考虑lambda表达式而非std::bind

C++11中的`std::bind`是C++98的`std::bind1st`和`std::bind2nd`的后续，但在2005年已经成为了标准库的一部分。那时标准化委员采用了TR1的文档，其中包含了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`，接口细节也有所不同）。这段历史意味着一些程序员有十年或更长时间的使用`std::bind`经验。如果您是其中之一，可能会不愿意放弃一个对您有用的工具。这是可以理解的，但是在这种情况下，改变是更好的，因为在C ++11中，`lambda`几乎是比`std :: bind`更好的选择。 从C++14开始，`lambda`的作用不仅强大，而且是完全值得使用的。

这个条目假设您熟悉`std::bind`。 如果不是这样，您将需要获得基本的了解，然后再继续。 无论如何，这样的理解都是值得的，因为您永远不知道何时会在必须阅读或维护的代码库中遇到`std::bind`的使用。

与第32项中一样，我们将从`std::bind`返回的函数对象称为绑定对象。

优先lambda而不是`std::bind`的最重要原因是lambda更易读。 例如，假设我们有一个设置闹钟的函数：

```
// 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，我们已经确定需要设置一个小时后响30秒的闹钟。 但是，具体声音仍未确定。我们可以编写一个lambda来修改`setAlarm`的界面，以便仅需要指定声音：

```
// 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
    };
```

我们在lambda中突出了对`setAlarm`的调用。这看来起是一个很正常的函数调用，即使是几乎没有lambda经验的读者也可以看到：传递给lambda的参数被传递给了`setAlarm`。

通过使用基于C++11对用户自定义常量的支持而建立的标准后缀，如秒(s)，毫秒(ms)和小时(h)等，我们可以简化C++14中的代码。这些后缀在`std::literals`命名空间中实现，因此上述代码可以按照以下方式重写：

```
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
    };
```

下面是我们第一次编写对应的`std::bind`调用。这里存在一个我们后续会修复的错误，但正确的代码会更加复杂，即使是此简化版本也会带来一些重要问题：

```
using namespace std::chrono; // as above
using namespace std::literals;
using namespace std::placeholders; // needed for use of "_1"
auto setSoundB = std::bind(setAlarm, // "B" for "bind"
                           steady_clock::now() + 1h, // incorrect! see below
                           _1,
                           30s);
```

我想像在lambda中一样突出显示对`setAlarm`的调用，但是没有这么做。这段代码的读者只需知道，调用`setSoundB`会使用在对`std :: bind`的调用中所指定的时间和持续时间来调用`setAlarm`。对于初学者来说，占位符**“ _1”**本质上是一个魔术，但即使是普通读者也必须从思维上将占位符中的数字映射到其在`std::bind`参数列表中的位置，以便明白调用`setSoundB`时的第一个参数会被传递进`setAlarm`，作为调用时的第二个参数。在对`std::bind`的调用中未标识此参数的类型，因此读者必须查阅`setAlarm`声明以确定将哪种参数传递给`setSoundB`。

但正如我所说，代码并不完全正确。在lambda中，表达式`steady_clock::now() + 1h`显然是是`setAlarm`的参数。调用`setAlarm`时将对其进行计算。这是合理的：我们希望在调用`setAlarm`后一小时发出警报。但是，在`std::bind`调用中，将`steady_clock::now() + 1h`作为参数传递给了`std::bind`，而不是setAlarm。这意味着将在调用std::bind时对表达式进行求值，并且该表达式产生的时间将存储在结果绑定对象中。结果，闹钟将被设置为在调用std::bind后一小时发出声音，而不是在调用setAlarm`一小时后发出。

要解决此问题，需要告诉`std::bind`推迟对表达式的求值，直到调用`setAlarm`为止，而这样做的方法是将对`std::bind`的第二个调用嵌套在第一个调用中：

```
auto setSoundB =
     std::bind(setAlarm,
     std::bind(std::plus<>(), steady_clock::now(), 1h), _1,
     30s);
```

如果您熟悉C++98的`std::plus`模板，您可能会惊讶地发现在此代码中，尖括号之间未指定任何类型，即该代码包含`std::plus<>`，而不是`std::plus<type>`。 在C ++14中，通常可以省略标准运算符模板的模板类型参数，因此无需在此处提供。 C++11没有提供此类功能，因此等效于lambda的C ++11`std::bind`使用为：

```
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)),
               seconds(30));
```

如果此时Lambda看起来不够吸引，那么应该检查一下视力了。

当setAlarm重载时，会出现一个新问题。 假设有一个重载函数，其中第四个参数指定了音量：

```
enum class Volume { Normal, Loud, LoudPlusPlus };
void setAlarm(Time t, Sound s, Duration d, Volume v);
```

lambda能继续像以前一样使用，因为根据重载规则选择了`setAlarm`的三参数版本：

```
auto setSoundL =
    [](Sound s)
    {
      using namespace std::chrono;
        setAlarm(steady_clock::now() + 1h, s,
        30s);
        };
```

然而，`std::bind`的调用将会编译失败：

```
auto setSoundB =                                // error! which
    std::bind(setAlarm,                         // setAlarm?
                           std::bind(std::plus<>(),
              steady_clock::now(),
                1h),
    _1,
        30s);
```

这里的问题是，编译器无法确定应将两个setAlarm函数中的哪一个传递给`std::bind`。 它们仅有的是一个函数名称，而这个函数名称是不确定的。
要获得对`std::bind`的调用能进行编译，必须将`setAlarm`强制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);
```

