动语义的类型开始。C++11 对整个 C++ 98 标准库进行了大修，对于可以实现移动的类型，添加了 move 操作，并且这些组件库可以充分利用这些操作，但是，你使用的 code base 很可能还未完全改版来充分利用 C++ 11。在你的应用程序或使用的库中，对于那些还未修改适应 C++11 的类型，编译器支持的 move 操作可能没给你带来啥好处。事实上，C++11 愿意为那些缺少 move 操作的类来生成 move 操作，但必须是在没有声明拷贝操作、移动操作或析构函数的情况下（见条款 17）。如果类型存在已经禁用移动操作（例如，通过 delete 移动操作，见条款 11）的数据成员或基类，也会阻止编译器为这些类型生成移动操作。对于没有显式支持移动且不符合编译器生成移动操作条件的类型，没有理由期望 C++ 11 会提供超越 C++ 98 的任何性能改进。

即使是显式支持移动的类型，性能也不可能提升到你希望的那么多。例如，C++ 11 标准库中的所有容器都支持移动，但如果你认为所有容器的移动都是廉价的，那就大错特错了。对于一些容器，是因为没有真正廉价的方式来移动他们的内容；对于其他容器，那是因为容器提供的真正廉价的移动操作，还附带了一些不能满足移动的容器元素说明。

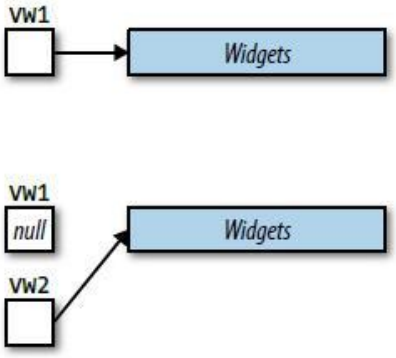
考虑看看 C++11 的新容器 std::array。std::array 本质上是一个具有 STL 接口的内建数组，它与其他标准容器完全不同，其他容器都是将其内容存储在堆上，这样的容器从概念上讲，只持有对象的指针。（现实是比较复杂的，但是为了本次分析的目的，差异并不重要。）这样就可能在常量时间内移动整个容器的内容：只需从源容器拷贝内容的指针到目标容器，并且将源的指针置为 null：

```
std::vector<Widget> vw1;

// put data into vw1

…

// move vw1 into vw2. Runs in
// constant time. Only ptrs
// in vw1 and vw2 are modified
auto vw2 = std::move(vw1);
```



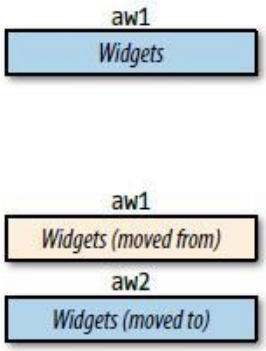
std::array 缺少这样的指针，因为 std::array 内容的数据直接存储在它的对象中：

```
std::array<Widget, 10000> aw1;

// put data into aw1

…

// move aw1 into aw2. Runs in
// linear time. All elements in
// aw1 are moved into aw2
auto aw2 = std::move(aw1);
```



注意，aw1 中的元素被移动到 aw2 中。假设 Widget 是一个移动比拷贝快的类型，移动 Widget 的一个 std::array 会比拷贝快的多。所以 std::array 当然是支持移动的。然而，移动和拷贝 std::array 都具有线性时间计算复杂度，因为容器中的每个元素都必须被拷贝或移动。这跟有时听到的说法， “移动容器现在和赋值一对指针一样廉价”，相距甚远。

另一方面，std::string 提供了常量时间的移动和线性时间的拷贝。这听起来好像移动比拷贝快，但事实可能并非如此。许多字符串实现都使用了小字符串优化（small string optimization）(SSO)。有了 SSO，“小”字符串(例如，容量不超过 15 个字符的字符串)是存储在 std::string 对象内部的缓冲区中，没有使用堆分配的存储。移动基于 SSO 实现的小字符串并不比拷贝快，因为移动比拷贝性能更优的基础是只拷贝一个指针（copy-only-a-pointer），而此处这个技巧不适用。

SSO 的动机是大量证据表明短字符串是许多应用程序常用的。使用内部缓冲区存储此类字符串的内容，可以省去动态分配内存，而这通常是一种效率提升。然而，这种提升的后果就是，移动的速度不会超过拷贝，尽管如此，你也可以乐观地说，对于这样的字符串，拷

贝并不比移动慢。

甚至对于支持快速移动操作的类型，一些看起来是确定无疑的移动情况可能会执行拷贝。条款 14 解释到，在标准库中，一些容器操作提供了强异常安全保证，并且为了确保依赖于该保证的 C++ 98 遗留代码，不会在升级到 C++ 11 的时候被破坏，可能只有在明确知道移动操作不抛出异常时，底层的拷贝操作才会替换成移动操作。这样的后果就是，即使一种类型提供了比拷贝操作更有效的移动操作，并且在代码中的某个特定点上，移动操作通常是合适的(例如，源对象是一个右值)，由于相应的移动操作没有声明为 noexcept，编译器仍然可能被迫调用拷贝操作。

因此，存在几个场景，C++ 11 的移动语义没给你带来好处：

- **没有移动操作**：要移动的对象未提供移动操作，因此移动变成了拷贝。
- **移动操作不够快**：要移动的对象具备移动操作，但并不比拷贝快。
- **移动操作不可用**：在可以移动的上下文中，要求移动操作不抛出异常，但移动操作却未声明为 noexcept。

值得一提的是，在另一个场景中，move 语义不会提供任何效率提升：

- **源对象是左值**：除极少数情况外(见条款 25)，只有右值才能作为移动操作的源。

但是本条款的标题是假设移动操作未提供、不廉价、且不可使用。这是一般代码中的典型情况，例如，在编写模板时，因为你不知道你工作的所有类型，在这种情况下，在移动语义存在之前，你在拷贝对象时必须像 C++ 98 一样保守。这也是“不稳定（unstable）”代码的情况，即，类型特性会频繁修改的代码。

不过，你通常会知道代码使用的类型，并且可以依赖它们的不变特征 (例如，是否支持廉价的移动操作)。在这种情况下，你不需要做假设。你可以简单地寻找你所用类型的 move 支持的详细信息。如果这些类型提供了廉价的移动操作，并且你使用的对象在需要调用移动操作的上下文中，你可以安全地依赖 move 语义，用移动替换拷贝。

需要铭记的要点
● 假定 move 操作未提供、不廉价、不可使用。 ● 在已知类型或支持 move 语义的代码中，就不需要这样的假设。

条款 30：熟悉完美转发失败的情形

C++ 11 中最突出的特性之一是完美转发。完美转发。它是完美的！唉，深扒进去，你会发现，现实不如理想那么“完美”。C++ 11 的完美转发非常好，但只有在你愿意忽略一两个例外的前提下。本条款就是为了带你熟悉那么例外情况。

在我们开始探索这些例外之前，有必要回顾一下“完美转发”的含义。转发只是一个函数将它的参数向前传递给另一个函数。第二个函数的目标是接收到的对象与第一个函数接收到的相同。这就排除了传值参数，因为它们是调用者传入的原始内容的副本。我们希望转发的函数能够使用原始传入的对象。指针参数也被排除在外，因为我们不想强迫调用者一定要传指针。当面临的是通用目的转发时，我们将要处理的是引用类型的参数。

完美转发意味着我们不仅转发对象，还转发它们的显著特征：它们的类型是左值还是右值，并且是 const 还是 volatile。结合刚才提到的，我们将要处理是引用参数，这意味着我们将使用通用引用(见条款 24)，因为只有通用引用参数才包含实参的左值和右值信息。

假设我们有一个函数 f，并且我们想写一个转发到 f 的函数(实际上是函数模板)。我们需要的核心代码是这样的：

```
template<typename T>
void fwd(T&& param)               // accept any argument
{
    f(std::forward<T>(param));    // forward it to f
}
```

转发函数本质上是通用的。例如，fwd 模板，接受任何类型的实参，然后转发。这个函数可以扩展成可变模板，从而接受任意数量的参数。fwd 的可变格式如下：

```
template<typename... Ts>
void fwd(Ts&&... params)              // accept any arguments
{
    f(std::forward<Ts>(params)...);   // forward them to f
}
```

这就是你会在其他地方看到的格式，如标准容器的 emplace 函数(见条款 42)以及智能指针工厂函数 std::make_shared 和 std::make_unique(见条款 21)。

给定目标函数 f 和转发函数 fwd，完美转发失败是指，如果用一个特定的实参调用 f 做一件事，但是用同样的实参调用 fwd 却做了不同的事情：

```
f( expression );     // if this does one thing,
```

```
fwd( expression );    // but this does something else, fwd fails
                      // to perfectly forward expression to f
```

有好几种类型的实参会导致这种失败。重要的是要知道有哪些并且是怎么工作的，所以让我们来看看那些不能完美转发的实参。

花括号初始化（Braced initializers）

假设 f 声明如下：

```
void f(const std::vector<int>& v);
```

用花括号初始化来调用 f，是可以编译通过的，

```
f({ 1, 2, 3 }); // fine, "{1, 2, 3}" 隐式转换成 std::vector<int>
```

但是同样调用 fwd，却无法编译：

```
fwd({ 1, 2, 3 }); // error! doesn't compile
```

那是因为使用花括号初始化是一个完美转发失败的情形。

所有这样的失败情形都有相同的原因。直接调用 f（如 f({1,2，3})），编译器会比较调用侧的实参和函数的形参，看它们是否兼容，如果可以兼容，就通过隐式转换来完成调用。在上面的例子中，它们从{1,2,3}生成一个临时的 std::vector<int>对象，以便绑定到 f 的形参 v。

当通过转发函数模板 fwd 间接调用 f 时，编译器不再比较实参和 fwd 的形参，相反，它们推导出传递给 fwd 的实参类型，并且与 f 的形参声明比较。当下面任一情况发生时，完美转发失败：

编译器无法为一个或多个 fwd 形参推导出类型。这种情况下，代码无法编译。

编译器为一个或多个 fwd 形参推导出“错误”的类型。此处，“错误”可能意味着 fwd 的实例化将无法使用推导出的类型进行编译，但这也可能意味着使用 fwd 推导的类型调用 f 与直接调用 f 的行为不同。这种差异行为的一种来源是，如果 f 是重载的函数名，并且由于“不正确”的类型推导， fwd 内部调用的重载不同于直接调用的重载。

在上面的“fwd({ 1, 2, 3 })”调用中，给未声明为 std::initializer_list 的函数模板传递一个花括号初始化，就像标准说的，这是一个“无法推导的上下文（non-deduced context）”。直白的说，就是编译器禁止从{ 1, 2, 3 }表达式推导类型，因为 fwd 的形参没有声明为 std::initializer_list。fwd 形参的类型推导被阻止了，编译器也就理所当然的拒绝这种调用了。

有趣的是，条款 2 解释了，用花括号初始化的 auto 变量是可以推导出类型的。这些变量被认为是 std::initializer_list 对象，这为上面这种情形提供了一个简单的变通方案，即，使

用 auto 声明一个局部变量，然后将局部变量传递给转发函数：

```
auto il = { 1, 2, 3 }; // il的类型推导为 std::initializer_list<int>
fwd(il);               // fine, perfect-forwards il to f
```

0 或者 NULL 作为空指针（0 or NULL as null pointers）

条款 8 解释了，当你试图将 0 或 NULL 作为空指针传递给模板时，类型推导会偏了十万八千里，推导出的是整数类型(通常是 int)而不是指针类型。结果是 0 和 NULL 都不能作为为空指针进行完全转发。但是，解决方法很简单：传 nullptr 而不是 0 或 NULL。详情请参阅条款 8。

仅声明的完整 static const 数据成员（Declaration-only integral static const data members）

一般来说，对于完整的 static const 数据成员，不需要在类中再定义了，有声明就足够了。这是因为编译器会对这些成员的值执行常量替换，因此也就不必为它们分配内存了。例如，考虑以下代码：

```
class Widget {
public:
    static const std::size_t MinVals = 28;    // MinVals' declaration
    ...
};
...                                           // no defn. for MinVals

std::vector<int> widgetData;
widgetData.reserve(Widget::MinVals);          // use of MinVals
```

这里，我们使用 Widget::MinVals(此后简称为 MinVals)指定 widgetData 的初始容量，即使 MinVals 缺少一个定义。编译器的工作原理在用到 MinVals 的地方，全面用 28 代替。事实上，不为 MinVals 开辟存储空间，也没有问题。如要取 MinVals 的地址(例如，如果有人创建了一个指向 MinVals 的指针)，那么 MinVals 将需要存储(这样指针才有东西可以指)，而上面的代码，尽管可以编译通过，但链接会失败，除非 MinVals 提供了定义。

带着这个概念，假设 f 声明如下：

```
void f(std::size_t val);
```

用 MinVals 调用 f 是没问题的，因为编译器仅仅是将 MinVals 替换为它的实际值：

```
f(Widget::MinVals); // fine, treated as "f(28)"
```

但如果通过 fwd 来调用 f 的话，就没那么顺畅了：

```
fwd(Widget::MinVals); // error! shouldn't link
```

这段代码编译可以通过，但链接不应该通过。如果这让你想起，如果我们编写了使用 MinVals 地址的代码，发生的事情，那就很好，因为这个底层问题是相同的。

尽管没有代码去取 MinVals 的地址，但是 fwd 的参数是通用引用，而在编译器生成的代码中，引用通常是当作指针处理的。在程序的底层二进制代码中(和硬件上)，指针和引用本质上是相同的。在这个层面上，引用可以简单理解为自动解引用的指针。在这种情况下，通过引用传 MinVals 实际上与传指针一样，而指针必须有内存来指向。因此，通过引用传递完整的 static const 数据成员，通常就需要它们被定义，也正是这个问题，导致完美转发失败。

但是，也许你也注意到了我在前面的讨论中提到的那些含糊其辞的话。代码“不应该”链接。引用“通常”被当作指针对待。通过引用传递完整的 static const 数据成员“一般”要求它们被定义。就好像我知道一些我不想告诉你的事……

那是因为我的确知道。根据标准，通过引用传递 MinVals 需要定义它。但是并不是所有的实现都有这个强制要求。所以，根据你的编译器和链接器，你可能会发现，你可以完美转发未定义的完整 static const 数据成员。如果你做到了，恭喜你，但是没有理由期望这样的代码是可移植的。要使得代码可移植，就简单地提供一个定义。对于 MinVals，看起来是这样的：

```
const std::size_t Widget::MinVals; // in Widget's .cpp file
```

请注意，该定义没有重复初始化(在 MinVals 例子中是 28)。不过，不要为这个细节而紧张。如果你忘记并在两个地方都提供了初始化，编译器会提醒你只指定一次即可。

重载的函数名和模板名（Overloaded function names and template names）

假设我们的函数 f 可以通过传递一个函数来定制其行为。假设这个函数接受并返回 int，f 可以这样声明：

```
void f(int (*pf)(int)); // pf = "processing function"
```

值得注意的是，f 也可以使用更简单的非指针语法声明。这样的声明看起来是这样的，

尽管它的含义与上面一样：

```
void f(int pf(int)); // declares same f as above
```

不管哪种方式，现在假设我们有一个重载函数 processVal：

```
int processVal(int value);
int processVal(int value, int priority);
```

我们可以给 f 传递 processVal，

```
f(processVal); // fine
```

令人惊讶的是我们居然能做到。f 需要一个函数指针作为它的实参，但 processVal 不是函数指针，甚至不是函数，它是两个不同函数的名称。然而，编译器知道它们需要哪个 processVal：匹配 f 参数类型的那个。因此，它们选择接受一个 int 的 processVal，并将把这个函数的地址传递给 f。

之所以能做到，是因为 f 的声明可以让编译器判断出需要的是哪个版本的 processVal。但是，作为一个函数模板，fwd 没有关于它需要什么类型的任何信息，这使得编译器不可能确定应该传递哪些重载：

```
fwd(processVal); // error! which processVal?
```

processVal 本身也没有类型。没有类型，也就不能进行类型推导，而没有类型推导，我们就发现了另一个完美转发失败的情形。

如果我们试图使用函数模板而不是重载的函数名，也会出现同样的问题。函数模板并不代表一个函数，它表示许多函数：

```
template<typename T>
T workOnVal(T param)      // template for processing values
{ … }
fwd(workOnVal);           // error! which workOnVal instantiation?
```

使 fwd 这样的完美转发函数接受重载函数或模板名称的方式是，手动指定你想要转发的重载或模板实例。例如，你可以创建一个与 f 的参数类型相同的函数指针，用 processVal 或 workOnVal 初始化该指针(从而导致选择正确的 processVal 版本或正确的 workOnVal 实例)，并将指针传递给 fwd：

```
using ProcessFuncType =                         // make typedef;
    int (*)(int);                               // see Item 9
ProcessFuncType processValPtr = processVal;     // specify needed signature
                                                // for processVal
fwd(processValPtr);                             // fine
fwd(static_cast<ProcessFuncType>(workOnVal)); // also fine
```

当然，这要求你知道 fwd 转发的函数指针类型。假设一个完美转发函数有文档写明这个，这也不是不合理。但毕竟，完美转发函数是设计成接受任何类型的，所以如果没有文档告诉你该传递什么，你怎么知道呢？

位字段（Bitfields）

完美转发的最后一个失败情形是，位字段被用作函数实参。要了解这在实践中意味着什么，请看下 IPv4 报头模型：

```
struct IPv4Header {
    std::uint32_t  version:4,
                   IHL:4,
                   DSCP:6,
                   ECN:2,
                   totalLength:16;
    ...
};
```

如果我们的函数 f 声明接受一个 std::size_t 参数，比如说，使用 IPv4Header 对象的 totalLength 字段调用它，编译时没有任何问题：

```
void f(std::size_t sz); // function to call

IPv4Header h;
...
f(h.totalLength); // fine
```

然而，试图通过 fwd 转发 h.totalLength 到 f，结果就不同了：

```
fwd(h.totalLength); // error!
```

问题是 fwd 的参数是一个引用，而 h.totalLength 是一个 non-const 位字段。这听起来可能没有那么糟糕，但是 C++标准说的很清楚：“non-const 引用不能绑定到位字段。”这项禁令有个很好的理由。位字段可以由机器字的任意部分组成(例如，32 位 int 的第 3-5 位)，没有办法直接定位地址。我前面提到过，引用和指针在硬件层是一样的，就像没有办法创建指向任意位（arbitrary bits）的指针一样（C++指出，指针的最小指向单元是 char），也没有办法将引用绑定到任意位。

如果你想绕过不能完美转发位字段的问题，那是很容易的，要知道，任何接受位字段作为实参的函数，接受的是位字段值的拷贝。毕竟，任何函数都不能将引用绑定到位字段，也不能接受指向位字段的指针，因为指向位字段的指针不存在。能传入位字段的形参就是传值

形参和 reference-to-const 形参。在传值形参的情况下，被调用的函数显然接受的是一个位字段值的副本；而在 references-to-const 形参的情况下，标准要求引用实际上绑定到存储在某个标准整数类型(如 int)对象中的位字段值的副本。references-to-const 不是绑定到位字段，它们绑定到拷贝了位字段值的“普通”对象。

那么，将位字段传递给完美转发函数的关键是，充分利用上面的事实，即要转发的函数总是接受位字段值的副本。因此，可以创建一个拷贝并用这个拷贝调用转发函数。在我们使用 IPv4Header 的示例中，这段代码可以执行以下操作技巧：

```
// copy bitfield value; see Item 6 for info on init. form
auto length = static_cast<std::uint16_t>(h.totalLength);
fwd(length); // forward the copy
```

小结（Upshot）

在大多数情况下，完美转发跟大家了解的完全一样。你很少需要关心它。但是当它不灵光时，即，看起来合理的代码编译不通过，或者更糟糕的是，编译通过，但是没有按照预期的方式执行，了解完美转发的不完美之处就很重要了。同样重要的是知道如何解决这些问题。在大多数情况下，这很简单。

需要铭记的要点
● 当模板类型推导失败或推导出错误类型时，完美转发失败。 ● 导致完美转发失败的实参类型包括：花括号初始化、表示空指针的 0 或 NULL、仅声明的完整 static const 数据成员、模板与重载函数名称、以及位字段。

