Lambda表达式是C++编程中的游戏规则改变者。这有点令人惊讶，因为它没有给语言带来新的表达能力。Lambda可以做的所有事情都可以通过其他方式完成。但是lambda是创建函数对象相当便捷的一种方法，对于日常的C++开发影响是巨大的。没有lambda时，标准库中的`_if`算法（比如，`std::find_if, std::remove_if, std::count_if`等）通常需要繁琐的谓词，但是当有lambda可用时，这些算法使用起来就变得相当方便。比较函数（比如，`std::sort, std::nth_element, std::lower_bound`等）与算法函数也是相同的。在标准库外，lambda可以快速创建`std::unique_ptr`和`std::shared_ptr`的自定义`deleter`，并且使线程API中条件变量的条件设置变得同样简单（参见Item 39）。除了标准库，lambda有利于即时的回调函数，接口适配函数和特定上下文中的一次性函数。Lambda确实使C++成为更令人愉快的编程语言。

与Lambda相关的词汇可能会令人疑惑，这里做一下简单的回顾：

- *lambda表达式就是一个表达式*。在代码的高亮部分就是lambda

```
std::find_if(container.begin(), container.end(),
             [](int val){ return 0 < val && val < 10; }); // 本行高亮
```

- *闭包*是lambda创建的运行时对象。依赖捕获模式，闭包持有捕获数据的副本或者引用。在上面的`std::find_if`调用中，闭包是运行时传递给`std::find_if`第三个参数。
- *闭包类（closure class）*是从中实例化闭包的类。每个lambda都会使编译器生成唯一的闭包类。Lambda中的语句成为其闭包类的成员函数中的可执行指令。

Lambda通常被用来创建闭包，该闭包仅用作函数的参数。上面对`std::find_if`的调用就是这种情况。然而，闭包通常可以拷贝，所以可能有多个闭包对应于一个lambda。比如下面的代码：

```
{
  int x; // x is local variable
  ...
  auto c1 = [x](int y) { return x * y > 55; }; // c1 is copy of the 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lambda产生的闭包的副本。

非正式的讲，模糊lambda，闭包和闭包类之间的界限是可以接受的。但是，在随后的Item中，区分编译期（lambdas 和 closure classes）还是运行时（closures）以及它们之间的相互关系是重要的。

避免使用默认捕获模式

C++11中有两种默认的捕获模式：按引用捕获和按值捕获。但按引用捕获可能会带来悬空引用的问题，而按值引用可能会诱骗你让你以为能解决悬空引用的问题（实际上并没有），还会让你以为你的闭包是独立的（事实上也不是独立的）。

这就是本条目的一个总结。如果你是一个工程师，渴望了解更多内容，就让我们从按引用捕获的危害谈起把。

按引用捕获会导致闭包中包含了对局部变量或者某个形参（位于定义lambda的作用域）的引用，如果该lambda创建的闭包生命周期超过了局部变量或者参数的生命周期，那么闭包中的引用将会变成悬空引用。举个例子，假如我们有一个元素是过滤函数的容器，该函数接受一个int作为参数，并返回一个布尔值，该布尔值的结果表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
```

我们可以添加一个过滤器，用来过滤掉5的倍数。

```
filters.emplace_back(                     // see Item 42 for
    [](int value) { return value % 5 == 0; } // info on
);
```

然而我们可能需要的是能够在运行期获得被除数，而不是将5硬编码到lambda中。因此添加的过滤器逻辑将会是如下这样：

```
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!
```

这个代码实现是一个定时炸弹。lambda对局部变量divisor进行了引用，但该变量的生命周期会在addDivisorFilter返回时结束，刚好就是在语句filters.emplace_back返回之后，因此该函数的本质就是容器添加完，该函数就死亡了。使用这个filter会导致未定义行为，这是由它被创建那一刻起就决定了的。

现在，同样的问题也会出现在divisor的显式按引用捕获。

```
filters.emplace_back(
    [&divisor](int value)           // danger! ref to
    { return value % divisor == 0; } // divisor will
);
```

但通过显式的捕获，能更容易看到lambda的可行性依赖于变量divisor的生命周期。另外，写成这种形式能够提醒我们要注意确保divisor的生命周期至少跟lambda闭包一样长。比起"[&]"传达的意思，显式捕获能让人更容易想起“确保没有悬空变量”。

如果你知道一个闭包将会被马上使用（例如被传入到一个stl算法中）并且不会被拷贝，那么在lambda环境中使用引用捕获将不会有风险。在这种情况下，你可能会争论说，没有悬空引用的危险，就不需要避免使用默认的引用捕获模式。例如，我们的过滤lambda只会用做C++11中std::all_of的一个参数，返回满足条件的所有元素：

```
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...
}
```

的确如此，这是安全的做法，但这种安全是不确定的。如果发现lambda在其它上下文中很有用（例如作为一个函数被添加在filters容器中），然后拷贝粘贴到一个divisor变量已经死亡的，但闭包生命周期还没结束的上下文中，你又回到了悬空的使用上了。同时，在该捕获语句中，也没有特别提醒了你注意分析divisor的生命周期。

从长期来看，使用显式的局部变量和参数引用捕获方式，是更加符合软件工程规范的做法。

额外提一下，C++14支持了在lambda中使用auto来声明变量，上面的代码在C++14中可以进一步简化，ContElemT的别名可以去掉，if条件可以修改为：

```
if (std::all_of(begin(container), end(container),
                [&](const auto& value) // C++14
                { return value % divisor == 0; }))
```