但这在`lambda`和`std::bind`的使用上带来了另一个区别。 在`setSoundL`的函数调用操作符（即lambda的闭包类对应的函数调用操作符）内部，对`setAlarm`的调用是正常的函数调用，编译器可以按常规方式进行内联：

```
setSoundL(Sound::Siren);    // body of setAlarm may
                                                // well be inlined here
```

但是，对`std::bind`的调用是将函数指针传递给`setAlarm`，这意味着在`setSoundB`的函数调用操作符（即绑定对象的函数调用操作符）内部，对`setAlarm`的调用是通过一个函数指针。 编译器不太可能通过函数指针内联函数，这意味着与通过`setSoundL`进行调用相比，通过`setSoundB`对`setAlarm`的调用，其函数不大可能被内联：

```
setSoundB(Sound::Siren);    // body of setAlarm is less
                                                // likely to be inlined here
```

因此，使用`lambda`可能会比使用`std::bind`能生成更快的代码。
`setAlarm`示例仅涉及一个简单的函数调用。如果您想做更复杂的事情，使用lambda会更有利。 例如，考虑以下C++14的lambda使用，它返回其参数是否在最小值（`lowVal`）和最大值（`highVal`）之间的结果，其中`lowVal`和`highVal` 是局部变量：

```
auto betweenL =
     [lowVal, highVal]
     (const auto& val)                          // C++14
     { return lowVal <= val && val <= highVal; };
```

使用`std::bind`可以表达相同的内容，但是该构造是一个通过晦涩难懂的代码来保证工作安全性的示例：

```
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`调用将如下所示：

```
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也不能采用`auto`参数，因此它也必须指定一个类型：

```
auto betweenL = // C++11 version
  [lowVal, highVal]
    (int val)
    { return lowVal <= val && val <= highVal; };
```

无论哪种方式，我希望我们都能同意，lambda版本不仅更短，而且更易于理解和维护。
之前我就说过，对于那些没有`std::bind`使用经验的人，其占位符（例如_1，_2等）本质上都是magic。 但是，不仅仅占位符的行为是不透明的。 假设我们有一个函数可以创建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
```

并且我们想创建一个函数对象，该函数对象允许我们指定应将特定`w`的压缩级别。这种使用`std::bind`的话将创建一个这样的对象：

```
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，必须将其存储起来，以便以后进行压缩。它存储在对象compressRateB中，但是这是如何存储的呢（是通过值还是引用）。之所以会有所不同，是因为如果在对`std::bind`的调用与对`compressRateB`的调用之间修改了`w`，则按引用捕获的`w`将反映其更改，而按值捕获则不会。

答案是它是按值捕获的，但唯一知道的方法是记住`std::bind`的工作方式；在对`std::bind`的调用中没有任何迹象。与lambda方法相反，其中`w`是通过值还是通过引用捕获是显式的：

```
auto compressRateL =            // w is captured by
  [w](CompLevel lev)                        // value; lev is
    { return compress(w, lev); };   // passed by value
```

同样明确的是如何将参数传递给lambda。 在这里，很明显参数`lev`是通过值传递的。 因此：

```
compressRateL(CompLevel::High); // arg is passed
                                                  // by value
```

但是在对由`std::bind`生成的对象调用中，参数如何传递？

```
compressRateB(CompLevel::High); // how is arg
                                                  // passed?
```

同样，唯一的方法是记住`std::bind`的工作方式。（答案是传递给绑定对象的所有参数都是通过引用传递的，因为此类对象的函数调用运算符使用完美转发。）
与lambda相比，使用`std::bind`进行编码的代码可读性较低，表达能力较低，并且效率可能较低。 在C++14中，没有`std::bind`的合理用例。 但是，在C ++11中，可以在两个受约束的情况下证明使用`std::bind`是合理的：

- 移动捕获。 C++11的lambda不提供移动捕获，但是可以通过结合lambda和`std::bind`来模拟。有关详细信息，请参阅条款32，该条款还解释了在C ++ 14中，lambda对初始化捕获的支持将少了模拟的需求。
- 多态函数对象。 因为绑定对象上的函数调用运算符使用完全转发，所以它可以接受任何类型的参数（以条款30中描述的完全转发的限制为例子）。当您要使用模板化函数调用运算符来绑定对象时，此功能很有用。 例如这个类，

```
class PolyWidget {
 public:
    template<typename T>
    void operator()(const T& param); ...
};
```

`std::bind`可以如下绑定一个`PolyWidget`对象：

```
PolyWidget pw;
auto boundPW = std::bind(pw, _1);
```

`boundPW`可以接受任意类型的对象了：

```
boundPW(1930);  // pass int to
                                  // PolyWidget::operator()
boundPW(nullptr); // pass nullptr to
                                    // PolyWidget::operator()
boundPW("Rosebud"); // pass string literal to
                                       // PolyWidget::operator()
```

这一点无法使用C++11的lambda做到。 但是，在C++14中，可以通过带有`auto`参数的lambda轻松实现：

```
auto boundPW = [pw](const auto& param) // C++14
                               { pw(param); };
```

当然，这些是特殊情况，并且是暂时的特殊情况，因为支持C++14 lambda的编译器越来越普遍了。 当`bind`在2005年被非正式地添加到C ++中时，与1998年的前身相比有了很大的改进。 在C ++11中增加了lambda支持，这使得`std::bind`几乎已经过时了，从C ++ 14开始，更是没有很好的用例了。

要谨记的是：

- 与使用`std::bind`相比，Lambda更易读，更具表达力并且可能更高效。
- 只有在C++11中，`std::bind`可能对实现移动捕获或使用模板化函数调用运算符来绑定对象时会很有用。