第六章Lambda 表达式

Lambda 表达式是 C++编程中的一个游戏改变者。这是有些令人惊讶，因为它们没有给语言带来新的表达能力。lambda 所能做的一切都是可以通过敲更多的代码来手工完成的。但 lambda 是创建函数对象的一种简便方法，对日常的 C++软件开发的影响是巨大的。没有 lambda， STL 的 " _if "算法(如 std::find_if, std::remove_if, std::count_if 等)往往是仅与最简单的判断一起使用，但有了 lambda，就可以通过各种各样的判断条件来使用这些算法。同样，算法也可以定制比较函数(例如，std::sort，std::nth_element,std::lower_bound,等等)。在 STL 外面，lambda 可以为 std::unique_ptr 和 std::shared_ptr(见条款 18 和 19)快速创建自定义删除器，并且使得线程 API 中条件变量的判断规范，同样简单明了(见条款 39)。除了标准库之外，lambdas 还简化了回调函数、接口适配函数以及一次性调用的上下文特定函数的动态规范。lambda 确实使 C++成为一种更令人愉快的编程语言。

lambda 相关的词汇可能会令人迷惑。这里有一个简要的补习：

- 一个 lambda 表达式就仅仅是一个表达式。下面是一个例子。

```
std::find_if(container.begin(), container.end(),
             [](int val) { return 0 < val && val < 10; });
```

  高亮的表达式就是 lambda。

- 闭包(closure)是由 lambda 创建的运行时对象。取决于捕捉模式，闭包保存捕获数据的副本或引用。在上面的 std::find_if 调用中，闭包就是一个对象，在运行时作为第三个实参传递给 std::find_if。
- 闭包类是实例化闭包的类。每个 lambda 会导致编译器生成一个独一无二的闭包类。lambda 内部的语句变成闭包类成员函数中的可执行指令。

lambda 通常用于创建仅用作函数实参的闭包。就是上面对 std::find_if 调用的情况。不过，闭包一般会被拷贝，所以通常可能有对应于单个 lambda 的闭包类型的多个闭包。例如，在以下代码中，

```
{
    int x;                                  // x is local variable
    ...

    auto c1 =                               // c1 is copy of the
        [x](int y) { return x * y > 55; };  // closure produced
                                            // by the lambda
```

```
    auto c2 = c1;                                   // c2 is copy of c1
    auto c3 = c2;                                   // c3 is copy of c2
    ...
}
```

c1、c2、c3 都是由 lambda 生成的闭包的副本。

通俗地说，模糊 lambda、闭包和闭包类之间的界限是完全可以接受的。但是在接下来的条款中，通常很重要的是，区分什么是在编译期间存在(lambda 和闭包类)，什么是在运行时存在(闭包)，以及它们之间是怎么关联的。

条款 31：避免使用默认捕获模式

C++ 11 中有两种默认捕获模式：按引用和按值。默认按引用捕获可能导致悬挂引用。默认按值捕获诱惑你会认为自己对这个问题免疫(其实你不是)，并且会让你误以为闭包是自给自足的（self-contained）(其实它们可能不是)。

按引用捕获会导致闭包包含对在 lambda 定义的作用域中可用的局部变量或参数的引用。如果闭包的生命周期超过局部变量或参数的生命周期，那么在闭包中引用就会悬挂。例如，假设我们有一个过滤函数的容器，每个函数接受一个 int 并返回一个 bool 指示传入的值是否满足过滤条件：

```
using FilterContainer =                             // see Item 9 for
    std::vector<std::function<bool(int)>>;          // "using", Item 2
                                                    // for std::function

FilterContainer filters;                            // filtering funcs
```

我们可以添加一个 5 的倍数过滤器，如下：

```
filters.emplace_back(                               // see Item 42 for
    [](int value) { return value % 5 == 0; }        // info on
);                                                   // emplace_back
```

然而，我们可能需要在运行时计算除数，即，我们不能硬编码 5 到 lambda。所以添加过滤器可能看起来更像这样：

```
void addDivisorFilter()
{
    auto calc1 = computeSomeValue1();
    auto calc2 = computeSomeValue2();

    auto divisor = computeDivisor(calc1, calc2);
```

```
    filters.emplace_back(                                   // danger!
        [&](int value) { return value % divisor == 0; }    // ref to
    );                                                      // divisor
}                                                           // will dangle!
```

这段代码是一个等待发生的问题。lambda 指向局部变量 divisor，但是当 addDivisorFilter 返回时，这个变量就不存在了。也就是说，filters.emplace_back 返回之后，添加到 filters 的函数基本上就挂了。几乎从它被创建的那刻起，使用该过滤器就会产生未定义的行为。

现在，如果显式指定 divisor 按引用捕获，也就存在同样的问题，

```
filters.emplace_back(
    [&divisor](int value)              // danger! ref to
    { return value % divisor == 0; }  // divisor will
);                                     // still dangle!
```

但是通过显式捕获，更容易看出 lambda 的生存是依赖于 divisor 的生命周期的。同时，写出名字“divisor”，提醒我们要确保 divisor 的生命周期至少和闭包的一样长。相较于一般的“确保没有东西悬挂”这样的告诫，“[&]”表达的是一种更加特别的记忆。

如果你知道闭包会立即使用(例如，将闭包传递给 STL 算法)并且不会被拷贝，那就不会存在上面的那种风险。在那种情况下，你可能会说，既然不存在引用悬挂的风险，因此就没有理由去回避默认的按引用捕获模式。例如，我们的过滤 lambda 可能只作为 C++11 的 std::all_of 的实参使用，它返回某一范围内所有元素是否满足条件：

```
template<typename C>
void workWithContainer(const C& container)
{
    auto calc1 = computeSomeValue1();              // as above
    auto calc2 = computeSomeValue2();              // as above

    auto divisor = computeDivisor(calc1, calc2);   // as above

    using ContElemT = typename C::value_type;      // type of
                                                   // elements in container

    using std::begin;                              // for
    using std::end;                                // genericity;
                                                   // see Item 13

    if (std::all_of(                               // if all values
            begin(container), end(container),      // in container
            [&](const ContElemT& value)            // are multiples
            { return value % divisor == 0; })      // of divisor...
```

```
        ) {
        …                                       // they are...
    } else {
        …                                       // at least one
    }                                           // isn't...
}
```

的确，这是安全的，但它的安全有点不稳定。如果 lambda 在其他上下文中被发现有用(例如，作为要添加到筛选器容器中的函数)，并被复制粘贴到闭包生命周期比 divisor 长的上下文中，你会回到悬挂之城（dangle-city），捕获子句里没有任何东西能特别提醒你，要对 divisor 进行生命周期分析。

长期来看，明确列出 lambda 依赖的局部变量和参数，是更好的软件工程。

顺便说一下，在 C++14 中 lambda 参数规范允许使用 auto，意味着上面的代码在 C++14 可以进行简化。ContElemT 类型定义可以省去，if 条件可以修改如下：

```
if (std::all_of(begin(container), end(container),
                [&](const auto& value)              // C++14
                { return value % divisor == 0; }))
```

解决 divisor 问题的一种方法是采用默认按值捕获模式。也就是说，我们可以像下面这样将 lambda 添加到过滤器中：

```
filters.emplace_back(                               // now
    [=](int value) { return value % divisor == 0; } // divisor
);                                                  // can't dangle
```

这对于本例来说已经足够了，但是，一般来说，默认按值捕获并不是你想象中的反悬挂（anti-dangling）长生不老药。问题是，如果你按值捕获指针，你就指针拷贝到由 lambda 产生的闭包中，但你无法阻止 lambda 之外的代码 delete 指针，并导致你的副本悬挂。

“那是不可能的!”你反驳道，“读完第四章以后，我已经热衷于使用智能指针，只有 C++98 程序员才使用原生指针和 delete。”这可能是真的，但这无关紧要，因为你确实会使用原生指针，而它们可以在你控制权之外被 delete。只是在现代 C++编程风格下，源代码中往往很少有它的影子。

假设 Widget 可以做的一件事是，向过滤器容器添加入口：

```
class Widget {
public:
    …                           // ctors, etc.
    void addFilter() const;     // add an entry to filters

private:
```

```
    int divisor; // used in Widget's filter
};
```

Widget::addFilter 可以定义成下面这样：

```
void Widget::addFilter() const
{
    filters.emplace_back(
        [=](int value) { return value % divisor == 0; }
    );
}
```

对于新手来说，这看起来像是安全的代码。lambda 依赖于 divisor，但默认按值捕获模式确保将 divisor 复制到由 lambda 产生的闭包，对吗？

错了。完全错误。严重错误。致命错误。

捕获仅应用于创建 lambda 的作用域内可见的非静态局部变量(包括参数)。在 Widget::addFilter 内部，divisor 不是局部变量，而是 Widget 类的数据成员。它不能被捕获。然而，如果取消默认捕获模式，代码将无法编译：

```
void Widget::addFilter() const
{
    filters.emplace_back(                                   // error!
        [](int value) { return value % divisor == 0; }     // divisor
    );                                                      // not
}                                                           // available
```

此外，如果尝试显式捕获 divisor (通过值或通过引用——没关系)，捕获无法编译，因为 divisor 不是局部变量或参数：

```
void Widget::addFilter() const
{
    filters.emplace_back(
        [divisor](int value)                // error! no local
        { return value % divisor == 0; }  // divisor to capture
    );
}
```

因此，如果默认按值捕捉子句不能捕捉 divisor，而没有默认按值捕获子句，代码又无法编译，那咋整呢？

这个解释取决于原生指针 this 的隐式使用。每个非静态成员函数都有一个 this 指针，每次你提到类的数据成员时，都会用到 this 指针。例如，在 Widget 的任一成员函数中，编译器内部用 this->divisor 替换 divisor。在使用默认按值捕获的 Widget::addFilter 中，

```
void Widget::addFilter() const
{
    filters.emplace_back(
        [=](int value) { return value % divisor == 0; }
    );
}
```

被捕获的是 Widget 的 this 指针，而不是 divisor。编译器是像下面这样处理的：

```
void Widget::addFilter() const
{
    auto currentObjectPtr = this;

    filters.emplace_back(
        [currentObjectPtr](int value)
        { return value % currentObjectPtr->divisor == 0; }
    );
}
```

理解这一点就等于理解，这个 lambda 产生的闭包的生命周期是与 Widget 的 this 指针紧密联系的。特别地，考虑一下这段代码，它与第 4 章一致，只使用智能指针：

```
using FilterContainer =                 // as before
    std::vector<std::function<bool(int)>>;

FilterContainer filters;                // as before

void doSomeWork()
{
    auto pw =                           // create Widget; see
        std::make_unique<Widget>();     // Item 21 for std::make_unique

    pw->addFilter();                // add filter that uses Widget::divisor
    ...
}                                   // destroy Widget; filters
                                    // now holds dangling pointer!
```

当调用 doSomeWork 时，将创建一个过滤器，该过滤器依赖于 std::make_unique 生成的 Widget 对象。这个过滤器被添加到 filters 中，但是当 doSomeWork 完成时，Widget 被管理它生命周期的 std::unique_ptr（见条款 18）销毁，从那时起，filters 包含了一个带有野指针的入口。

这个特殊问题的解决方法是，创建想要捕获的数据成员的本地副本，然后捕获副本：

```
void Widget::addFilter() const
{
```

```
        auto divisorCopy = divisor;                     // copy data member

        filters.emplace_back(
            [divisorCopy](int value)                    // capture the copy
            { return value % divisorCopy == 0; }        // use the copy
        );
    }
```

老实说，如果采用这种方法，默认按值捕获也可以工作，

```
    void Widget::addFilter() const
    {
        auto divisorCopy = divisor;                     // copy data member

        filters.emplace_back(
            [=](int value)                              // capture the copy
            { return value % divisorCopy == 0; }        // use the copy
        );
    }
```

但为什么要冒险呢？当你认为首先捕获的是 divisor 时，默认捕获模式可能会意外地捕获 this 指针。

在 C++ 14 中，捕获数据成员的更好方法是使用泛型 lambda 捕获（见条款 32）：

```
    void Widget::addFilter() const
    {
        filters.emplace_back(                   // C++14:
            [divisor = divisor](int value)      // copy divisor to closure
            { return value % divisor == 0; }    // use the copy
        );
    }
```

但是，对于一个泛型 lambda 捕获，没有默认捕获模式这样的东东，所以即使在 C++ 14 中，本文的建议，即避免使用默认捕获模式，也是站得住脚的。

默认按值捕获的另一个缺点是，它们暗示相应的闭包是自给自足的，并且与闭包外部数据的更改隔离。一般来说，这是不正确的，因为 lambda 可能不仅依赖于局部变量和参数(它们可能被捕获)，而且还依赖于具有静态存储时期（static storage duration）的对象。这些对象在全局或命名空间作用域内定义，或者在类、函数或文件中声明为 static。这些对象可以在 lambda 中使用，但是它们不能被捕获。然而，默认按值捕获模式的规范会给人一种它们是可以被捕获的错觉。考虑之前的 addDivisorFilter 函数的一个修订版本：

```
    void addDivisorFilter()
    {
```

```
    static auto calc1 = computeSomeValue1();  // now static
    static auto calc2 = computeSomeValue2();  // now static

    static auto divisor =                     // now static
        computeDivisor(calc1, calc2);

    filters.emplace_back(
        [=](int value)                        // captures nothing!
        { return value % divisor == 0; }      // refers to above static
    );

    ++divisor;                                // modify divisor
}
```

偶尔阅读这段代码的人，如果看到“[=]”并认为“OK，lambda 拷贝了它使用的所有对象，因此是自给自足的”，这是可以理解的。但它不是自给自足的。这个 lambda 不使用任何非静态局部变量，因此没有捕获任何东西。相反，lambda 的代码引用了静态变量 divisor。每次调用 addDivisorFilter 后，divisor 递增时，通过该函数添加到 filters 中的任何 lambda 都会展示新的行为(对应于 divisor 的新值)。实际上，这个 lambda 是通过引用捕获 divisor，这与默认按值捕获子句的含义是直接矛盾的。如果不使用默认按值捕获子句，就可以消除这种误读的风险。

需要铭记的要点
● 默认按引用捕获可能导致引用悬挂。 ● 默认按值捕获容易受野指针影响（特别是 this 指针），并且会误导我们，认为 lambda 是自给自足的。

条款 32：使用初始化捕获来移动对象至闭包

有时，按值捕获和按引用捕获都不是你想要的。如果你有个 move-only 对象(例如，std::unique_ptr 或 std::future)，要捕获进闭包，C++11 没辙。如果你有个对象，它的拷贝成本很高，但是移动成本很低(例如，标准库中的大多数容器)，并且希望将该对象捕获进闭包，那么你更愿意移动而不是拷贝它。然而，C++11 同样没辙。

但那是针对 C++11。C++14 则是另一回事了，它直接支持将对象移动到闭包中。如果你的编译器符合 C++ 14，请开心地阅读下去。如果你仍然在使用 C++ 11 编译器，那么你也应该开心地阅读下去，因为在 C++ 11 中有许多方法可以近似移动捕获。

即使接受了 C++ 11，缺少移动捕捉也是一个公认的缺点。直接的补救方法是在 C++ 14 中添加它，但是标准化委员会选择了一条不同的道路。他们引入了一种新的捕获机制，这种机制非常灵活，capture-by-move 只是它能够执行的技巧之一。这个新功能称为初始化捕获（init capture）。它几乎可以做 C++ 11 捕捉及以外的任何事情。初始化捕获不能表达的一件事是默认捕获模式，但是条款 31 说明无论如何你都应该远离那些模式。(对于 C++ 11 捕获所涉及的情况，初始化捕获的语法稍微有些冗长，所以在 C++ 11 捕获可以胜任的情况下，最好用 C++11 捕获。)

使用初始化捕获可以指定

1. **由 lambda 生成的闭包类中的数据成员的名称**
2. **初始化该数据成员的表达式。**

下面是如何使用初始化捕获将 std::unique_ptr 移动到闭包中：

```
class Widget {                    // some useful type
public:
    ...

    bool isValidated() const;
    bool isProcessed() const;
    bool isArchived() const;

private:
    ...
};


auto pw = std::make_unique<Widget>();      // create Widget; see
```

```
                                                    // Item 21 for info on
                                                    // std::make_unique

...                                                 // configure *pw

auto func = [pw = std::move(pw)]                        // init data mbr
            { return pw->isValidated()                  // in closure w/
                     && pw->isArchived(); };            // std::move(pw)
```

突出显示的文本包含初始化捕获。在“=”的左边是你指定的闭包类中数据成员的名称，而右边是初始化表达式。有趣的是，“=”左边的作用域与右边的作用域不同。左边的作用域是闭包类的作用域，右边的作用域与定义 lambda 的作用域相同。在上面的例子中，“=”左边的名称 pw 表示闭包类中的一个数据成员，而右边的名称 pw 表示 lambda 上面声明的对象，即，调用 std::make_unique 初始化的变量。因此，“pw = std::move(pw)”意味着“在闭包中创建一个数据成员 pw，并用 std::move 局部变量 pw 的结果来初始化该数据成员。”

与往常一样，lambda 中的代码属于闭包类的作用域，因此 pw 指的是闭包类的数据成员。

本例中的注释“configure *pw”表明，在 std::make_unique 创建 Widget 之后，lambda 捕获 Widget 的 std::unique_ptr 之前，Widget 将以某种方式进行修改。如果不需要这样的配置，即，如果 std::make_unique 创建的 Widget 处于适合 lambda 捕获的状态，则不需要局部变量 pw，因为闭包类的数据成员可以直接由 std::make_unique 初始化：

```
auto func = [pw = std::make_unique<Widget>()]       // init data mbr
            { return pw->isValidated()              // in closure w/
                     && pw->isArchived(); };        // result of call
                                                    // to make_unique
```

这应该很清楚，C++ 14 的“捕获”概念是对 C++ 11 相当地泛化，因为在 C++ 11 中，不可能捕获表达式的结果。因此，初始化捕获的另一个名称是泛型 lambda 捕获（generalized lambda capture）。

但是，如果你使用的一个或多个编译器不支持 C++ 14 的初始化捕获，该怎么办呢？如何在缺乏对移动捕捉支持的语言中完成移动捕捉？

请记住，lambda 表达式只是一种生成类和创建该类对象的方法。对于 lambda，没有什么是手工无法完成的。举个例子，我们刚刚看到的 C++ 14 代码可以用 C++ 11 写成这样：

```
class IsValAndArch {                                // "is validated
public:                                             // and archived"
    using DataType = std::unique_ptr<Widget>;
```

```
    explicit IsValAndArch(DataType&& ptr)          // Item 25 explains
    : pw(std::move(ptr)) {}                        // use of std::move

    bool operator()() const
    { return pw->isValidated() && pw->isArchived(); }

private:
    DataType pw;
};

auto func = IsValAndArch(std::make_unique<Widget>());
```

这比编写 lambda 的工作量大得多，但它不会改变这样一个事实：如果你希望 C++11 中的类支持对其数据成员进行移动初始化，那么你和你的愿望之间惟一的区别就是花点时间敲键盘。

如果你想继续使用 lambda(考虑到它们的便利性，你可能会这样做)，可以使用 C++11 模拟 move 捕获

1. 将要捕获的对象移动到由 std::bind 生成的函数对象中

2. 为 lambda 提供对“捕获”对象的引用。

如果你熟悉 std::bind，那么代码非常简单。如果你不熟悉 std::bind，那么代码需要一些时间来适应，但这是值得的。

假设你希望创建一个局部 std::vector，在其中放入一组适当的值，然后将其移动到闭包中。在 C++14 中，这很简单：

```
std::vector<double> data;              // object to be moved into closure

...                                    // populate data

auto func = [data = std::move(data)]   // C++14 init capture
            { /* uses of data */ };
```

我突出显示了这段代码的关键部分：你想要移动的对象的类型 (std::vector<double>)，该对象的名称(data)，以及初始化捕获的初始化表达式(std::move(data))。C++ 11 的等价代码如下，我在这里高亮了相同的关键内容：

```
std::vector<double> data;                      // as above

...                                            // as above
```

```
auto func =
    std::bind(                                  // C++11 emulation
        [](const std::vector<double>& data)     // of init capture
        { /* uses of data */ },
        std::move(data)
    );
```