一个解决问题的方法是，divisor按值捕获进去，也就是说可以按照以下方式来添加lambda：

```
filters.emplace_back(                           // now
    [=](int value) { return value % divisor == 0; } // divisor
);                                             // can't
                                               // dangle
```

这足以满足本实例的要求，但在通常情况下，按值捕获并不能完全解决悬空引用的问题。这里的问题是如果你按值捕获的是一个指针，你将该指针拷贝到lambda对应的闭包里，但这样并不能避免lambda外删除指针的行为，从而导致你的指针变成悬空指针。

也许你要抗议说：“这不可能发生。看过了第四章，我对智能指针的使用非常热衷。只有那些失败的C++98的程序员才会用裸指针和delete语句。”这也许是正确的，但却是不相关的，因为事实上你的确会使用裸指针，也的确存在被你删除的可能性。只不过在现代的C++编程风格中，不容易在源代码中显露出来而已。

假设在一个Widget类，可以实现向过滤容器添加条目：

```
class Widget {
public:
    …                           // ctors, etc.
    void addFilter() const; // add an entry to filters
private:
    int divisor;            // used in Widget's filter
};
```

这是Widget::addFilter的定义：

```
void Widget::addFilter() const
{
    filters.emplace_back(
        [=](int value) { return value % divisor == 0; }
    );
}
```

这个做法看起来是安全的代码，lambda依赖于变量divisor，但默认的按值捕获被拷贝进了lambda对应的所有比保重，这真的正确吗？

错误，完全错误。

闭包只会对lambda被创建时所在作用域里的非静态局部变量生效。在Widget::addFilter()的视线里，divisor并不是一个局部变量，而是Widget类的一个成员变量。它不能被捕获。如果默认捕获模式被删除，代码就不能编译了：

```
void Widget::addFilter() const
{
    filters.emplace_back(                                // error!
        [](int value) { return value % divisor == 0; }  // divisor
    );                                                   // not
}                                                        // available
```

另外，如果尝试去显式地按引用或者按值捕获divisor变量，也一样会编译失败，因为divisor不是这里的一个局部变量或者参数。

```
void Widget::addFilter() const
{
    filters.emplace_back(
      [divisor](int value)            // error! no local
      { return value % divisor == 0; } // divisor to capture
    );
}
```

因此这里的默认按值捕获并不是不会变量divisor，但它的确能够编译通过，这是怎么一回事呢？

解释就是这里隐式捕获了this指针。每一个非静态成员函数都有一个this指针，每次你使用一个类内的成员时都会使用到这个指针。例如，编译器会在内部将divisor替换成this->divisor。这里Widget::addFilter()的版本就是按值捕获了this。

```
void Widget::addFilter() const
{
    filters.emplace_back(
      [=](int value) { return value % divisor == 0; }
    );
}
```

真正被捕获的是Widget的this指针。编译器会将上面的代码看成以下的写法：

```
void Widget::addFilter() const
{
    auto currentObjectPtr = this;

    filters.emplace_back(
      [currentObjectPtr](int value)
      { return value % currentObject->divisor == 0; }
    );
}
```

明白了这个就相当于明白了lambda闭包的生命周期与Widget对象的关系，闭包内含有Widget的this指针的拷贝。特别是考虑以下的代码，再参考一下第四章的内容，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，就会创建一个过滤器，其生命周期依赖于由std::make_unique管理的Widget对象。即一个含有Widget this指针的过滤器。这个过滤器被添加到filters中，但当doSomeWork结束时，Widget会由std::unique_ptr去结束其生命。从这时起，filter会含有一个悬空指针。

这个特定的问题可以通过做一个局部拷贝去解决：

```
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
    );
}
```

事实上如果采用这种方法，默认的按值捕获也是可行的。

```
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？当你一开始捕获divisor的时候，默认的捕获模式就会自动将this指针捕获进来了。

在C++14中，一个更好的捕获成员变量的方式时使用通用的lambda捕获：

```
void Widget::addFilter() const
{
    filters.emplace_back(               // C++14:
        [divisor = divisor](int value)   // copy divisor to closure
        { return value % divisor == 0; } // use the copy
    );
}
```

这种通用的lambda捕获并没有默认的捕获模式，因此在C++14中，避免使用默认捕获模式的建议仍然时成立的。

使用默认的按值捕获还有另外的一个缺点，它们预示了相关的闭包是独立的并且不受外部数据变化的影响。一般来说，这是不对的。lambda并不会独立于局部变量和参数，但也没有不受静态存储生命周期的影响。一个定义在全局空间或者指定命名空间的全局变量，或者是一个声明为static的类内或文件内的成员。这些对象也能在lambda里使用，但它们不能被捕获。但按值引用可能会因此误导你，让你以为捕获了这些变量。参考下面版本的addDivisorFilter()函数：

```
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
}
```

随意地看了这份代码的读者可能看到"[=]"，就会认为“好的，lambda拷贝了所有使用的对象，因此这是独立的”。但上面的例子就表现了不独立闭包的一种情况。它没有使用任何的非static局部变量和形参，所以它没有捕获任何东西。然而lambda的代码引用了静态变量divisor，任何lambda被添加到filters之后，divisor都会递增。通过这个函数，会把许多lambda都添加到filiters里，但每一个lambda的行为都是新的（分别对应新的divisor值）。这个lambda是通过引用捕获divisor，这和默认的按值捕获表示的含义有着直接的矛盾。如果你一开始就避免使用默认的按值捕获模式，你就能解除代码的风险。

建议

- 默认的按引用捕获可能会导致悬空引用；
- 默认的按值引用对于悬空指针很敏感（尤其是this指针），并且它会误导人产生lambda是独立的想法；