与 lambda 表达式一样，std::bind 也生成函数对象。我调用 std::bind 返回的函数对象来绑定对象。std::bind 的第一个实参是一个可调用对象，后续实参表示要传递给该对象的值。

绑定对象包含传递给 std::bind 的所有参数的副本。对于每个左值参数，bind 对其是拷贝构造，对于每个右值，都是移动构造。在这个例子中，第二个参数是右值 (std::move 的结果，见条款 23)，所以 data 被移动构造到 bind 对象中。这种移动构造是移动捕获模拟的关键，因为将右值移动到绑定对象中就是我们解决无法将右值移动到 C++ 11 闭包中的方法。

当 bind 对象被“调用”时，它存储的参数被传递给最初传递给 std::bind 的可调用对象。在本例中，这意味着，当调用 func(bind 对象)时，func 中移动构造的 data 副本作为参数传递给 lambda。

这个 lambda 与我们在 C++ 14 中使用的 lambda 相同，除了添加了一个参数 data 来对应伪移动捕获的对象。该参数是对 bind 对象中 data 副本的左值引用。(它不是一个右值引用，因为尽管用于初始化 data 副本表达式(“std::move(data)”)是右值，但 data 副本本身是左值。)因此，lambda 内部对 data 的使用，操作的是 bind 对象内部移动构造的 data 副本。

默认情况下，lambda 生成的闭包类中的运算符()成员函数是 const。作用就是闭包中的所有数据成员都表现为 const。然而，bind 对象内移动构造的 data 副本不是 const，所以，为了防止在 lambda 中修改 data 副本，lambda 形参声明为 reference-to-const。如果 lambda 声明为 mutable，那么闭包类中的运算符()将不会声明为 const，那么就可以省去 lambda 参数声明中的 const：

```
auto func =
    std::bind(                                  // C++11 emulation
        [](std::vector<double>& data) mutable   // of init capture
        { /* uses of data */ },                 // for mutable lambda
        std::move(data)
    );
```

因为 bind 对象存储传递给 std::bind 的所有参数的副本，所以我们示例中的 bind 对象包含 lambda 生成的闭包的副本，这是它的第一个参数。因此，闭包的生命周期与绑定对象

的生命周期相同。这很重要，因为这意味着只要闭包存在，包含伪移动捕获对象的 bind 对象也就存在。

如果这是你第一次接触 std::bind，在前面讨论的所有细节就位之前，你可能需要咨询你最喜欢的 C++11 参考文献。即使这样，这些基本要点也应该是清晰的：

- 将对象移动到 C++11 闭包中是不可能的，但是将对象移动到 C++ 11 的 bind 对象中是可能的。
- 在 C++11 中模拟移动捕获包括，将对象移动构造至 bind 对象，然后通过引用将一定能够构造的对象传递给 lambda。
- 因为 bind 对象的生命周期与闭包的生命周期相同，所以可以将 bind 对象中的对象视为闭包中的对象。

作为使用 std::bind 来模拟移动捕获的第二个例子，下面是我们前面看到的 C++14 代码，它用于在闭包中创建 std::unique_ptr：

```
auto func = [pw = std::make_unique<Widget>()]      // as before,
            { return pw->isValidated()             // create pw
                     && pw->isArchived(); };       // in closure
```

而下面是 C++11 模拟的：

```
auto func = std::bind(
                [](const std::unique_ptr<Widget>& pw)
                { return pw->isValidated()
                         && pw->isArchived(); },
                std::make_unique<Widget>()
            );
```

具有讽刺意味的是，我正在展示如何使用 std::bind 来绕过 C++11 lambda 的限制，而条款 34 中，我又提倡使用 lambda 而不是 std::bind。不过，本条款解释了在 C++11 中有些情况下 std::bind 是有用的，而这就是其中之一。(在 C++ 14 中，初始化捕获和 auto 参数等特性避免了这些情况。)

需要铭记的要点
● 使用 C++14 的初始化捕获来移动对象至闭包。 ● C++11 通过手写类或 std::bind 来模拟初始化捕获。

条款 33：对 auto&&形参使用 decltype 来 std::forward

C++14 最令人兴奋的特性之一是在参数规范中使用 auto 的泛型 lambda。这个特性的实现非常简单：lambda 闭包类中的 operator()是一个模板。例如，

```
auto f = [](auto x){ return func(normalize(x)); };
```

闭包类的函数调用运算符如下：

```
class SomeCompilerGeneratedClassName {
public:
    template<typename T>                    // see Item 3 for
    auto operator()(T x) const              // auto return type
    { return func(normalize(x)); }

    ...                                     // other closure class
};                                          // functionality
```

在这个例子中，lambda 对其参数 x 所做的唯一一件事就是将它转发给 normalize。如果 normalize 对左值和右值的处理方式不同，那么这个 lambda 写的就不太对，因为它总是传个左值(参数 x)给 normalize，即使传递给 lambda 的实参是右值。

正确的写法是完美转发 x。要做到这一点，需要对代码做两处更改。首先，x 必须成为通用引用（见条款 24），其次，它必须通过 std::forward(见条款 25)传给 normalize。概念上来说，这些都是微不足道的修改：

```
auto f = [](auto&& x)
        { return func(normalize(std::forward<???>(x))); };
```

然而，在概念和实现之间，问题是传什么类型给 std::forward，即，上面???位置，我应该写什么。

当使用完美转发时，你通常都是在一个使用类型参数 T 的模板函数中，所以你只需编写 std::forward<T>。但是在泛型 lambda 中，没有类型参数 T 给你使用。在 lambda 生成的闭包类内部，模板化 operator()倒是有一个 T，但是不可能从 lambda 引用，因此它对你没有任何作用。

条款 28 解释到，如果将左值实参传递给通用引用，该参数的类型将成为左值引用，如果传递的是右值，该参数将成为一个右值引用。这意味着，在 lambda 中，我们可以通过检查的参数 x 的类型，来判断实参是左值还是右值。decltype(见条款 3)给了我们一个实现途径。如果是左值，decltype(x)得到的类型是左值引用，如果是右值，decltype(x)得到的是右

值引用。

条款 28 同样解释到，调用 std::forward，根据规定，类型实参是左值引用，得到是左值，类型实参是非引用，得到的是右值。在我们的 lambda 中，如果 x 绑定到左值，decltype(x)得到一个左值引用。这符合规定。但是，如果 x 绑定到一个右值, decltype(x)得到的是右值引用，而不是通常的非引用。

但看下条款 28 中展示的 std::forward 简单 C++14 实现：

```
template<typename T>                                // in namespace
T&& forward(remove_reference_t<T>& param)           // std
{
    return static_cast<T&&>(param);
}
```

如果用户代码想要完美转发 Widget 类型的右值，它通常会用 Widget 类型(即非引用类型)实例化 std::forward，并且 std::forward 模板生成如下函数：

```
Widget&& forward(Widget& param)                     // instantiation of
{                                                   // std::forward when
    return static_cast<Widget&&>(param);            // T is Widget
}
```

但是请考虑一下，如果用户代码想要完美转发 Widget 类型的右值，但却将 T 指定为右值引用，会发生什么情况呢。也就是说，考虑一下如果将 T 指定为 Widget&&会发生什么。在 std::forward 的初始实例化和 std::remove_reference_t 的应用之后，但是在引用折叠之前(再次参见条款 28)，std::forward 看起来是这样的：

```
Widget&& && forward(Widget& param)                  // instantiation of
{                                                   // std::forward when
    return static_cast<Widget&& &&>(param);         // T is Widget&&
}                                                   // (before reference-collapsing)
```

应用引用崩溃规则，即右值引用的右值引用变成单独的右值引用，这个实例化就变成：

```
Widget&& forward(Widget& param)                     // instantiation of
{                                                   // std::forward when
    return static_cast<Widget&&>(param);            // T is Widget&&
}                                                   // (after reference-collapsing)
```

如果将这个与 T 为 Widget 时的结果进行比较，就会发现它们是相同的。这意味着使用右值引用类型与非引用类型，实例化 std::forward 的结果是相同的。

这是好消息，因为当 x 的实参是右值时，decltype (x)得到的是右值引用类型。上面我们已经确定，左值传给 lambda 时，decltype(x)会得到我们习惯的类型（即左值引用），并传给 std::forward；现在我们了解到，对于右值，decltype(x)得到的类型不是我们习惯的，不过传

给 std::forward 后得到的结果跟常见类型一样。所以无论是左值还是右值，将 decltype(x)传给 std::forward 都会得到我们想要的结果。我们的完美转发 lambda 就可以写成下面这样：

```
auto f =
    [](auto&& param)
    {
        return func(normalize(std::forward<decltype(param)>(param)));
    };
```

因为 C++ 14 的 lambda 支持可变参数，所以，从接受单个参数，到接受任意数量参数的完美转发 lambda，不过是多加 6 个点而已：

```
auto f =
    [](auto&&... params)
    {
        return func(normalize(std::forward<decltype(params)>(params)...));
    };
```

需要铭记的要点
● 对 auto&&形参使用 decltype 来 std::forward

条款 34：优先使用 lambda 而非 std::bind

bind 是 C++11 继承自 C++98 的 std::bind1 和 std::bind2nd，但是，非正式地说，它从 2005 年起就成为标准库的一部分。这是当标准委员会通过了一份名为 TR1 的文件，其中包括了 bind 的规范。(在 TR1 中，bind 位于不同的命名空间，因此它是 std:: TR1::bind，而不是 std::bind，而且一些接口细节也不同。)这段历史意味着，有些程序员具有 10 年或更长的使用 std::bind 的经验。如果你是其中的一员，你可能不愿意放弃一个对你有用的工具。这是可以理解的，但是在这种情况下，最好是能改变，因为 lambda 几乎总是比 std::bind 更好。在 C++14 中，lambda 不仅更加强大，而且是牢不可破。

本条款假设你熟悉 std::bind。如果你不熟悉，你需要在继续之前获得一个基本的理解。在任何情况下，这样的理解都是值得的，因为你永远不知道什么时候会在必须阅读或维护的 code base 中遇到 std::bind。

像条款 32 一样，我将 std::bind 返回的函数对象称为 bind 对象。

选择 lambda 而不是 std::bind 最重要的原因是 lambda 可读性更好。例如，假设我们有一个可以设置声音报警的函数：

```
// typedef for a point in time (see Item 9 for syntax)
using Time = std::chrono::steady_clock::time_point;
// see Item 10 for "enum class"
enum class Sound { Beep, Siren, Whistle };
// typedef for a length of time
using Duration = std::chrono::steady_clock::duration;
// at time t, make sound s for duration d
void setAlarm(Time t, Sound s, Duration d);
```

进一步假设在程序的某个时刻，我们已经确定需要一个报警，它将在设置完一小时后响起，并持续 30 秒，不过，报警声音还是不确定。我们可以写一个 lambda 来修改 setAlarm 的接口，以便只需要指定一个声音：

```
// setSoundL ("L" for "lambda") is a function object allowing a
// sound to be specified for a 30-sec alarm to go off an hour
// after it's set
auto setSoundL =
    [](Sound s)
    {
        // make std::chrono components available w/o qualification
        using namespace std::chrono;
```

```
        setAlarm(steady_clock::now() + hours(1),      // alarm to go off
                 s,                                   // in an hour for
                 seconds(30));                        // 30 seconds
    };
```

我在 lambda 中高亮显示了对 setAlarm 的调用。这是一个看起来很正常的函数调用，即使是稍有 lambda 经验的读者也可以看到传递给 lambda 的参数 s 是作为实参传递给 setAlarm 的。

在 C++14 中，我们可以用标准后缀 seconds (s)、milliseconds (ms)、hours (h)等，来简化代码，这些后缀建立在 C++11 对 user-defined literals 支持的基础上，在 std::literals 命名空间中实现的，所以上面的代码可以这样重写：

```
auto setSoundL =
    [](Sound s)
    {
        using namespace std::chrono;
        using namespace std::literals;        // for C++14 suffixes

        setAlarm(steady_clock::now() + 1h,    // C++14, but
                 s,                           // same meaning
                 30s);                        // as above
    };
```

下面是我们编写相应的 std::bind 调用的第一次尝试。它有一个错误，我们稍后会修复，但是正确的代码更加复杂，即使是这个简化的版本也会带来一些重要的问题：

```
using namespace std::chrono;              // as above
using namespace std::literals;

using namespace std::placeholders;        // needed for use of "_1"

auto setSoundB =                          // "B" for "bind"
    std::bind(setAlarm,
              steady_clock::now() + 1h,   // incorrect! see below
              _1,
              30s);
```

我想在这里高亮显示对 setAlarm 的调用，就像我在 lambda 中做的那样，但这儿没有调用可以高亮显示。这段代码的读者只需要知道，调用 setSoundB 就会用 std::bind 中指定的时间和持续时间调用 setAlarm。对新手来说，占位符“_1”本质上是不可思议的，但即使是知道这的读者，他也要在心里寻找占位符在 std::bind 参数列表中的位置，才能搞明白 setSoundB 的第一个实参对应于 setAlarm 的第二个参数。该参数的类型在 std::bind 的调用

中没有标识，因此读者必须参考 setAlarm 声明来确定传递给 setSoundB 的参数类型。

但是，正如我所说，代码并不完全正确。在 lambda 中，很明显“steady_clock::now() + 1h”这个表达式是 setAlarm 的一个实参，它将在调用 setAlarm 时计算。这是有道理的：我们希望报警在调用 setAlarm 一个小时后响起。然而，在 std::bind 调用中，“steady_clock::now() + 1h”作为实参传递给 std::bind，而不是传递给 setAlarm。这意味着表达式在调用 std::bind 时计算，并且将得到的时间存储在返回的 bind 对象中。因此，报警将被设置为 std::bind 调用后一个小时响起，而不是在 setAlarm 调用后一个小时响！

解决这个问题需要告诉 std::bind 将表达式的求值延迟到调用 setAlarm 之时，而实现这一点的方法是将第二个 std::bind 调用嵌套到第一个中：

```
auto setSoundB =
    std::bind(setAlarm,
              std::bind(std::plus<>(), steady_clock::now(), 1h),
              _1,
              30s);
```

如果你熟悉 C++ 98 中的 std::plus 模板，你可能会惊讶地发现，在这段代码中，尖括号之间没有指定任何类型，即，代码包含“std::plus<>”，而不是“std::plus<type>”。在 C++14 中，标准运算符模板的模板类型实参通常可以省略，因此这里不需要提供。C++11 没有提供这样的特性，所以 C++ 11 的是：

```
using namespace std::chrono; // as above
using namespace std::placeholders;

auto setSoundB =
    std::bind(setAlarm,
              std::bind(std::plus<steady_clock::time_point>(),
                        steady_clock::now(),
                        hours(1)),
              _1,
              seconds(30));
```

如果，在这一点上，lambda 看起来没有多少吸引力，你可能应该检查一下你的视力。

当重载 setAlarm 时，会出现一个新问题。假设有一个重载接受第四个参数指定报警音量：

```
enum class Volume { Normal, Loud, LoudPlusPlus };
void setAlarm(Time t, Sound s, Duration d, Volume v);
```

lambda 继续像以前一样工作，因为重载解析选择了 setAlarm 的三个参数版本：

```
auto setSoundL =                                // same as before
    [](Sound s)
    {
        using namespace std::chrono;
        setAlarm(steady_clock::now() + 1h,      // fine, calls
                 s,                             // 3-arg version
                 30s);                          // of setAlarm
    };
```

另一方面，std::bind 调用现在不能编译：

```
auto setSoundB =                                // error! which
    std::bind(setAlarm,                         // setAlarm?
              std::bind(std::plus<>(),
                        steady_clock::now(),
                        1h),
              _1,
              30s);
```

问题就是编译器无法确定应该将哪个 setAlarm 传递给 std::bind。编译器只有一个函数名，而这个函数名本身是模棱两可的。

要想使 std::bind 调用编译通过，必须将 setAlarm 转换为适当的函数指针类型：

```
using SetAlarm3ParamType = void(*)(Time t, Sound s, Duration d);

auto setSoundB =                                            // now
    std::bind(static_cast<SetAlarm3ParamType>(setAlarm),    // okay
              std::bind(std::plus<>(),
                        steady_clock::now(),
                        1h),
              _1,
              30s);
```

但是这又引出了 lambda 和 std::bind 之间的另一个区别。在 setSoundL 的函数调用运算符符(即 lambda 的闭包类的函数调用运算符)内部，对 setAlarm 的调用是一个普通的函数调用，编译器可以按照通常的方式内联：

```
setSoundL(Sound::Siren); // body of setAlarm may well be inlined here
```

然而，对 std::bind 的调用，将一个函数指针传递给 setAlarm，这意味着在 setSoundB 的函数调用运算符(即 bind 对象的函数调用运算符)内部，对 setAlarm 的调用通过函数指针进行。编译器不太可能通过函数指针内联函数调用，这意味着通过 setSoundB 对 setAlarm 的调用不太可能像 setSoundL 那样完全内联：

```
setSoundB(Sound::Siren); // setAlarm不太可能被内联
```

因此，使用 lambda 可能比 std::bind 生成更快的代码。

setAlarm 示例只涉及一个简单的函数调用。如果你想做更复杂的事情，lambda 更具优势。例如，考虑下面这个 C++ 14 lambda，返回它的实参是否在最小值(lowVal)和最大值(highVal)之间，其中 lowVal 和 highVal 是局部变量：

```
auto betweenL =
    [lowVal, highVal]
    (const auto& val)                          // C++14
    { return lowVal <= val && val <= highVal; };
```

Bind 同样可以实现，就是代码可读性比较差：

```
using namespace std::placeholders;                      // as above

auto betweenB =
    std::bind(std::logical_and<>(),                     // C++14
              std::bind(std::less_equal<>(), lowVal, _1),
              std::bind(std::less_equal<>(), _1, highVal));
```

在 C++11 中，我们必须指定要比较的类型，然后 std::bind 调用看起来是这样的：

```
auto betweenB =                          // C++11 version
    std::bind(std::logical_and<bool>(),
              std::bind(std::less_equal<int>(), lowVal, _1),
              std::bind(std::less_equal<int>(), _1, highVal));
```

当然，在 C++11 中，lambda 不能使用 auto 参数，因此也必须有个类型：

```
auto betweenL =                          // C++11 version
    [lowVal, highVal]
    (int val)
    { return lowVal <= val && val <= highVal; };
```

无论如何，我希望我们能够同意 lambda 版本不仅代码量少，而且更易于理解和维护。

早些时候，我谈到，对于那些 std::bind 经验很少的人来说，占位符 (例如_1、_2 等)本质上是不可思议的。但难懂的不仅仅是占位符的行为。假设我们有一个函数来创建 Widget 的压缩副本，

```
enum class CompLevel { Low, Normal, High };    // compression level

Widget compress(const Widget& w,               // make compressed
                CompLevel lev);                // copy of w
```

并且我们想要创建一个函数对象，它允许我们指定 Widget w 应该被压缩多少。使用 std::bind 创建这样一个对象：

```
Widget w;

using namespace std::placeholders;

auto compressRateB = std::bind(compress, w, _1);
```

现在，当我们将 w 传递给 std::bind 时，它必须被存储起来，以便稍后调用 compress。它存储在 compressRateB 中，但是它是如何存储的，通过值还是通过引用呢？这是有区别的，因为如果在 std::bind 和 compressRateB 的调用之间修改了 w，那么通过引用存储 w 将反映出更改，而通过值存储则不会。

答案是，它是按值存储的，但是知道它的唯一方法就是记住 std::bind 是如何工作的，在 std::bind 的调用中没有任何这样的信息。相对而言，lambda 是显式指定按值捕获还是按引用捕获：

```
auto compressRateL =                    // w is captured by
    [w](CompLevel lev)                  // value; lev is
    { return compress(w, lev); };       // passed by value
```

（备注：std::bind 总是拷贝它的实参，但是调用者可以通过应用 std::ref 来实现引用存储的效果。如，`auto compressRateB = std::bind(compress, std::ref(w), _1);`的结果就是 compressRateB 表现的就像持有的是 w 的引用而非拷贝）

同样显式的是参数如何传递给 lambda。这里，很明显参数 lev 是按值传递的。因此：

```
compressRateL(CompLevel::High); // arg is passed by value
```

但是在对 std::bind 返回对象的调用中，参数是如何传递的呢？

```
compressRateB(CompLevel::High); // how is arg passed?
```

同样，唯一知道的方法是记住 std::bind 是如何工作的。(答案是，传递给 bind 对象的所有实参都是通过引用传递的，因为此类对象的函数调用运算符用的是完美转发。)

那么，与 lambdas 相比，使用 std::bind 的代码可读性更差，表达性更差，可能效率也更低。在 C++ 14 中，std::bind 没有任何合理的使用场景。然而，在 C++ 11 中，std::bind 可以在两种受限的情形下合理使用：

1. **移动捕获**。C++ 11 lambda 不提供移动捕获，但是可以通过 lambda 和 std::bind 的组合来模拟。详细信息，请参考条款 32，它还解释了在 C++ 14 中，lambda 支持初始化捕获，就不需要这种模拟了。
2. **多态函数对象**。因为 bind 对象上的函数调用运算符使用的是完美转发，所以它可以接受任何类型的实参(条款 30 描述了完美转发的限制)。当你希望 bind 一个带有

模板化函数调用运算符的对象时，这可能非常有用。例如，给定这个类，

```
class PolyWidget {
public:
    template<typename T>
    void operator()(const T& param);
    ...
};
```

std::bind 可以像下面这样绑定 PolyWidget：

```
PolyWidget pw;
auto boundPW = std::bind(pw, _1);
```

然后就可以用不同的实参来调用 boundPW：

```
boundPW(1930);        // pass int to PolyWidget::operator()
boundPW(nullptr);     // pass nullptr to PolyWidget::operator()
boundPW("Rosebud");   // pass string literal to PolyWidget::operator()
```

没有办法用 C++11 来做这个。然而，在 C++14 中，它很容易通过带 auto 参数的 lambda 实现：

```
auto boundPW = [pw](const auto& param) // C++14
                   { pw(param); };
```

当然，这些都是边界情况，而且它们还是暂时的边界情况，因为支持 C++ 14 的编译器越来越常见。

当 bind 在 2005 年被非正式地添加到 C++中时，它是对 1998 年的一个很大改进。C++11 的 lambda 使 std::bind 几乎过时，而到 C++ 14，std::bind 已经没有用武之地了。

需要铭记的要点
● lambda 可读性更强，表达能力更强，并且比 std::bind 更高效。 ● 仅在 C++11 中，std::bind 可能对实现移动捕获或 bind 具有模板化函数调用运算符的对象有用。

第七章并发 API

C++ 11 最大的成功之一是将并发集成到语言和库中。熟悉其他线程 API(例如 pthread 或 Windows 线程)的程序员有时会对 C++提供了相当简单的特性集感到惊讶，但这是因为 C++对并发的大量支持是以对编译器-编写器约束的形式提供的。由此产生的语言保证意味着，在 C++的历史上，程序员第一次可以跨所有平台编写具有标准行为的多线程程序。这为构建极具表现力的库奠定了坚实的基础，并且标准库的并发元素 (tasks, futures, threads, mutexes, condition variables, atomic objects 等等)只是一组工具的开始，这些工具必将成为并发 C++软件开发中越来越丰富的工具集。

在下面的条款中，请记住标准库有两个用于 future 的模板：std::future 和 std::shared_future。在很多情况下，区别并不重要，所以一般在我谈到 future 时，我指的是这两种。

条款 35：优先使用 task-based 编程而非 thread-based

如果希望异步运行 doAsyncWork 函数，有两个基本选择。你可以创建 std::thread 来运行 doAsyncWork，从而使用基于线程的方法：

```
int doAsyncWork();
std::thread t(doAsyncWork);
```

或者你可以将 doAsyncWork 传递给 std::async，这是一种基于任务的策略：

```
auto fut = std::async(doAsyncWork); // "fut" for "future"
```

在这些调用中，传递给 std::async(例如 doAsyncWork)的函数对象被认为是一个任务。

基于任务的方法通常优于基于线程的方法，我们已经看到的少量代码已经说明了一些原因。此处，doAsyncWork 有个返回值，我们可以合理地假设调用 doAsyncWork 的代码对此感兴趣。对于基于线程的调用，没有直接访问它的方法。使用基于任务的方法很容易，因为从 std::async 返回的 future 提供了 get 函数。如果 doAsyncWork 发出异常，get 函数甚至更重要，因为 get 也提供了对异常的访问。使用基于线程的方法，如果 doAsyncWork 抛出异常，程序就挂了(通过调用 std::terminate)。

基于线程的编程和基于任务的编程之间最根本的区别在于，基于任务的编程所包含的抽象层次更高。它将你从线程管理的细节中解放出来，这个也提醒我需要总结一下并发 C++

软件中“线程”的三种含义：

- **硬件线程**是实际执行计算的线程。现代机器架构为每个 CPU 核心提供一个或多个硬件线程。
- **软件线程**(也称为操作系统线程或系统线程)是操作系统横跨所有进程和调度，管理其在硬件线程上的执行。通常可以创建比硬件线程更多的软件线程，因为当软件线程阻塞时(例如，I/O，等待互斥锁或条件变量)，可以通过执行其他未阻塞的线程来提高吞吐量。
- std::thread 是 C++进程中充当底层软件线程句柄的对象。有些 std::thread 对象表示“null”句柄，即。没有关联任何软件线程，因为它们可能处于默认构造状态(因此没有要执行的函数)，已经被 move 过了(move 过去的 std::thread 再充当底层软件线程的句柄)，已经 join 了(运行的函数已经结束)，或者已经 detach(与底层软件线程之间的连接已被切断)。

软件线程是一种有限的资源。如果你试图创建超出系统所能提供的极限，就会抛出一个 std::system_erro 异常，即使你要运行的函数不抛异常，那也可能会有异常抛出。例如，即使 doAsyncWork 是 noexcept，

```
int doAsyncWork() noexcept; // see Item 14 for noexcept
```

下面的语句也可能抛出异常：

```
std::thread t(doAsyncWork); // throws if no more
                            // threads are available
```

好的软件必须以某种方式处理这种可能性，但是如何处理呢？一种方法是在当前线程上运行 doAsyncWork，但是这会导致负载不平衡，如果当前线程是 GUI 线程，则会出现响应问题。另一个选项是等待一些现有的软件线程完成，然后再次尝试创建一个新的 std:: thread，但是现有的线程可能正在等待 doAsyncWork 某个操作的执行(比如，生成一个结果或通知一个条件变量)。

即使你没有耗尽线程，也可能会遇到超额申请（oversubscription）的问题，就是待运行（ready-to-run）软件线程比硬件线程多。当发生这种情况时，线程调度器(通常是操作系统的一部分)对硬件上的软件线程进行时间分片（time-slice）。当一个线程的时间片完成而另一个线程的时间片开始时，将执行上下文切换。这种上下文切换增加了系统的总体线程管理开销，特别上在调度的硬件线程与上次时间片不在同一个 CPU 核心时，这种上下文切换的开销尤其大。在这种情况下，(1)CPU 缓存对于该软件线程是 cold 状态(即，只有很少的数据

和指令对它有用)，(2)在该内核上运行“新”软件线程，会“破坏”一些“旧”线程的 CPU 缓存，那些“旧”线程已经在该内核上运行过并可能再次调度过来。

避免超额申请是很难的，因为软件与硬件线程的最优比例取决于软件线程可运行的频率，而这可以动态地改变，例如，当程序从重 I/O(I/O-heavy)区域进入重计算(computation-heavy)区域时。软硬件线程的最佳比例还取决于上下文切换的成本以及软件线程如何有效地使用 CPU 缓存。此外，硬件线程的数量和 CPU 缓存的细节(比如，缓存空间多大以及对应的速度)取决于机器体系架构，所以即使你在一个平台上对应用程序进行了优化从而避免了超额申请(同时仍然保持硬件繁忙)，但不能保证你的解决方案在其他类型的机器上也能很好的工作。

如果你可以将这些问题丢出去，那么你的生活就会变得无比惬意，正好 std::async 就是接盘侠：

```
auto fut = std::async(doAsyncWork);   // 把线程管理丢给标准库
```

这个调用将线程管理责任转移给 C++的标准库。例如，接收线程耗尽异常的可能性大大降低，因为这个调用可能永远不会产生异常。“怎么可能呢?”你可能会想知道，“如果我要求的软件线程比系统所能提供的要多，那么我是用 std:: thread 还是 std::async 又有什么关系呢?” 当然有关系了，因为以这种形式调用 std::async 时(即，使用默认的启动策略，见条款 36)，并不保证它会创建一个新的软件线程。相反，它允许调度器将指定的函数 (此例中 doAsyncWork)分配在请求 doAsyncWork 结果的线程上运行（即，在调用 future 的 get 或 wait 的线程上），而合理的调度程序可以在超额申请或线程耗尽时利用这种自由度。

如果你是自己悟出了 “在获取结果的线程上运行”这个技巧，我要提醒你，它可能会导致负载均衡问题，而这些问题不会因为 std::async 和运行时调度器而消失。不过，在负载均衡方面，运行时调度器可能比你更全面地了解计算机上正在发生的事情，因为它管理来自所有进程的线程，而不仅仅是你的代码运行的进程中的线程。

用了 std::async, GUI 线程可能存在响应问题，因为调度器无法知道哪个线程具有严格的响应性需求。在这种情况下，你需要使用 std::launch::async 启动策略，这将确保你要运行的函数在不同的线程上执行(参见条款 36)。

最先进的线程调度程序使用系统范围内的线程池来避免超额申请，并通过 work-stealing 算法提高硬件核心的负载均衡。C++标准没有要求必须使用线程池或 work-stealing，而且，老实说，C++11 并发规范的有些技术方面，使得应用这两个技术比我们想象的要困难。然而，一些供应商在他们的标准库中利用了这项技术，我们有理由期待在这一领域继续取得进

展。如果你在并发编程中采用基于任务的方法，那么当这种技术变得更加广泛时，你将自动获得它的好处。另一方面，如果你直接使用 std::threads 进行编程，那么你就要处理线程耗尽、超额申请和负载均衡的问题，更不用说你的解决方案如何与同一机器上的其他进程的解决方案相匹配了。

与基于线程的编程相比，基于任务的设计免去了手工线程管理的痛苦，并且提供了一种自然的方式来检查异步执行函数的结果(即，返回值或异常)。不过，在某些情况下，直接使用线程可能比较合适，包括：

- **需要访问底层线程实现的 API**。C++并发 API 通常是使用底层平台特定的 API 实现的，通常是 pthread 或 Windows 线程。这些 API 目前比 C++提供的更丰富。(例如，C++没有线程优先级或亲缘关系的概念。)为了提供对底层线程 API 的访问，std::thread 对象通常提供 native_handle 成员函数。而 std::future(即，std::async 返回的)没有对应的功能。
- **需要并且能够优化应用程序的线程使用**。例如，情况可能是这样，你已知知道部署的机器有固定的硬件特性，并且你开发的服务器软件是部署在那台机器上唯一重要的软件。
- **需要实现 C++并发 API 之外的线程技术**，例如，在 C++未提供线程池的平台上，实现线程池技术。

然而，这些情况并不常见。大多数情况下，你应该选择基于任务的设计，而不是使用线程编程。

需要铭记的要点
● std::thread API 没有提供从异步运行的函数中获取返回值的直接方法，并且如果这些函数抛出异常，程序就会终止。 ● 基于线程的编程需要手工管理线程耗尽、超额申请、负载均衡，以及适应新平台。 ● 通过 std::async 进行基于任务的编程，使用默认的启动策略，可以为你处理上面大部分问题。

条款 36：当必须是异步时，指定 std::launch::async

当你调用 std::async 来执行一个函数(或其他可调用对象)时，你通常希望异步运行该函数。但这不一定是你要求 std::async 做的。你实际上是要求函数按照 std::async 启动策略运行。有两种标准策略，由 std::launch 这个 scoped enum (有关 scoped enum 的信息，请参阅条款 10) 定义。假设函数 f 被传递给 std::async 执行，

- **std::launch::async 启动策略**意味着 f 必须异步运行，即在另一条线程上。
- **std::launch::deferred 启动策略**意味着，只有在 std::async 返回的 future 上调用 get 或 wait 时，f 才可能运行。也就是说，f 的执行延迟（deferred），直到发出这样的调用。调用 get 或 wait 时，f 将同步执行，即，调用者将阻塞，直到 f 完成运行。如果既不调用 get 也不调用 wait, f 将永远不会运行。

也许令人惊讶的是，std::async 的默认启动策略不是这两种策略的任何一种，而是这两个的组合。以下两个调用的含义完全相同：

```
auto fut1 = std::async(f);                          // 默认启动策略

auto fut2 = std::async(std::launch::async |         // 要么异步，要么延迟
                       std::launch::deferred,
                       f);
```

因此，默认策略允许异步或同步运行 f。正如条款 35 所指出的，这种灵活性允许 std::async 和标准库的线程管理组件共同负责创建和销毁线程、避免超额申请和负载均衡。这就是 std::async 并发编程如此方便的原因之一。

但是使用默认启动策略的 std::async 有一些有趣的含义。给定执行下面语句的线程 t，

```
auto fut = std::async(f); // run f using default launch policy
```

- **不可能预测 f 是否会与 t 同时运行**，因为 f 可能被调度为延迟运行。
- **无法预测 f 是否运行在与调用 fut 的 get 或 wait 的不同线程上**。如果那个线程是 t，这意味着，不可能预测 f 是否运行在与 t 不同的线程上。
- **根本不可能预测 f 是否运行**，因为不可能保证在程序的每条路径上都调用 fut 的 get 或 wait。

默认的启动策略的调度灵活性，常常与 thread_local 变量的使用，很糟糕地混在一起，因为这意味着，如果 f 读取或写入这样的线程本地存储(thread-local Storage，TLS)，就不可能预测哪个线程的变量会被访问：

```
auto fut = std::async(f);     // TLS for f possibly for
                              // independent thread, but
                              // possibly for thread
                              // invoking get or wait on fut
```

它还会影响到使用超时的基于等待的循环，因为在延迟的任务上调用 wait_for 或 wait_until(参见条款 35)会生成 std::launch::deferred 值。这意味着下面的循环，看起来最终应该会终止，实际上可能永远运行下去：

```
using namespace std::literals;        // C++14的持续时间后缀；见条款34

void f()                              // f sleeps for 1 second,
{                                     // then returns
    std::this_thread::sleep_for(1s);
}

auto fut = std::async(f);             // 异步运行f(概念上看)

while (fut.wait_for(100ms) !=         // 循环知道f运行完成，
       std::future_status::ready)     // 但f可能永远都不运行！
{
    ...
}
```

如果 f 与调用 std::async 的线程同时运行(即，选择的启动策略是 std::launch::async)，这里没有问题(假设 f 最终会结束)，但是如果 f 被延迟，fut.wait_for 就总是返回 std::future_status::deferred，那就永远不会等于 std::future_status:: ready，所以循环永远不会终止。

在开发和单元测试期间，这种 bug 很容易被忽略，因为它可能只在重载（heavy loads）时才会出现，就是可能导致超额申请或线程耗尽的情况，那时任务最有可能被延迟。毕竟，如果硬件没有超额申请或线程耗尽，运行时系统没有理由不为并发执行调度任务。

解决方法很简单：只需检查与 std::async 调用相对应的 future，看看任务是否被延迟，如果延迟，避免进入基于超时的循环。不幸的是，没有直接的方法来询问 future 它的任务是否被推迟，相反，你必须调用一个基于超时的函数，如 wait_for。在这种情况下，你并不是真的想等待任何东西，你只是想看看返回值是否为 std::future_status::deferred，因此在必要的条件下，别犹豫，就用 0 调用 wait_for：

```
auto fut = std::async(f);                 // as above

if (fut.wait_for(0s) ==
```

```
    std::future_status::deferred)          //如果任务被延迟
{
                                           // ...使用wait or get
    …                                      // 同步调用f

} else {                                   // 任务未延迟
    while (fut.wait_for(100ms) !=          // 不可能永远循环
           std::future_status::ready) {    // (assuming f finishes)
        …  // 任务要么在运行（英文原版用的是deferred，
           // 个人觉得应该是错了，译者注），要么已经完成，
           // 所以可以做一些并行工作，直到任务完成
    }

    …      // fut is ready
}
```

这些分析的结果是，只要满足以下条件，使用 std::async 的默认启动策略是可以的：

- 该任务不需要与调用 get 或 wait 的线程同时运行。
- 读写哪个线程的 thread_local 变量无关紧要。
- 要么保证 get 或 wait 会被调用，要么可以接受任务可能永远不执行。
- 使用 wait_for 或 wait_until 的代码考虑了延迟状态的可能性。

如果这些条件不能满足，你可能要确保 std::async 调度任务以异步执行。这样做的方法是在调用时将 std::launch::async 作为第一个参数：

```
auto fut = std::async(std::launch::async, f); // launch f asynchronously
```

实际上，如果有一个类似 std::async 的函数，它自动使用 std::launch::async 作为启动策略，这会是一个非常方便的工具，高兴的是，它很容易编写。下面是 C++ 11 版本：

```
template<typename F, typename... Ts>
inline
std::future<typename std::result_of<F(Ts...)>::type>
reallyAsync(F&& f, Ts&&... params)             // return future
{                                              // for asynchronous
    return std::async(std::launch::async,      // call to f(params...)
                      std::forward<F>(f),
                      std::forward<Ts>(params)...);
}
```

这个函数接受一个可调用对象 f 和任意个参数，并完美转发给 std::async(参见条款 25)，使用 std::launch::async 启动策略。与 std::async 类似，它为在 params 上调用 f 的结果返回一个 std::future。确定结果的类型很容易，因为类型特性 std::result_of 可以实现。(类型

特征的一般信息见条款 9。)

reallyAsync 就可以像 std::async 使用：

```
auto fut = reallyAsync(f);    // run f asynchronously;
                              // throw if std::async
                               // would throw
```

C++14 可以简化返回类型：

```
template<typename F, typename... Ts>
inline
auto                                  // C++14
reallyAsync(F&& f, Ts&&... params)
{
    return std::async(std::launch::async,
                      std::forward<F>(f),
                      std::forward<Ts>(params)...);
}
```

这个版本清楚地表明，reallyAsync 只使用 std::launch::async 启动策略调用 std::async。

需要铭记的要点
● std::async 的默认启动策略允许异步和同步两种执行方式。 ● 上面这种灵活性，导致了访问 thread_local 时的不确定性，意味着任务可能永远不会执行，并影响基于超时的 wait 调用的程序逻辑。 ● 如果必须执行异步任务，则指定 std::launch::async。

条款 37：确保所有路径上 std::thread 都是 unjoinable

每个 std::thread 对象都处于这两种状态之一：可连接的(joinable)或不可连接的(unjoinable)。一个可连接的 std::thread 对应于正在或可能运行的底层异步执行线程。举个栗子，如果 std::thread 对应的底层线程是阻塞的或正等待调度，那它就是可连接的；如果对应的的底层线程已经执行完毕，也可以认为 std::thread 对象是可连接的。

一个 unjoinable std::thread 就是如你所想的，非 joinable std::thread，包括：

- 默认构造的 std::thread。这样的 std::thread 没有函数可执行，因此不对应底层的执行线程。
- 被移走的 std::thread 对象。移动的结果就是，以前 std::thread 对应的底层执行线程(如果有的话)现在对应于另一个不同的 std::thread。
- 已经连接过的 std::thread。在 join 之后，std::thread 对象不再与已结束运行的底层执行线程相对应。
- 已经分离的 std::thread。分离（detach）将断开 std::thread 对象与底层执行线程之间的连接。

线程的可连接性很重要的一个原因是，如果调用了可连接线程的析构函数，程序的执行就会终止。例如，假设我们有一个函数 doWork，它以过滤函数 filter 和最大值 maxVal 作为参数。doWork 检查参数以确保满足其计算所需的所有条件，然后计算 0 到 maxVal 所有符合过滤条件的值并展示。如果进行过滤很费时，并且确定 doWork 的条件是否满足也很费时，那么这两件事理由当然地应该并行执行。

我们倾向于为此采用基于任务的设计(见条款 35)，但是让我们假设我们想设置执行过滤线程的优先级，条款 35 解释到，这需要使用线程的原生句柄，并且只能通过 std::thread API 访问，基于任务的 API(即，futures)不提供这种接口。因此，我们的方法是基于线程，而不是任务。

我们写出如下代码：

```
constexpr auto tenMillion = 10000000;          // see Item 15 for constexpr

bool doWork(std::function<bool(int)> filter,   // returns whether
            int maxVal = tenMillion)           // computation was
{                                              // performed; see
                                               // Item 2 for
                                               // std::function
```

```
    std::vector<int> goodVals;                  // values that
                                                // satisfy filter

    std::thread t([&filter, maxVal, &goodVals] // populate
                  {                             // goodVals
                      for (auto i = 0; i <= maxVal; ++i)
                          { if (filter(i)) goodVals.push_back(i); }
                  });

    auto nh = t.native_handle(); // use t's native
    …                            // handle to set t's priority

    if (conditionsAreSatisfied()) {
        t.join();                           // let t finish

        performComputation(goodVals);
        return true;                        // computation was
    }                                       // performed

    return false;                           // computation was
}                                           // not performed
```

在解释这段代码有问题的原因之前，我想指出，C++ 14 可以通过撇号分隔数字，使数千万的初始化值更具可读性：

```
constexpr auto tenMillion = 10'000'000; // C++14
```

我还想指出，在t开始运行之后设置它的优先级，有点像在马跑掉之后关闭马厩的大门。更好的设计应该是 t 以挂起状态开始(这样就可以在它执行任何计算之前调整它的优先级)，但是我不想用这段代码分散你的注意力。如果代码的缺失让你更加心烦，请转到条款 39，因为它显示了如何启动挂起的线程。

回到 doWork。如果 conditionsAreSatisfied()返回 true，一切正常，但是如果它返回 false 或抛出异常，std::thread 对象 t 在 doWork 结束调用其析构函数时，状态是可连接的，那就会导致程序终止。

你可能想知道为什么 std::thread 析构函数会这样做。这是因为另外两个显而易见的选择可能更糟糕：

- **隐式连接（implicit join）**。在这种情况下，std::thread 的析构函数将等待其底层异步执行线程完成。这听起来很合理，但可能会导致难以追踪的性能异常。例如，如果 conditionsAreSatisfied()已经返回 false，doWork 还要等待计算过滤所有值，

这违反常理。

- **隐式分离（implicit detach）**。在这种情况下，std::thread 的析构函数将切断 std::thread 对象与其底层执行线程之间的连接，底层线程将继续运行。这个比连接提起来更不合理，它可能导致更糟糕的调试问题。例如，在 doWork 中，goodVals 是引用捕获的一个局部变量。它还在 lambda 内部进行了修改(通过调用 push_back)。那么，假设在 lambda 异步运行时，conditionsAreSatisfied()返回 false。在这种情况下，doWork 就返回了，它的局部变量(包括 goodVals)将被销毁，它的栈将被弹出，其线程的执行将在 doWork 的调用点继续。

  该调用点之后的语句在某些时候会执行额外的函数调用，并且至少有一个这样的调用可能最终会使用 doWork 栈曾经占用的部分或全部内存。让我们调用这样一个函数 f。当 f 运行时，doWork 启动的 lambda 仍然异步运行。那个 lambda 可以在栈内存上调用 push_back，这个栈内存曾经是 goodVals 的，但现在它位于 f 的栈内。这样的调用将修改曾经是 goodVals 的内存，这意味着从 f 的角度来看，它的栈中的内存内容可以自发地改变！想象一下调试它的乐趣。

标准委员会认为销毁可连接线程的后果非常可怕，以至于他们从根本上禁止了它，即，析构可连接线程就会导致程序终止。

这使你有责任确保，如果使用 std::thread 对象，那么它在超出其作用域的每个路径上都是不可连接的。但是覆盖每条路径可能很复杂，包括溢出作用域的末尾，以及通过 return、continue、break、goto 或异常跳出作用域。路径太多了。

任何时候，如果你想在跳出区块的每条路径上都执行某个操作，通常的方法是将该操作放入本地对象的析构函数中。这些对象称为 RAII 对象，它们的类称为 RAII 类。(RAII 代表“资源获取即初始化(Resource Acquisition Is Initialization)”，尽管该技术的关键是销毁，而不是初始化)。RAII 类在标准库中很常见，包括 STL 容器(每个容器的析构函数销毁容器的内容并释放其内存)、标准智能指针(条款 18 - 20 解释了，std::unique_ptr 的析构函数在其指向的对象上调用 deleter，std::shared_ptr 和 std::weak_ptr 的析构函数递减引用计数)、std::fstream 对象(它们的析构函数关闭它们对应的文件)等等。但是 std::thread 对象还没有标准的 RAII 类，这可能是因为标准化委员会拒绝将 join 和 detach 作为默认选项，根本不知道这样的类应该做什么。

幸运的是，自己写一个并不困难。例如，下面的类允许调用者指定，当 ThreadRAII 对象(std::thread 的 RAII 对象)销毁时，应该调用 join 还是 detach：

```
class ThreadRAII {
public:
    enum class DtorAction { join, detach };   // see Item 10 for
                                              // enum class info

    ThreadRAII(std::thread&& t, DtorAction a) // in dtor, take
    : action(a), t(std::move(t)) {}           // action a on t

    ~ThreadRAII()
    {                                         // see below for
        if (t.joinable()) {                   // joinability test
            if (action == DtorAction::join) {
                t.join();
            } else {
                t.detach();
            }
        }
    }

    std::thread& get() { return t; }          // see below

private:
    DtorAction action;
    std::thread t;
};
```

我估计这段代码在很大程度上已经不言自明，但以下几点还是可以帮你再深入理解一下：

- 构造函数只接受 std::thread 右值，因为我们希望将传入的 std::thread 移动到 ThreadRAII 对象中。(回想一下，std::thread 对象是不可拷贝的。)
- 构造函数中的参数顺序对调用者来说是一目了然 (首先指定 std::thread，然后指定析构操作，这比反过来更直观)，但是成员初始化列表的顺序是为了匹配数据成员声明的顺序，数据成员将 std::thread 对象放在最后。在这个类中，顺序没有区别，但是一般来说，一个数据成员的初始化可能依赖于另一个数据成员，而且因为 std::thread 对象可能在初始化之后，立即开始运行一个函数，在类中最后声明是一个好习惯，这确保在构造 std::thread 数据成员时，前面的所有数据成员都已经初始化，因此可以被 std::thread 数据成员所对应的异步运行线程安全地访问。
- ThreadRAII 提供了一个 get 函数来提供对底层 std::thread 对象的访问。这类似于标准智能指针类提供的 get 函数，提供对其底层原生指针的访问。提供 get 可以避免 ThreadRAII 重复提供完整的 std::thread 接口，也意味着 ThreadRAII 对象可以

在需要 std::thread 对象的上下文中使用。

- 在 ThreadRAII 析构函数调用 std::thread 对象 t 上的成员函数之前，它检查以确保 t 是可连接的。这是必要的，因为在不可连接的线程上调用 join 或 detach 会产生不确定行为。可能存在的情况是，用户构造一个 std::thread，并创建一个 ThreadRAII 对象，调用 get 获得对 t 的访问，然后执行 move 或 join 或 detach，这些操作中的每一个都会导致 t 不可连接。

  如果你担心下面这段代码

```
if (t.joinable()) {
    if (action == DtorAction::join) {
        t.join();
    } else {
        t.detach();
    }
}
```

  存在竞争，因为在执行 t.joinable()和调用 join 或 detach 之间，另一个线程可能导致 t 不可连接，那么你的直觉是值得称赞的，但是你的担心是没有根据的。线程对象只能通过成员函数调用(例如，join、detach 或移动操作)将状态从可连接变为不可连接。在调用 ThreadRAII 对象的析构函数时，其他线程不应该再调用该对象的成员函数，如果存在同步调用，当然就会有竞争，但它不是在析构函数中，而是在尝试同时调用该对象两个成员函数(析构函数和其他函数)的用户代码中。一般来说，只有都是 const 成员函数时，同时调用才是安全的(请参阅条款 16)。（所以，是否存在竞争取决于用户调用代码，ThreadRAII 析构函数内部不存在——译者注）

在我们的 doWork 例子中，使用 ThreadRAII 实现如下：

```
bool doWork(std::function<bool(int)> filter,   // as before
            int maxVal = tenMillion)
{
    std::vector<int> goodVals;                 // as before

    ThreadRAII t(                              // use RAII object
        std::thread([&filter, maxVal, &goodVals]
                    {
                    for (auto i = 0; i <= maxVal; ++i)
                    { if (filter(i)) goodVals.push_back(i); }
                    }),
        ThreadRAII::DtorAction::join           // RAII action
    );
```

```
        auto nh = t.get().native_handle();
        ...

        if (conditionsAreSatisfied()) {
            t.get().join();
            performComputation(goodVals);
            return true;
        }

        return false;
    }
```

在本例中，我们选择在 ThreadRAII 析构函数执行 join，因为，正如我们前面看到的，执行 detach 可能会导致一些真正可怕的调试问题。我们前面也看到，执行 join 可能会导致性能异常(坦率地说，也可能出现令人讨厌的调试问题)，但矮子当中选将军，还是 join 好些，毕竟 detach 可能引起不确定行为，从而导致程序终止。

唉，条款 39 演示了，使用 ThreadRAII 在 std::thread 销毁上执行 join，有时不仅会导致性能异常，还可能会导致程序挂起。这类问题的“适当”解决方案是与异步运行的 lambda 通信，告诉它我们不再需要它运行了，它应该尽早返回，但是 C++ 11 不支持可中断线程（interruptible thread）。不过，它们可以手工实现，但这个超出了本书的范围。（备注：可以参阅 Anthony Williams 的《C++ Concurrency in Action》 (Manning Publications, 2012),section 9.2）

条款 17 解释说，因为 ThreadRAII 声明了一个析构函数，所以不会有编译器生成的移动操作，但是没有理由不让 ThreadRAII 对象移动。如果编译器生成这些函数能胜任工作，那么就可以显式声明 default：

```
class ThreadRAII {
public:
    enum class DtorAction { join, detach };          // as before

    ThreadRAII(std::thread&& t, DtorAction a)        // as before
    : action(a), t(std::move(t)) {}

    ~ThreadRAII()
    {
        ...                                          // as before
    }

    ThreadRAII(ThreadRAII&&) = default;              // support
```

```
    ThreadRAII& operator=(ThreadRAII&&) = default;     // moving

    std::thread& get() { return t; }                   // as before
private:                                               // as before
    DtorAction action;
    std::thread t;
};
```

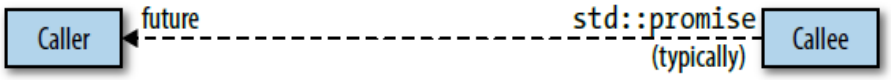
需要铭记的要点
● 使 std::thread 在所有路径上不可连接。 ● 销毁时连接（join-on-destruction）可能导致难以调试的性能异常。 ● 销毁时分离（detach-on-destruction）会导致难以调试的不确定行为。 ● std::thread 对象声明放在数据成员列表的最后。

条款 38：注意不同的线程句柄析构行为

条款 37 解释了可连接的 std::thread 对应于底层系统执行线程。非延迟任务的 future(见条款 36)与系统线程的关系类似。因此，可以将 std::thread 对象和 future 对象都看作是系统线程的句柄。

从这个角度来看，有趣的是，std::thread 和 future 的析构行为却又如此不同。如条款 37 所述，销毁可连接的 std::thread 会直接终止程序，因为两种明显的可选方案——隐式 join 和隐式 detach——都更糟糕。然而，future 的析构函数有时表现得好像做了隐式 join，有时又表现得好像做了隐式 detach，有时又两者都不做。它永远不会导致程序终止。这个东东还是值得进一步研究的。

我们开始吧！future 是某种通信通道的一端，通过这种通信通道，被调用者可以将结果传递给调用者。被调用者(通常异步运行)将其计算结果写入通信通道 (通常通过 std::promise 对象)，调用者使用 future 读取结果。如下图所示：

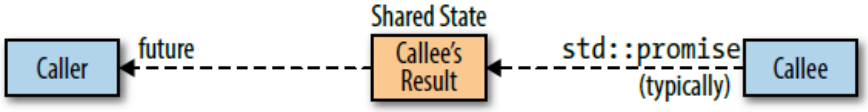


但是被调用者的结果存储在哪里呢？被调用者可能在调用者调用 future 的 get 之前结束，因此结果不能存储在被调用者的 std::promise 中，这是被调用者的本地对象，在被调用者结束时会销毁。

结果也不能存储在调用者的 future 中，因为(其他某种原因)std::future 可能用于创建 std::shared_future(从而将被调用者结果的所有权从 std::future 转移到 std::shared_future)，然后在销毁原始 std::future 之后，std::shared_future 可能被多次拷贝。鉴于并非所有结果类型都可以拷贝(即 move-only 类型)，并且结果存活时间必须与引用它的最后一个 future 一样长，那么在众多潜在的 future 对象中，到底哪个应该保存对应被调用者的结果呢？

因为无论是调用者还是被调用者，它们关联的对象都不适合存储被调用者的结果，所以存储位置在两者之外，这个位置称为共享状态（shared state）。共享状态通常由基于堆的对象表示，但是标准并未指定它的类型、接口和实现。标准库作者可以以他们喜欢的任何方式自由地实现。

我们可以设想被调用者、调用者和共享状态之间的关系如下：



共享状态的存在很重要，因为 future 的析构(本文的主题)的行为取决于其关联的共享状态。特别是，

- 通过 std::async 启动的非延迟任务，引用其共享状态的最后一个 future，析构函数会阻塞，直到任务完成。本质上来讲，此类 future 的析构函数在异步执行任务的线程上执行了隐式 join。
- 其他所有 future 的析构函数只是简单地销毁 future 对象。对于异步运行的任务，这类似于对底层线程的隐式 detach。对于延迟任务，这是最终的 future，那就意味着延迟任务永远不会执行。

这些规则听起来比实际上复杂。我们真正要处理的是一个简单的“正常”行为和一个唯一的例外情况。“正常行为”是，future 的析构函数销毁 future 对象。就是这么简单。它不 join 任何东东，不 detach 任何东东，不运行任何东东。它只销毁 future 的数据成员。(实际上，它还干了一件事，递减了共享状态的引用计数。这个引用计数使库能够知道何时可以销毁共享状态。有关引用计数的通用信息，见条款 19。)

只有满足下面所有条件的 future，才会出现例外情况：

- 它引用的共享状态是由于调用 std::async 而创建的。
- 任务的启动策略是 std::launch::async(参见条款 36)，要么是运行时系统自动选择的，要么是在调用 std::async 时指定的。
- 它是指向共享状态的最后一个 future。对于 std::future 来说，它总是最后一个。对于 std::shared_future 来说，如果还有其他 std::shared_future 引用共享状态，那么 future 的销毁遵循正常行为(即，只简单地销毁它的数据成员)。

只有当所有这些条件都满足时，future 的析构才会显示出特殊行为，也就是阻塞直到异步运行的任务完成。实际上，这相当于隐式 join 任务线程。

我们经常听到，这个例外被概括为“从 std::async 返回的 future 阻塞在它们的析构函数。”大致来说，这是正确的，但有时你需要的不仅仅是个大致概念，而现在你已经完全掌握了它的内涵。

你可能会有个疑问，类似于“我想知道，对于由 std::async 启动的非延迟任务，为什么

它的共享状态有一个特殊规则”。这是一个合理的问题。据我所知，标准委员会希望通过使用隐式 detach(见条款 37)来避免此类情况，但他们又不希望采用像强制终止程序那样的激进策略 (就像析构可连接 std::thread 那样，同样见条款 37)，所以他们退而求其次，采用隐式 join。这个决定并不是没有争议的，而且对于 C++14 是否放弃这种行为进行了很多严肃讨论。但最后没有做任何改变，所以在 C++11 和 C++14 中，future 析构函数的行为是一致的。

future 的 API 无法确定 future 是否引用了共享状态，因此对于任意 future 对象，不可能知道它的析构函数是否会阻塞，等待异步运行任务结束。这就有一些有趣的暗示：

```
// futs可能在析构函数阻塞，
// 因为一个或多个future可能指向通过std::async启动的非延迟任务的共享状态
std::vector<std::future<void>> futs;  // std::future<void>见条款39

class Widget {                        // Widget对象可能在析构函数阻塞
public:
    ...
private:
    std::shared_future<double> fut;
};
```

当然，如果你知道给定的 future 不满足触发特殊析构函数行为的条件(例如，由于程序逻辑的原因)，则可以确保该 future 不会在析构函数阻塞。例如，只有从 std::async 调用产生的共享状态才符合特殊行为，但是创建共享状态还有其他方法，一种是使用 std::packaged_task。packaged_tas 对异步执行的函数(或其他可调用对象)进行包装，使其结果存入共享状态，引用这种共享状态的 future 可以通过 std::packaged_task 的 get_future 函数获取：

```
int calcValue();              // func to run

std::packaged_task<int()>     // wrap calcValue so it
    pt(calcValue);            // can run asynchronously

auto fut = pt.get_future();   // get future for pt
```

此处，我们知道 fut 没有引用 std::async 调用创建的共享状态，因此它的析构函数将正常工作。

一旦创建，std::packaged_task pt 就可以在线程上运行。(也可以通过调用 std::async 来运行，但是如果你想使用 std::async 来运行一个任务，那么几乎没有理由再创建

std::packaged_task，因为 std::async 在调度任务执行之前完成了 std::packaged_task 所做的一切工作。)

packaged_task 不可拷贝，因此当 pt 传递给 std::thread 构造函数时，它必须转换为右值(通过 std:: move，见条款 23)：

```
std::thread t(std::move(pt)); // run pt on t
```

这个例子让我们对 future 析构函数的正常行为有了一些了解，但是把上面的语句放在一个代码块中，更容易看明白：

```
{                                   // begin block
    std::packaged_task<int()>

    pt(calcValue);
        auto fut = pt.get_future();


    std::thread t(std::move(pt));

    ...                             // see below
}                                   // end block
```

这里最有趣的代码是“...”，它位于 std::thread 对象 t 的创建之后，位于块的末尾之前。有趣的是 t 在“...”区域内会发生什么。有三种基本可能：

- **没有任何操作**。在这种情况下，t 在作用域结束时是可连接的，这将导致程序终止(见条款 37)。
- **执行 join**。在这种情况下，fut 不需要在它的析构函数中阻塞，因为 join 已经出现在调用代码中。
- **执行 detach**。在这种情况下，fut 不需要在它的析构函数中 detach，因为调用代码已经做到了这一点。

换句话说,当你有一个 future 关联到由 std:: packaged_task 产生的共享状态，通常不需要采取特殊策略，因为操作 std::thread 的代码会决定到底是终止、join 还是 detach。

需要铭记的要点
● future 的析构函数正常是只销毁它的数据成员。 ● 由 std::async 启动的非延迟任务的共享状态，引用它的最后一个 future 会阻塞，直到任务结束。

条款 39：考虑在一次性事件通信上使用 void futures

有时，对于一个任务来说，告诉另一个异步运行的任务某个特定事件已经发生是很有用的，因为第二个任务在事件发生之前不能继续前行。这种事件也许是初始化了一个数据结构，完成了一个计算阶段，或者检测到了一个重要的传感器值。在这种情况下，进行这种线程间通信的最佳方式是什么？

一个明显的方法是使用条件变量(condvar)。我们将检测条件的任务成为检测任务（detecting task），将对条件作出反应的任务成为响应任务（reacting task），策略很简单：响应任务等待条件变量，检测任务在事件发生时通知 condvar。如下：

```
std::condition_variable cv;   // condvar for event
std::mutex m;                 // mutex for use with cv
```

检测任务的代码就像下面这么简单：

```
…                             // detect event
cv.notify_one();              // tell reacting task
```

如果有多个响应任务需要通知，就用 notify_all 替换 notify_one，但是现在，我们假设只有一个响应任务。

响应任务的代码稍微复杂一些，因为在调用 condvar 上的 wait 之前，它必须通过 std::unique_lock 对象锁定互斥锁。(在等待条件变量之前锁定互斥锁是线程库的典型做法。通过 std::unique_lock 对象锁定互斥锁，这个要求只是 C++11 API 的一部分)下面是一个概念上的实现：

```
…                                       // 准备响应

{                                       // 进入临界区
    std::unique_lock<std::mutex> lk(m);  // lock mutex

    cv.wait(lk);                        // 等待通知
                                        // this isn't correct!

    …                                   // react to event
                                        // (m is locked)

}                                       // 离开临界区；通过析构unlock m

…                                       // continue reacting
                                        // (m now unlocked)
```

这种方法的第一个问题是有时被称为代码味道（code smell）的东东：即使代码工作，有些东西看起来也不太正确。在这种情况下，问题来自互斥锁的使用。互斥锁用于控制对共享数据的访问，但是完全有可能，检测和响应任务不需要这样的中介。例如，检测任务可能负责初始化一个全局数据结构，然后把它交给响应任务使用。如果初始化数据结构后，检测任务永远不会访问它，并且响应任务在检测任务指示它准备就绪之前也绝不访问它，那么这两个任务就可以通过程序逻辑相互避开。这样就不需要互斥锁了。

即使你忽略了这一点，你也应该注意另外两个问题：

- 如果检测任务在响应任务等待之前通知 condvar，响应任务将挂起。为了通知 condvar 唤醒另一个任务，另一个任务必须等待该 condvar。如果检测任务恰好在响应任务执行等待之前执行通知，响应任务将错过通知，并永远等待下去。
- 等待语句无法说明伪唤醒（spurious wakeup）。线程 API(在许多语言中——不仅仅是 C++)中存在的一个事实是，即使没有通知 condvar，等待条件变量的代码也可能被唤醒。这种唤醒被称为伪唤醒。适当的代码可以通过确认所等待的条件是否确实发生了来处理这种情况，并将此作为唤醒后的第一个动作。C++ condvar API 实现起来特别容易，因为它允许用 lambda(或其他函数对象)测试传递给 wait 的等待条件。也就是说，响应任务的 wait 调用可以写成下面这样：

  ```
  cv.wait(lk,
      []{ return whether the event has occurred; });
  ```

  利用这种能力要求响应任务能够确定它等待的条件是否为真。但是在我们考虑的场景中，它等待的条件是检测线程负责识别的事件的发生。响应线程可能无法确定它等待的事件是否已经发生。这就是它等待条件变量的原因！

在许多情况下，使用 condvar 进行任务通信非常适合解决手头的问题，但上面这种情况似乎不适合。

对于许多开发人员来说，另一个技巧是共享布尔标志。这个标志最初是 false。当检测线程识别出它正在寻找的事件时，它设置标志为 true：

```
std::atomic<bool> flag(false);    // shared flag; see
                                  // Item 40 for std::atomic

...                               // detect event

flag = true;                      // tell reacting task
```

响应线程只是简单的轮询标志。当它看到标志为 true 时，它知道等待的事件已经发生：

```
…                   // prepare to react

while (!flag);      // wait for event

…                   // react to event
```

这种方法没有任何基于 condvar 设计的缺点。不需要互斥锁，如果检测任务在响应任务开始轮询之前设置了标志也没有问题，而且没有类似于伪唤醒的东西。赞、赞、赞。

不足之处就是响应任务中轮询的成本。在等待标志的过程中，任务基本上是阻塞的，但它仍在运行。因此，它占用了另一个任务可能可以使用的硬件线程，每次启动或完成它的时间片时，都带来上下文切换的成本，并且它还使得 CPU 核心持续运行，本来是可以关闭掉从而节能电力的。一个真正阻塞的任务是不会做这些事情的。这是基于 condvar 方法的优点，因为 wait 调用中的任务真正阻塞了。

我们通常将 condvar 和基于标志的设计组合使用。标志指示感兴趣的事件是否已经发生，而对标志的访问是由互斥锁同步的。由于互斥锁阻止对标志的并发访问，正如条款 40 所解释的，不需要将标志声明为 std::atomic，一个简单的 bool 就可以了。检测任务可以像下面这样：

```
std::condition_variable cv;             // as before
std::mutex m;

bool flag(false);                       // not std::atomic

…                                       // detect event

{
    std::lock_guard<std::mutex> g(m); // lock m via g's ctor

    flag = true;                        // tell reacting task (part 1)

}                                       // unlock m via g's dtor


cv.notify_one();                        // tell reacting task (part 2)
```

而响应任务如下：

```
…                                       // prepare to react

{                                       // as before
```

```
        std::unique_lock<std::mutex> lk(m);   // as before

        cv.wait(lk, [] { return flag; });     // use lambda to avoid
                                              // spurious wakeups

        ...                                   // react to event
                                              // (m is locked)
    }

    ...                                       // continue reacting
                                              // (m now unlocked)
```

这种方法避免了我们已经讨论过的问题。无论响应任务是否在检测任务通知之前等待，它都可以工作，而且它在出现伪唤醒时也可以工作，并且不需要轮询。然而怪味仍然存在，因为检测任务以一种非常奇怪的方式与响应任务进行通信。通知条件变量会告诉响应任务它一直等待的事件可能已经发生，但是响应任务必须检查标志进行确认；设置标志会告诉响应任务事件确实已经发生，但检测任务仍然必须通知条件变量，以便响应任务被唤醒并检查标志。这种方法是可行的，但看起来并不十分清晰。

另一种选择是让响应任务 wait 由检测任务设置的 future，从而避免使用条件变量、互斥锁和标志。这似乎是个奇怪的想法。毕竟，条款 38 解释了 future 表征的是从被调用者到(异步)调用者的通信通道的接收端，在这里，检测和响应任务之间没有被调用者-调用者关系。不过，条款 38 还提到，发送端为 std::promise 且接收端为 future 的通信信道不仅可以用于被调用者-调用者通信，这种通信通道可以用于任何需要将信息从程序中的一个位置传输到另一个位置的情况。在这种情况下，我们可以使用它将信息从检测任务传输到响应任务，我们要传递的信息就是感兴趣的事件已经发生。

设计很简单。检测任务有一个 std::promise 对象(即，通信通道的写入端)，而响应任务有一个对应的 future。当检测任务发现事件已经发生时，它设置 std::promise(即，写入通信通道)。与此同时，响应任务 wait 它的 future，这种 wait 会阻塞响应任务，直到设置了 std::promise。

现在，std::promise 和 futures (即，std::future 和 std::shared_future)两个都是需要类型参数的模板，该参数表明通过通信通道传输的数据类型。然而，在我们的例子中，没有数据需要传递。响应任务唯一感兴趣的是，它的 future 已经被设置过。对于 std::promise 和 future 模板，我们需要的是一种类型，它表明不需要通过通信通道传递数据，那个类型就是 void。因此，检测任务使用 std::promise<void>，而响应任务使用 std::future<void> 或

std::shared_future<void>。检测任务在事件发生时设置其std::promise<void>，响应任务wait其 future。尽管响应任务不会从检测任务接收到任何数据，但是通过在检测任务的std::promise 上调用 set_value，通信通道允许响应任务知道检测任务何时“写入”了它的 void。

所以，给定

```
std::promise<void> p; // promise for communications channel
```

检测任务的代码就很简单，

```
…                   // detect event

p.set_value();      // tell reacting task
```

并且，响应任务的代码也同样简单：

```
…                       // prepare to react

p.get_future().wait();  // wait on future corresponding to p

…                       // react to event
```

就像使用标志的方法一样，这种设计不需要互斥量，不管检测任务是否在响应任务等待之前设置了 std::promise，它都能正常工作，并且不受伪唤醒的影响。(只有条件变量才容易受到这个问题的影响。)与基于 condvar 的方法一样，响应任务在发出等待调用后确实被阻塞，因此在等待时不消耗系统资源。完美，对吗？

不完全是。当然，基于 future 的方法可以避开这些问题，但是还有其他危险存在。例如，条款 38 解释了 std::promise 和 future 之间的是共享状态，共享状态通常是动态分配的。因此，你应该假设这种设计会带来基于堆的分配和回收成本。

也许更重要的是，std::promise 只能设置一次。std::promise 和 future 之间的通信通道是一个一次性的机制：它不能重复使用。这与基于 condvar 和基于标志的设计有显著区别，这两种设计都可以用于多次通信。(condvar 可以被多次通知，而标志总是可以被清除并重新设置。)

一次性约束并不像你想的那么作用有限。假设你希望创建处于挂起状态的系统线程，也就是说，你希望消除与线程创建相关的所有开销，以便当你准备在线程上执行某些操作时，可以避免正常的线程创建延迟。或者你可能希望创建一个挂起的线程，以便在让它运行之前对其进行配置，这种配置可能包括设置其优先级或核心亲属（core affinity）等内容。C++并发 API 没有提供实现这些功能的方法，但是 std::thread 对象提供了 native_handle 成员函数，其结果可以让你访问平台的底层线程 API(通常是 POSIX 线程或 Windows 线程)。低层

级 API 通常可以配置线程特性，例如优先级和亲属（affinity）。

假设你只想挂起某个线程一次(在创建之后，但是在它运行线程函数之前)，那么使用 void future 是一个合理的选择。下面是这项技术的精髓：

```
std::promise<void> p;

void react();                                   // func for reacting task

void detect()                                   // func for detecting task
{
    std::thread t([]                            // create thread
                  {
                      p.get_future().wait();    // suspend t until
                      react();                  // future is set
                  });

    ...                                         // here, t is suspended
                                                // prior to call to react

    p.set_value();                              // unsuspend t (and thus
                                                // call react)

    ...                                         // do additional work

    t.join();                                   // make t unjoinable
}                                               // (see Item 37)
```

因为重要的是，t 在 detect 的所有路径上都要是不可连接的。所以使用像条款 37 的 ThreadRAII 那样的 RAII 类，似乎是可取的。这样的代码出现在脑海中：

```
void detect()
{
    ThreadRAII tr(                              // use RAII object
    std::thread([]
                {
                    p.get_future().wait();
                    react();
                }),
                ThreadRAII::DtorAction::join    // risky! (see below)
               );

    ...                                         // thread inside tr
                                                // is suspended here
```

```
        p.set_value();                                  // unsuspend thread
                                                        // inside tr
        ...
    }
```

这看起来比实际更安全。问题是，如果在第一个“...”区域(带有“thread inside tr is suspended here”注释的区域)，出现异常，set_value 将永远不会被调用。这意味着在 lambda 中 wait 调用将永远不会返回。这又意味着运行 lambda 的线程永远不会结束，这是一个问题，因为 RAII 对象 tr 已经配置为在 tr 的析构函数中对该线程执行 join。换句话说，如果从代码的第一个“...”区域发出异常，该函数将挂起，因为 tr 的析构函数永远不会完成。

有很多方法可以解决这个问题，但是我把它们作为神圣的练习留给读者。在这里，我想展示一下如何将原始代码(即，不使用 ThreadRAII)扩展为可以挂起并取消挂起不止一个响应任务。这是一个简单的泛化，因为关键是在 react 代码中使用 std::shared_futures 而不是 std::future。一旦你知道 std::future 的 share 成员函数将其共享状态的所有权转移到由 share 生成的 std::shared_future 对象，代码就几乎顺理成章了。唯一的微妙之处在于，每个响应线程需要自己的 std::shared_future 副本，该副本引用共享状态，因此从 share 获得的 std::shared_future 由运行在响应线程上的 lambdas 按值捕获：

```
std::promise<void> p;                               // as before

void detect()                                       // 支持多个响应任务
{
    auto sf = p.get_future().share();               // sf类型是
                                                    // std::shared_future<void>

    std::vector<std::thread> vt;                    // 响应线程的容器

    for (int i = 0; i < threadsToRun; ++i) {
        vt.emplace_back([sf]{ sf.wait();            // 在sf的本地拷贝上wait;
                              react(); });          // see Item 42
    }                                               // for info on emplace_back

    ...                                             // 如果"..." 抛出异常，detect挂起

    p.set_value();                                  // unsuspend all threads

    ...

    for (auto& t : vt) {                    // make all threads unjoinable;
        t.join();                           // see Item 2 for info on "auto&"
```

```
        }
    }
```

使用 futures 的设计可以实现这种效果，这一点值得注意，这就是为什么你应该考虑将其用于一次性事件通信。

需要铭记的要点
● 对于简单的事件通信，基于 condvar 的设计需要一个多余的互斥量，对检测和响应任务进行约束，并且需要响应任务验证事件是否已经发生。 ● 使用标志的设计避免了上面那个问题，但是基于轮询，而不是阻塞。 ● condvar 和 flag 可以一起使用，但由此产生的通信机制有些笨拙。 ● 使用 std::promise 和 futures 可以避免上面那些问题，但是这种方法对于共享状态用的是堆内存，并且仅限于一次通信。

条款 40：并发使用 std::atomic，特殊内存使用 volatile

可怜的 volatile，如此被误解，它甚至不应该出现在本章中，因为它与并发编程无关。但是在其他编程语言中 (如，Java 和 C#)，它对这类编程很有用，甚至在 C++中，一些编译器也为 volatile 注入了语义，使其适用于并发软件(但只有在使用这些编译器编译时)。因此，如果只是为了消除围绕 volatile 的混乱，那么在关于并发性的一章中讨论 volatile 是值得的。

程序员有时会与 volatile 混淆的 C++特性是 std::atomic 模板(这一特性肯定属于本章)。该模板的实例化(如，std::atomic<int>，std::atomic<bool>，std::atomic<Widget*>，等等)提供了线程间原子操作。一旦构造了 std::atomic 对象，对它的操作就会表现得好像它们在互斥保护的临界区中一样，但是这些操作通常使用特殊的机器指令来实现，这些指令比互斥锁更高效。

考虑使用 std::atomic:的如下代码：

```
std::atomic<int> ai(0);  // initialize ai to 0

ai = 10;                 // atomically set ai to 10

std::cout << ai;         // atomically read ai's value

++ai;                    // atomically increment ai to 11

--ai;                    // atomically decrement ai to 10
```

在执行这些语句期间，读取 ai 的其他线程可能只看到 0、10 或 11 的值。没有其他可能的值 (当然，假设这是唯一修改 ai 的线程)。

这个例子有两个方面值得注意。首先，在“std::cout << ai;”语句中，ai 是 std::atomic 只保证 ai 的读取是原子的，不能保证整个语句以原子方式进行。在读取 ai 的值和调用 operator<<将其写入标准输出之间，另一个线程可能修改了 ai 的值。这对语句的行为没有影响，因为 operator<<对 int 使用的是传值(因此输出的值就是从 ai 读取的值)，但重要的是要明白，那个语句里面，只有 ai 的读取是原子的。

第二个值得注意的方面是前两个语句的行为——ai 的递增和递减。它们都是读-修改-写（read-modify-write）（RMW）操作，但是它们是原子执行的。这是 std::atomic 类型最优秀的特性之一:一旦构造了 std::atomic 对象，它的所有成员函数，包括那些包含 RMW 操作的，对其他线程来说都是原子的。

相反，使用 volatile 的相应代码在多线程上下文中几乎什么都不能保证：

```
volatile int vi(0);  // initialize vi to 0

vi = 10;             // set vi to 10

std::cout << vi;     // read vi's value

++vi;                // increment vi to 11

--vi;                // decrement vi to 10
```

在执行这段代码时，如果其他线程正在读取 vi 的值，它们可能会看到诸如-12、68 、4090727 等任意数！这样的代码会有未定义的行为，因为这些语句会修改 vi，所以如果其他线程同时读取 vi，那么这块内存会同时存在读和写，而这个内存既不是 std::atomic，也不受互斥锁保护，这就是数据竞争。

要阐述多线程程序中 std::atomic 和 volatile 的行为如何不同，有个具体例子，请考虑这两个类型的简单计数器，该计数器由多个线程递增。我们将其初始化为 0：

```
std::atomic<int> ac(0);   // "atomic counter"
volatile int vc(0);       // "volatile counter"
```

然后，我们在两个同时运行的线程中增加每个计数器一次：

```
/*----- Thread 1 ----- */ /*------- Thread 2 ------- */
        ++ac;                       ++ac;
        ++vc;                       ++vc;
```

当两个线程都完成时，ac 的值(即， std::atomic)的值必定为 2，因为每个递增都是不可分割的操作。另一方面，vc 的值不一定是 2，因为它的递增可能不是原子发生的。每个递增包括读取 vc 的值，递增被读取的值，并将结果写回 vc。但这三个操作并不能保证对 volatile 对象进行原子化处理，所以 vc 的两个递增的组成部分可能是交错的，如下所示：

1. 线程 1 读取 vc 的值，即 0。
2. 线程 2 读取 vc 的值，仍然是 0。
3. 线程 1 将读取的 0 增加到 1，然后将该值写入 vc。
4. 线程 2 将读取的 0 增加到 1，然后将该值写入 vc。

因此 vc 的最终值是 1，尽管它递增了两次。

这不是唯一可能的结果。一般来说，vc 的最终值是不可预测的，因为 vc 涉及了一场数据竞争，而标准认为数据竞争会导致未定义的行为，这意味着编译器可以生成代码来做任何

事情。当然，编译器不会利用这种灵活性进行恶意操作。相反，它们会执行那些对无数据竞争的程序来说有效的优化，而这些优化在存在竞争的程序中会产生意外和不可预测的行为。

RMW 操作并不是 std::atomic 并发成功而 volatile 失败的唯一情况。假设一个任务计算第二个任务所需的重要值。当第一个任务计算完该值后，它必须将该值传递给第二个任务。条款 39 解释了，一种可用的通信方式是 std::atomic<bool>，任务中计算值的代码应该是这样的：

```
std::atomic<bool> valAvailable(false);

auto imptValue = computeImportantValue(); // compute value

valAvailable = true;                      // tell other task
                                          // it's available
```

当人们阅读这段代码时，我们知道 imptValue 在 valAvailable 之前赋值是至关重要的，但是所有编译器看到的都是一对独立变量的赋值。一般来说，编译器允许对这些不相关的赋值进行重新排序。也就是说，给定如下赋值序列 (a、b、x、y 对应独立变量)，

```
a = b;
x = y;
```

编译器可能按如下顺序重新排序：

```
x = y;
a = b;
```

即使编译器不重新排序，底层硬件也可能会这样做(或者可能会让其他核心觉得它是这样做的)，因为这有时会让代码运行得更快。

然而，std::atomic 的使用对代码可以怎样重新排序作了限制，其中一个限制是，在源代码中，std::atomic 变量写操作之前的任何代码，都不能在写操作后面执行(或者在其他核心中执行)。这意味着在我们的代码中，

```
auto imptValue = computeImportantValue(); // compute value

valAvailable = true;                      // tell other task
                                          // it's available
```

编译器不仅要保持 imptValue 和 valAvailable 的赋值顺序，同时生成的代码也必须确保底层硬件这样做。因此，将 valAvailable 声明为 std::atomic 可以确保我们的关键顺序得到保持，即所有线程必须看到 imptvalue 不会在 valAvailable 之后改变。

声明 valAvailable 为 volatile 就没了代码重新排序限制：

```
volatile bool valAvailable(false);
```

```
auto imptValue = computeImportantValue();
valAvailable = true;          // other threads might see this assignment
                              // before the one to imptValue!
```

此处，编译器可能会颠倒 imptValue 和 valAvail 的赋值顺序，即使没有颠倒，也可能无法生成那样的机器码，阻止底层硬件在其他核心上的代码可能在 imptValue 之前看到 valAvailable 改变。

这两个问题——没有操作原子性的保证和对代码重新排序的限制不足——解释了为什么 volatile 对并发编程没有用处，但是它没有解释它的用处。简而言之，它用于告诉编译器它们正在处理行为不正常的内存。

“正常”内存的特征是，如果你将一个值写入内存位置，该值将一直保持在那里，直到有东西覆盖它。所以，如果有一个“正常”int，

```
int x;
```

并且编译器看到下面的操作序列，

```
auto y = x; // read x
y = x;      // read x again
```

编译器可以通过剔除对 y 的赋值来优化生成的代码，因为它与 y 的初始化是冗余的。

“正常”内存还具有这样一个特性，如果你将一个值写入内存位置，但从未读取该值，然后再次写入该内存位置，则可以剔除第一次写入，因为它从未使用过。所以，已知这两个相邻的语句，

```
x = 10;      // write x
x = 20;      // write x again
```

编译器可以剔除第一个。这意味着如果我们在源代码中有这些，

```
auto y = x;  // read x
y = x;       // read x again
x = 10;      // write x
x = 20;      // write x again
```

编译器可以当做如下处理：

```
auto y = x; // read x
x = 20;     // write x
```

为了避免你怀疑谁会编写执行这种冗余读取和不必要的写入（技术上称为冗余加载(redundant loads)和废弃存储(dead stores) ）的代码，告诉你答案是，人类不会直接编写它——至少我们希望不会。然而，在编译器获得看起来合理的源代码并执行模板实例化、内联

和各种常见的重新排序优化之后，这种结果并不少见。

这种优化只有在内存表现正常时才恰当。“特殊”内存不恰当。最常见的一种特殊内存可能是用于内存映射（memory-mapped）I/O 的内存。这些内存中的位置实际上与外围设备通信，例如外部传感器或显示器、打印机、网络端口等，而不是读取或写入正常内存(即 RAM)。在这种上下文中，再考虑那些看起来冗余的代码：

```
auto y = x; // read x
y = x;      // read x again
```

例如，如果 x 对应于温度传感器报告的值，那么 x 的第二次读取就不是多余的，因为温度可能在第一次和第二次读取之间发生了变化。

对于看似多余的写入来说，情况也是类似的。例如，在这段代码中，

```
x = 10; // write x
x = 20; // write x again
```

如果 x 对应于无线电发射机的控制端口，则可能是代码向无线电发出命令，值 10 对应的命令与值 20 不同。优化剔除第一个任务将改变发送到无线电的命令序列。

volatile 是我们告诉编译器我们正在处理特殊内存的一种方式。它对编译器的意义是“不要对这个内存上的操作执行任何优化”。因此，如果 x 对应于特殊内存，它将被声明为 volatile：

```
volatile int x;
```

考虑一下对前面代码序列的影响：

```
auto y = x;      // read x
y = x;           // read x again (can't be optimized away)
x = 10;          // write x (can't be optimized away)
x = 20;          // write x again
```

这正是我们想要的，如果 x 是内存映射的(或者已经映射到进程间共享的内存位置，等等)。

突击测验！在最后一段代码中，y 的类型是什么：int 还是 volatile int？

在处理特殊内存时，必须保留表面上的冗余负载和废弃存储，这一事实解释了为什么 std::atomic 不适合这种工作。编译器可以剔除 std::atomic 上的这种冗余操作。std::atomic 的代码写出来与 volatile 不太一样，但是如果我们暂时忽略这一点，只关注编译器可以做什么，我们可以说，从概念上讲，编译器可能得到下面这样的代码，

```
std::atomic<int> x;

auto y = x;  // conceptually read x (see below)
```

```
y = x;  // conceptually read x again (see below)

x = 10; // write x
x = 20; // write x again
```

并且优化成这样：

```
auto y = x; // conceptually read x (see below)
x = 20;     // write x
```

对于特殊内存，这显然是不可接受的行为。

现在，碰到的是，当 x 是 std::atomic 时，这两个语句都无法编译通过：

```
auto y = x;  // error!
y = x;       // error!
```

这是因为 std::atomic 的复制操作被删除了(参见条款 11)。这是有充分理由的。考虑一下如果用 x 初始化 y 可以编译通过。因为 x 是 std::atomic，所以 y 的类型也推导为 std::atomic (见条款 2)。我之前说过，std::atomic 最好的事情之一是，它们所有的操作都是原子的，但是为了确保从 x 到 y 的拷贝构造是原子的，编译器生成的代码，必须保证读取 x 和写入 y 在一个单一的原子操作中。硬件通常无法做到这一点，所以不支持 std::atomic 类型的拷贝构造。出于同样的原因删除了拷贝赋值，这就是为什么从 x 到 y 的赋值无法编译通过。(移动操作没有明确声明，所以，根据条款 17 描述的编译器生成特殊函数的规则，std::atomic 也不提供移动构造和移动赋值)

将 x 的值赋给 y 是有可能的，但是它需要使用 std::atomic 的 load 和 store 成员函数。load 成员函数以原子方式读取 std::atomic 的值，而 store 成员函数以原子方式写入。要用 x 初始化 y，然后把 x 的值代入 y，代码必须这样写：

```
std::atomic<int> y(x.load()); // read x
y.store(x.load());            // read x again
```

这是可以编译的，但是读取 x(通过 x.load())是与初始化或存入 y 分开的函数调用，这表明，没有理由期望这两个语句作为一个整体按照单个原子操作执行。

这个代码，编译器可以通过将 x 的值存储在寄存器中来“优化”，而不是读取两次 x：

```
register = x.load();          // read x into register
std::atomic<int> y(register); // init y with register value
y.store(register);            // store register value into y
```

结果，正如你所看到的，只从 x 读取一次，在处理特殊内存时必须避免这种优化。(对 volatile 变量，这种优化是不允许的。)

因此，情况应该很明朗了：

- atomic 对于并发编程是有用的，但是对于访问特殊内存是没用的。
- volatile 对访问特殊内存有用，但不适用于并发编程。

因为 std::atomic 和 volatile 用途不同，它们甚至可以一起使用：

```
volatile std::atomic<int> vai; // operations on vai are
                               // atomic and can't be
                               // optimized away
```

如果 vai 对应于多个线程并发访问的内存映射 I/O 位置，那么这可能非常有用。

最后要注意的是，一些开发人员更喜欢使用 std::atomic 的 load 和 store 成员函数，即使在不必要的时候，因为它在源代码中明确表示所涉及的变量不是“正常的”。强调这一事实并非没有道理。访问 std::atomic 通常比访问非 std::atomic 慢得多，而且我们已经看到 std::atomic 的使用阻止编译器执行某些类型的代码重排。因此，调用 std::atomic 的 load 和 store 可以帮助识别潜在的可伸缩性瓶颈。从正确性的角度来看，对于一个将信息传递给其他线程的变量(如，一个指示数据可用性的标志)，没有看到 store 调用，可能意味着该变量没有在应该声明 std::atomic 的时候声明 std::atomic。

不过，这在很大程度上是一个风格问题，并且同样的是，std::atomic 和 volatile 之间的选择相当不同。

需要铭记的要点
● atomic 用于被多个线程访问并且没有互斥锁的数据。它是一个编写并发软件的工具。 ● volatile 是用于不应该优化剔除读写的内存。它是一种处理特殊内存的工具。

第八章改进

对于 C++中的每一种通用技术或特性，都有合理使用它的情况，也有不合理使用它的情况。描述什么时候值得使用通用技术或特性，通常是相当简单的，但是本章包含两个例外。通用技术是按值传递，通用特征是放置（emplacement）。何时使用它们，受到很多因素影响，我能提供的最好的建议是仔细考虑它们的使用。尽管如此，它们都是高效现代 C++编程中的重要角色，并且下面的条款提供了你需要的信息，以确定它们是否适合你的软件。

条款 41：当 move 操作代价低并始终需要副本时，对可拷贝形参考虑传值

有些函数参数就是要被拷贝的。例如，成员函数 addName 可能将其参数拷贝到私有容器中。为了提高效率，这样的函数应该拷贝左值实参，但是移动右值实参：

```
class Widget {
public:
    void addName(const std::string& newName)  // take lvalue;
    { names.push_back(newName); }              // copy it

    void addName(std::string&& newName)        // take rvalue;
    { names.push_back(std::move(newName)); }   // move it; see
    …                                          // Item 25 for use
                                               // of std::move
private:
    std::vector<std::string> names;
};
```

这是可行的，但它需要编写两个函数，它们的功能本质上是相同的。这有点让人恼火：需要声明两个函数，需要实现两个函数，需要记载两个函数，需要维护两个函数。啊!!

此外，目标代码中还存在两个函数——如果你关心程序的占用空间，你可能会关心这两个函数。在这个例子中，这两个函数可能都是内联的，而这会消除与两个函数相关的膨胀问题，但是如果这些函数不是处处内联的，那么在你的目标代码中确实会存在两个函数。

另一种方法是使 addName 成为一个具有通用引用的函数模板(见条款 24)：

```
class Widget {
public:
    template<typename T>                       // take lvalues
```

```
    void addName(T&& newName)                    // and rvalues;
    {                                            // copy lvalues,
        names.push_back(std::forward<T>(newName));// move rvalues;
    }                                            // see Item 25
                                                 // for use of
    ...                                          // std::forward
};
```

这减少了源代码，但是通用引用会带来其他复杂性。作为模板，addName 的实现通常必须位于头文件中。它可能在目标代码中产生好几个函数，因为它不仅对 lvalues 和 rvalues 实例化不同，而且对 std::string 和可转换为 std::string 的类型实例化也不同(参见条款 25)。同时，有些实参类型不能通过通用引用传递(请参阅条款 30)，并且如果用户传递了不恰当的实参类型，那么编译器的错误消息可能令人生畏(参见条款 27)。

如果有一种方法可以编写类似 addName 这样的函数，既可以复制 lvalues，又可以移动 rvalues，而且在源代码和目标代码中都只需要处理一个函数，并且可以避免通用引用的特殊问题，那不是很爽吗？事实上，有这样的方式。你所要做的就是放弃作为 C++程序员可能学到的第一个规则，该规则就是避免按值传递用户自定义类型的对象。对于类似 addName 函数中这种类似 newName 的参数，按值传递可能是一种完全合理的策略。

在我们讨论为什么按值传递可能适合于 newName 和 addName 之前，让我们看看它是如何实现的：

```
class Widget {
public:
    void addName(std::string newName)          // take lvalue or
    { names.push_back(std::move(newName)); }   // rvalue; move it

    ...

};
```

这段代码中唯一不一目了然的部分是 std::move。通常，std::move 是用于右值引用，但在这个例子中，我们知道：(1)无论来自哪个调用者，newName 都是一个完全独立的对象，所以 newName 的变化不会影响到调用者；(2)这是 newName 的最后使用，所以移动不会对函数其他部分有任何影响。

只用了一个 addName 函数，这一情况阐明了，我们如何避免代码重复，无论是在源代码中还是在目标代码中。我们没有使用通用引用，所以这种方法不会导致头文件膨胀、奇怪的失败案例或烦人的错误消息。但是这种设计的效率如何呢？我们用的是按值传递呀，那不是代价大吗？

在 C++ 98 中，的确是存在代价问题，无论调用者传入什么，参数 newName 都将拷贝构造。然而，在 C++11 中，addName 只有对 lvalues 拷贝构造，对于 rvalues，它将移动构造，如下：

```
Widget w;

...

std::string name("Bart");

w.addName(name);              // call addName with lvalue

...

w.addName(name + "Jenne");    // call addName with rvalue (see below)
```

在对 addName 的第一次调用中(传递 name 时)，参数 newName 使用 lvalue 初始化，因此，newName 拷贝构造，就像在 C++ 98 中一样。在第二个调用中，使用 std::string 的 operator+返回的 std::string 对象来初始化 newName，该对象是一个 rvalue，因此 newName 移动构造。

这样就拷贝了 lvalue，移动了 rvalue，就像我们想要的那样。很简洁，是吧？

它的确很简洁，但是有一些事项需要注意。如果我们回顾一下我们考虑过的 addName 的三个版本，就会更容易记住：

```
class Widget {                                  // Approach 1:
public:                                         // overload for
    void addName(const std::string& newName)    // lvalues and
    { names.push_back(newName); }               // rvalues

    void addName(std::string&& newName)
    { names.push_back(std::move(newName)); }

    ...

private:
    std::vector<std::string> names;
};

class Widget {                                  // Approach 2:
public:                                         // use universal
    template<typename T>                        // reference
    void addName(T&& newName)
```

```
    { names.push_back(std::forward<T>(newName)); }

    ...
};

class Widget {                                          // Approach 3:
public:                                                 // pass by value
    void addName(std::string newName)
    { names.push_back(std::move(newName)); }

    ...
};
```

我将前两个版本称为“by-reference 方法”，因为它们都基于引用传递参数。

以下是我们研究过的两个调用场景：

```
Widget w;
...
std::string name("Bart");

w.addName(name);                // pass lvalue
...
w.addName(name + "Jenne");      // pass rvalue
```

现在从拷贝和移动操作两个方面，考虑 addName 三个版本在这两个场景的成本。计算在很大程度上将忽略编译器优化抛弃拷贝和移动操作的可能性，因为这种优化依赖于上下文和编译器，并且在实践中不会改变此次分析的本质。

- **重载**：无论传递的是 lvalue 还是 rvalue，实参都绑定到一个名为 newName 的引用。就拷贝和移动操作而言，这是无代价的。在 lvalue 重载中，将 newName 拷贝进 Widget::names。在 rvalue 重载中，它是移动进去。代价汇总：lvalues 的一次拷贝，rvalues 的一次移动。
- **通用引用**：与重载一样，实参绑定到引用 newName。这是一个无代价的操作。由于使用 std::forward, lvalue std::string 实参被拷贝进 Widget::names 中，而 rvalue std::string 实参被移动进去。std::string 实参的总代价与重载相同：lvalues 的一次拷贝，rvalues 的一次移动。

  条款 25 解释到，如果实参类型不是 std::string，那么它将被转发到 std::string 构造函数，这可能导致需要执行的 std::string 拷贝或移动操作几乎为 0。因此，引用通用引用的函数可以说是唯一高效的。但是，这不会影响本条款的分析，因此我们假

设调用者总是传递 std::string 实参，从而简化问题。

- **按值传递**：无论传递的是 lvalue 还是 rvalue，都必须构造参数 newName。如果传递了 lvalue，则需要进行拷贝构造；如果传递 rvalue，则需要移动构造。在函数体中，newName 被无条件地移动到 Widget::names 中。因此，总代价是 lvalues 的一次拷贝加一次移动，rvalues 的两次移动。与传引用的方法相比，lvalues 和 rvalues 都是多了一个额外的 move。

再看看这个条款的标题：

> 当 move 操作代价低并始终需要副本时，对可拷贝形参考虑传值。

这是有原因的。事实上，有四个原因：

1. 你应该只考虑按值传递。没错，它只需要编写一个函数。没错，它只在目标代码中生成一个函数。没错，它避免了通用引用相关的问题。但是它的代价比其他选择要高，而且，正如我们将在下面看到的，在某些情况下，有些代价我们还没有讨论。
2. 仅对可拷贝形参考虑传值。不符合条件的形参必须是 move-only 类型，因为如果形参不可拷贝，而函数总是要创建副本，则必须通过移动构造来创建（备注：如果有专业术语来区分拷贝构造的副本和移动构造的副本就好了）。回顾一下，按值传递相对于重载的优点是，只需要编写一个函数。但是对于 move-only 类型，不需要为 lvalue 参数提供重载，因为 lvalue 副本需要调用拷贝构造函数，而 move-only 类型的拷贝构造函数是禁用的。这意味着只需要支持 rvalue 参数，在这种情况下，“重载”解决方案只需要一个重载：接受 rvalue 引用的那个重载。

   考虑一个带有 std::unique_ptr<std::string>数据成员和一个 setter 的类。unique_ptr 是一种 move-only 类型，所以它的 setter 的“重载”方法由一个函数组成：

```
class Widget {
public:
    ...
    void setPtr(std::unique_ptr<std::string>&& ptr)
    { p = std::move(ptr); }

private:
    std::unique_ptr<std::string> p;
};
```

   调用者可能这样调用：

```
Widget w;
...
```

```
w.setPtr(std::make_unique<std::string>("Modern C++"));
```

此处，从 std::make_unique(见条款 21)返回的 std::unique_ptr<std::string>右值，通过右值引用传递给 setPtr，并将其移动到数据成员 p 中，总代价就是一次 move。如果 setPtr 的形参按值传递，

```
class Widget {
public:
    ...

    void setPtr(std::unique_ptr<std::string> ptr)
    { p = std::move(ptr); }

    ...
};
```

同样的调用将移动构造参数 ptr，然后将 ptr 移动到数据成员 p。因此，总代价将是两次移动，是“重载”方法的两倍。

3. 仅对于移动代价低的参数，传值才是值得考虑的。当移动代价低时，一个额外移动的代价可能是可接受的，否则，执行不必要的移动就类似于执行不必要的拷贝，而 C++ 98 规则中把避免传值放在第一位，就是为了避免不必要的拷贝操作！
4. 你应该只对始终需要副本的参数考虑传值。要了解为什么这很重要，我们假设在将参数复制到 names 容器之前，addName 要检查新名称是否太短或太长，如果是，添加名称的请求将被忽略。传值的实现可以这样写：

```
class Widget {
public:
    void addName(std::string newName)
    {
        if ((newName.length() >= minLen) &&
            (newName.length() <= maxLen))
        {
            names.push_back(std::move(newName));
        }
    }
    ...
private:
    std::vector<std::string> names;
};
```

即使没有添加，这个函数也需要构造和销毁 newName。这是传引用方法不存在的

代价。

即使处理的函数在可拷贝类型上执行无条件复制，而且移动代价也低，但有时传值可能也并不合适。这是因为函数可以通过两种方式复制参数：构造 (即，拷贝构造或移动构造)和赋值(即，拷贝或移动赋值)。addName 用的是构造：newName 传给 vector::push_back，在这个函数中，newName 被拷贝构造到 std::vector 末尾创建的新元素中。对于使用构造来复制参数的函数，我们前面看到的分析已经完成：传值会为 lvalue 和 rvalue 参数带来一次额外的移动代价。

当使用赋值复制参数时，情况会更加复杂。例如，假设我们有一个表示密码的类。因为密码可以更改，所以我们提供了一个 setter 函数 changeTo。使用传值策略，我们可以实现 Password 如下：

```
class Password {
public:
    explicit Password(std::string pwd)     // pass by value
    : text(std::move(pwd)) {}              // construct text

    void changeTo(std::string newPwd)      // pass by value
    { text = std::move(newPwd); }          // assign text

    ...
private:
    std::string text;                      // text of password
};
```

将密码存储为纯文本会让软件安全 SWAT 团队抓狂，但请忽略这一点，考虑以下代码：

```
std::string initPwd("Supercalifragilisticexpialidocious");
Password p(initPwd);
```

此处没有任何令人惊讶的地方：p.text 是用给定的密码构造的，在构造函数中使用传值会导致 std::string move 构造的额外开销，而如果使用重载或完美转发，则不存在。一切都好。

然而，这个程序的用户可能对密码不那么乐观，因为 “Supercalifragilisticexpialidocious” 在许多词典中都有。因此，他或她可能执行以下代码：

```
std::string newPassword = "Beware the Jabberwock";
p.changeTo(newPassword);
```

新密码是否比旧密码更好尚存争议，但这是用户的问题。我们的问题是，changeTo 使用赋值来复制参数 newPwd，这可能会导致传值策略的代价激增。

传递给 changeTo 的实参是 lvalue (newPassword)，因此在构造参数 newPwd 时，调用的是 std::string 拷贝构造函数。该构造函数分配内存来保存新密码。然后 newPwd 被移动赋值到 text，这将导致 text 已经持有的内存被释放。因此在 changeTo 中有两个动态内存管理操作：一个为新密码分配内存，另一个为旧密码释放内存。

但在本例中，旧密码(“Supercalifragilisticexpialidocious”)比新密码(“Beware the Jabberwock”)长，因此其实不必要分配或释放任何东西。如果使用重载方法，很可能就不会发生那种情况：

```
class Password {
public:
    ...
    void changeTo(const std::string& newPwd)  // the overload
    {                                         // for lvalues
        text = newPwd;   // can reuse text's memory if
                         // text.capacity() >= newPwd.size()
    }
    ...
private:
    std::string text; // as above
};
```

在这个场景中，传值的代价包括额外的内存分配和回收——这些代价可能比 std::string 移动操作高出好几个数量级。

有趣的是，如果旧密码比新密码短，通常不可能在赋值期间避免分配-回收，在这种情况下，传值和传引用的运行速度一样。因此，基于赋值的参数复制的代价取决于对象的值！这种分析适用于值在动态分配内存中的任何参数类型，不是所有类型都符合条件，但是很多类型都符合条件，包括 std::string 和 std::vector。

这种潜在的代价增加通常只适用于传递 lvalue 参数时，因为通常只在真正的复制操作(即，不是移动)时才执行内存分配和回收。对于 rvalue 参数，几乎总是执行移动操作。

总结下来就是，对于使用赋值来复制参数的函数，传值的额外代价取决于：(1)传递的类型，(2)左值和右值的比例，(3)类型是否使用动态分配的内存，而如果用了，则取决于该类型的赋值实现以及目标内存至少与源内存一样大的可能性。对于 std::string，它还取决于赋值实现是否使用小字符串优化(SSO，见条款 29)，而如果使用了，则取决于值是否适合 SSO 缓冲区。

所以，正如我所说，当通过赋值复制参数时，传值的代价分析很复杂。通常，最实际的

方法是采用“被证明是无辜的之前都是有罪的”策略，在这种策略中，使用重载或通用引用，而不是传值，除非已经证明传值可以生成可接受的高效代码。

现在，对于必须尽可能快的软件来说，传值可能不是一个可行的策略，因为即使是避免代价低的移动也很重要。此外，还不清楚会有多少次移动。在 Widget::addName 示例中，传值只会导致一次额外的移动操作，但是假设 Widget::addName 调用 Widget::validateName，并且这个函数也传值。(假设有个理由，它总是复制它的参数，如，将参数存储在它验证过的所有值的数据结构中。)并且假设 validateName 调用了第三个传值的函数……

你可以看到它的发展方向。当存在函数调用链时，每个函数调用都传值，因为“它只花费一次廉价的移动”，那么整个调用链的成本可能不是你可以容忍的。使用传引用，调用链就不会产生这种累积的开销。

一个与性能无关，但仍然值得记住的问题是，与传引用不同，传值容易受到切片问题影响。这是 C++98 基本原则，所以我不细讲了，但如果你有设计一个函数,它接受一个基类类型或任何派生类型，你不会声明一个传值的参数类型，因为你会“割掉”派生类对象的派生类特性：

```
class Widget { … };                    // base class

class SpecialWidget: public Widget { … }; // derived class

void processWidget(Widget w);          // func for any kind of Widget,
                                       // including derived types;
…                                      // suffers from slicing problem

SpecialWidget sw;

…

processWidget(sw);                     // processWidget sees a
                                       // Widget, not a SpecialWidget!
```

如果你不熟悉切片问题，搜索引擎和互联网是你的朋友，那儿有很多可用的信息。你将会发现，切片问题是传值在 C++ 98 中名声不好另一个原因(除了效率问题)。关于 C++编程，你可能首先学到的一件事就是避免按值传递用户定义类型的对象，这是有充分理由的。

C++11 并没有从根本上改变 C++98 关于传值的至理名言。通常，传值仍然会导致性能损失，并且仍然会导致切片问题。C++11 的新特性是 lvalue 和 rvalue 参数之间的区别。对

可拷贝类型的右值利用 move 语义的实现函数，要么重载，要么使用通用引用，而这两种方法都有缺点。对于可拷贝类型的特殊情况，即，函数总是要复制参数，也不存在切片问题，并且传给函数的类型是 cheap-to-move，传值可以提供一种易于实现的替代方法，这种方法几乎与传引用一样高效，但避免了重载和通用引用的缺点。

需要铭记的要点
● 对于可拷贝、move 代价低并且总要复制的参数，传值可能几乎与传引用一样高效，还很容易实现，并且生成更少的目标代码。 ● 通过构造复制参数可能比赋值的代价大得多。(如，SSO) ● 传值受切片问题影响，因此它通常不适用于基类参数类型。

条款 42：考虑使用 emplace 代替 insert

如果你有一个容器，存储 std::string，那么当你通过插入函数(即，insert、push_front、push_back 或 std::forward_list 的 insert_after)，传递给函数的元素类型是 std::string，这个看上去很符合逻辑，毕竟，这就是容器存储的内容。

这也许合乎逻辑，但并不总是如此。考虑这段代码：

```
std::vector<std::string> vs; // container of std::string
vs.push_back("xyzzy");       // add string literal
```

此处，容器保存 std::string，但是传给 push_back 的是一个字符串。字符串不是 std::string，这意味着你传递给 push_back 的参数不是容器存储的类型。

std::vector 的 push_back 对于 lvalues 和 rvalues 重载如下：

```
template <class T,                              // from the C++11
          class Allocator = allocator<T>>       // Standard
class vector {
public:
    ...
    void push_back(const T& x);                 // insert lvalue
    void push_back(T&& x);                      // insert rvalue
    ...
};
```

在下面调用中

```
vs.push_back("xyzzy");
```

编译器会看到实参的类型(const char[6])与 push_back 形参的类型(std::string 引用)不匹配。它们通过从字符串创建临时 std::string 对象来解决不匹配问题，并将该临时对象传递给 push_back。换句话说，这个调用就相当于下面这样：

```
vs.push_back(std::string("xyzzy"));   // create temp. std::string
                                      // and pass it to push_back
```

代码编译并运行，所有人都可以开开心心回家了。这个所有人不包括性能狂人，因为性能狂人认为这段代码并没有达到它应有的效率。

要在 std::string 的容器中创建新元素，编译器知道，必须调用 std::string 构造函数，但是上面的代码不只是调用一次构造，它调用了两次，并且也调用了 std::string 析构函数。以下是运行时的情况：

1. 临时 std::string 对象是从字符串“xyzzy”创建的。该对象没有名称，我们称它为 temp，

temp 的构造是第一次 std::string 构造。因为它是一个临时对象，所以 temp 是一个 rvalue。

2. temp 传给 push_back 的 rvalue 重载，它绑定到 rvalue 引用参数 x，然后在内存中为 std::vector 构造一个 x 的副本。这个构造，即第二次构造，实际上是在 std::vector 中创建一个新对象。(用于将 x 复制到 std::vector 中的构造函数是 move 构造函数，因为 x 是一个 rvalue 引用，在复制之前被转换为 rvalue。有关将 rvalue 引用参数转换为 rvalues 的信息，请参见条款 25)。
3. 在 push_back 返回后，立即销毁临时变量 temp，因此调用了 std::string 析构函数。

性能狂人不禁注意到，如果有一种方法可以将字符串直接传递给步骤 2 中在 std::vector 中构造 std::string 对象的代码，那么我们就可以避免构造和销毁 temp。这才是最有效的，这样，那些性能狂人才会心满意足地离开。

因为你是一个 C++程序员，所以有很大可能你会成为性能狂人。如果你不是，你可能也会认同他们的观点。(如果你对性能一点都不感兴趣，那么你不应该去用 Python 吗？)所以我很高兴地告诉你们，有一种方法可以达到 push_back 调用的最大性能要求。不是 push_back 函数，而是 emplace_back。

emplace_back 正是我们想要的：它使用传递给它的任何参数直接在 std::vector 中构造 std::string。不涉及临时对象：

```
vs.emplace_back("xyzzy");     // construct std::string inside
                              // vs directly from "xyzzy"
```

emplace_back 使用了完美转发，因此，只要你没有遇到完美转发的限制 (见条款 30)，就可以通过 emplace_back 传递任意数量的任意类型组合的参数。例如，如果你想在 vs 中通过 std::string 构造函数创建一个 std::string，该构造函数接受一个字符和重复次数，可以这样调用：

```
vs.emplace_back(50, 'x');     // insert std::string consisting
                              // of 50 'x' characters
```

emplace_back 可用于支持 push_back 的每个标准容器。类似地，每个支持 push_front 的标准容器都支持 emplace_front。每个支持 insert 的标准容器(除了 std::forward_list 和 std::array)都支持 emplace，相应的容器提供 emplace_hint 来弥补接受“hint”迭代器的 insert 函数，std::forward_list 用 emplace_after 来匹配它的 insert_after。

使放置函数优于插入函数的是它们更灵活的接口。插入函数接受要插入的对象，而放置函数接受要插入的对象的构造函数参数。这种差异允许放置函数避免创建和销毁插入函数可

能需要的临时对象。

因为容器存储类型的实参也可以传给放置函数(导致拷贝或移动构造)，所以即使插入函数不需要临时对象，也可以使用放置函数。在这种情况下，插入和放置基本上做相同的事情。例如，给定

```
std::string queenOfDisco("Donna Summer");
```

以下两个调用都是合法的，并且对容器具有相同的效率：

```
vs.push_back(queenOfDisco);         // copy-construct queenOfDisco
                                    // at end of vs
vs.emplace_back(queenOfDisco);      // ditto
```

因此，放置函数可以做插入函数所能做的一切。他们有时还更高效，而且，至少在理论上，它们永远不会低效。所以为什么不一直使用它们呢？

因为，俗话说，理论上来说，理论和实践之间没有区别，但在实践中，是存在区别的。在标准库的现有实现中，有一些情况下，正如预期的那样，放置优于插入，但遗憾的是，也有一些情况下插入函数运行得更快。这种情况不容易描述，因为它们取决于传递的实参类型，使用的容器，容器中请求插入或放置的位置，容器存储类型的构造函数的异常安全性，以及，对于禁止重复值的容器(即， std::set, std::map, std::unordered_set, std::unordered_map)，要添加的值是否已经在容器中。因此，通常的性能调优建议是：要确定是放置还是插入运行得更快，就对它们进行基准测试。

当然，这并不令人十分满意，所以你会很高兴地了解到，有一种启发式方法可以帮助你识别最值得使用放置函数的情况。如果下面条件这些都成立，放置几乎肯定会优于插入：

- **要添加的值是构造而不是赋值到容器中**。本条款的示例(向 std::vector vs 中添加一个值为“xyzzy”的 std::string)表明，将该值添加到 vs 的末尾——一个还不存在对象的地方。因此，必须将新值构造到 std::vector 中。如果我们修改这个例子，使新的 std::string 进入一个已经被对象占用的位置，情况就不同了。考虑：

  ```
  std::vector<std::string> vs;      // as before
  …                                 // add elements to vs
  vs.emplace(vs.begin(), "xyzzy");  // add "xyzzy" to beginning of vs
  ```

  对于这段代码，很少有实现会将添加的 std::string 构造到 vs[0]占用的内存中。相反，它们会移动赋值到相应位置。但是移动赋值需要一个对象来移动，这意味着需要创建一个临时对象来作为移动的源。由于放置优于插入的主要优点是临时对象既不创建也不销毁，所以当通过赋值将添加的值放入容器时，放置的边界往往会消失。

遗憾的是，向容器添加值是通过构造还是通过赋值来完成的，这通常取决于实现者。但是，启发式也有帮助。

基于节点的容器实际上总是使用构造来添加新值，而大多数标准容器都是基于节点的。唯一例外的是 std::vector、std::deque 和 std::string。(std::array 也不是，但它不支持插入或放置，所以跟主题无关。)在非基于节点的容器中，你可以依赖 emplace_back 来使用构造而不是赋值来存入新值，而对于 std::deque，emplace_front 也是如此。

- **传递的实参类型与容器存储的类型不同**。同样，与插入相比，放置的优势通常源于这样一个事实，即当传递的实参不是容器存储的类型时，它的接口不需要创建和销毁临时对象。当将类型为 T 的对象添加到 container<T>中时，没有理由期望放置比插入运行得更快，因为此时不需要创建临时对象来满足插入接口。
- **容器未必会拒绝重复的新值**。这意味着容器要么允许重复，要么你添加的大多数值都是惟一的。这很重要的原因是，为了检测值是否已经在容器中，放置实现通常会创建一个具有新值的节点，以便可以将该节点的值与现有的容器节点进行比较。如果要添加的值不在容器中，则将节点链接进来，但是，如果该值已经存在，则放置中止，节点被销毁，这意味着节点的构造和销毁是浪费的。这些节点更多的是为放置函数而非插入函数创建的。

下面的调用满足上述所有条件，也比 push_back 的调用运行得更快。

```
vs.emplace_back("xyzzy");      // 在容器的末尾构造新值
                               // 未传递容器存储的类型
                               // 容器不拒绝重复值

vs.emplace_back(50, 'x');      // ditto
```

在决定是否使用放置函数时，还有两个问题值得考虑。首先是资源管理。假设你有一个存储 std::shared_ptr<Widget>的容器，

```
std::list<std::shared_ptr<Widget>> ptrs;
```

并且你希望添加一个 std::shared_ptr，它通过自定义 deleter 释放 (见条款 19)。条款 21 说明，你应该尽可能地使用 std::make_shared 来创建 std::shared_ptr，但它也承认，有些情况下你做不到，其中一种情况就是，你希望指定自定义 deleter，在这种情况下，你必须直接使用 new 来获取由 std::shared_ptr 管理的原生指针。

如果自定义 deleter 是下面这个函数，

```
void killWidget(Widget* pWidget);
```

使用插入函数的代码可能是这样的：

```
ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));
```

它也可以是这样的，虽然意思是一样的：

```
ptrs.push_back({ new Widget, killWidget });
```

无论哪种方式，在调用 push_back 之前都会构造一个临时 std::shared_ptr，因为 push_back 的参数是 std::shared_ptr 引用，因此必须有一个 std::shared_ptr 来引用。

临时 std::shared_ptr 的创建就是 emplace_back 要避免的，但是在这个例子中，这个临时对象的价值远超它的构造代价。考虑以下可能发生的事件序列：

1. 在上面的两个调用中，都构造了一个临时 std::shared_ptr<Widget>对象来保存由“new Widget”产生的原生指针。将这个对象称作 temp。
2. push_back 接受 temp 引用，在分配存储 temp 副本的 list 节点内存时，产生内存不足异常。
3. 当异常从 push_back 抛出后，temp 被销毁。作为惟一引用 Widget 的 std::shared_ptr，它会自动释放该 Widget，在本例中是通过调用 killWidget。

即使发生了异常，也不会发生任何泄漏：通过“new”创建的 Widget 在 std::shared_ptr 的析构函数中释放。生活是如此美好。

现在考虑一下，如果调用 emplace_back 而不是 push_back，会发生什么情况：

```
ptrs.emplace_back(new Widget, killWidget);
```

1. 由“new Widget”产生的原生指针完美转发到 emplace_back 中分配 list 节点的位置，节点分配失败，并抛出内存不足异常。
2. 当异常从 emplace_back 抛出时，原生指针丢失了，而原生指针是在堆上获取 Widget 的唯一途径。那个 Widget (以及它拥有的所有资源)也就泄漏了。

在这个场景中，生活就不美好了，而错不在 std::shared_ptr。使用带有自定义 deleter 的 std::unique_ptr 也会出现同样的问题。基本上，像 std::shared_ptr 和 std::unique_ptr 这种资源管理类的有效性，是基于资源(例如来自 new 的原生指针)会立即传递给资源管理对象的构造函数。像 std::make_shared 和 std::make_unique 这样的函数就可以自动做到，这也是它们如此重要的原因之一。

在存储资源管理对象的容器(如，std::list<std::shared_ptr<Widget>>)的插入函数调用中，函数的参数类型通常确保在获取资源(如，使用 new)和管理资源的对象的构造之间不发生任

何事情。在放置函数中，完全转发延迟了资源管理对象的创建，直到它们可以在容器的内存中构造，而这将打开一个窗口，在此窗口中，异常可能导致资源泄漏。所有标准容器都受此问题影响。在使用资源管理对象容器时，必须确保如果选择放置函数而不是插入函数，你不会为了提高代码效率而降低了异常安全性。

坦率地说，无论如何你都不应该将“new Widget”之类的表达式传递给 emplace_back 或 push_back 或大多数其他函数，因为正如条款 21 所解释的那样，这可能带来我们刚刚检查过的那种异常安全问题。这种实现需要在独立语句中将“new Widget”转换为资源管理对象，然后将该对象作为右值传递给最初希望传递“new Widget”的函数。(条款 21 更详细地介绍了这项技术。)因此，使用 push_back 的代码应该如下：

```
std::shared_ptr<Widget> spw(new Widget,        // create Widget and
                            killWidget);       // have spw manage it

ptrs.push_back(std::move(spw));                // add spw as rvalue
```

emplace_back 版本类似：

```
std::shared_ptr<Widget> spw(new Widget, killWidget);
ptrs.emplace_back(std::move(spw));
```

无论哪种方式，这种方法都要付出创建和销毁 spw 的代价。考虑到选择放置而不是插入的动机是为了避免临时对象的代价，而概念上来说，spw 就是这个对象，当你将资源管理对象添加到容器中，并且你遵循原则，确保在获取资源和将资源转换为资源管理对象之间没有任何东西可以进行干预，这时，放置函数不太可能优于插入函数。

放置函数第二个值得注意的方面是它们与显式(explicit)构造函数的交互。为了纪念 C++ 11 对正则表达式的支持，假设你创建了一个包含正则表达式对象的容器：

```
std::vector<std::regex> regexes;
```

同事们为每天查看 Facebook 账号的理想次数争吵不休，你心烦意乱，一不小心写下了下面看似毫无意义的代码：

```
regexes.emplace_back(nullptr); // add nullptr to container of regexes?
```

你在键入时不会注意到这个错误，编译器也会毫无怨言地接受这个代码，因此你最终会浪费大量时间来调试。在某个时刻，你会发现在正则表达式容器中插入了一个空指针。但这怎么可能呢？指针不是正则表达式，如果你想写成下面这样，

```
std::regex r = nullptr; // error! won't compile
```

编译不通过。有趣的是，如果你调用 push_back 而不是 emplace_back，编译也不通

过：

```
regexes.push_back(nullptr); // error! won't compile
```

你正在经历的奇怪行为源于这样一个事实：std::regex 对象可以由字符串构造。这就是为什么像这样有用的代码是合法的：

```
std::regex upperCaseWord("[A-Z]+");
```

从字符串创建 std::regex 可能需要相当大的运行时开销，因此，为了将这种开销无意中产生的可能性降到最低，接受 const char*指针的 std::regex 构造函数是 explicit。这就是为什么这些无法编译通过：

```
std::regex r = nullptr;        // error! won't compile
regexes.push_back(nullptr);  // error! won't compile
```

在这两种情况下，我们都请求从指针到 std::regex 的隐式转换，而构造函数的 explicit 特性阻止了此类转换。

然而，在 emplace_back 调用中，我们并不声称传递的是 std::regex 对象，相反，我们是为 std::regex 对象传递构造函数参数。这不是隐式转换请求。相反，它被看作是你编写的如下代码：

```
std::regex r(nullptr); // compiles
```

如果简短的注释“编译”表明缺乏热情，那很好，因为这段代码虽然可以编译，却存在未定义行为。接受 const char*指针的 std::regex 构造函数要求指向的字符串包含一个有效的正则表达式，而空指针不满足这一要求，如果你编写并编译这样的代码，你所能期望的最好结果是它在运行时崩溃。如果运气不好，你将体验一场特殊的调试之旅。

暂且不考虑 push_back、emplace_back 和调试之旅，请注意这些非常相似的初始化语法如何产生不同的结果：

```
std::regex r1 = nullptr; // error! won't compile
std::regex r2(nullptr); // compiles
```

在标准的官方术语中，用于初始化 r1(使用等号)的语法对应于所谓的复制初始化（copy initialization）。相反，用于初始化 r2 的语法(使用括号，但也可以使用花括号)产生所谓的直接初始化（direct initialization）。复制初始化不允许使用显式构造函数，而直接初始化允许。这就是为什么初始化 r1 不能编译通过，但是初始化 r2 可以编译通过。

但回到 push_back 和 emplace_back，以及更一般的插入和放置函数。放置函数使用直接初始化，这意味着它们可以使用显式构造函数，而插入函数使用复制初始化，因此它们不

能使用显式构造函数。因此：

```
regexes.emplace_back(nullptr); // compiles. Direct init permits
                               // use of explicit std::regex
                               // ctor taking a pointer

regexes.push_back(nullptr);    // error! copy init forbids
                               // use of that ctor
```

要吸取的教训是，在使用放置函数时，要特别小心，确保传递了正确的参数，因为在编译器试图找到一种方法将代码解释为合法时，即使是显式构造函数也会被考虑进去。

需要铭记的要点
● 原则上，放置函数有时应该比插入函数更有效，而且永远不会比插入函数低效。 ● 在实践中，当(1)要添加的值被构造而不是赋值到容器中；(2)传递的参数类型与容器持有的类型不同；并且(3)容器不会因为它是一个重复值而拒绝要添加的值，这些条件满足时，放置函数最有可能更快。 ● 放置函数可以执行插入函数拒绝的类型转换。