Item 8: Prefer nullptr to 0 and NULL.

条款八:优先考虑nullptr而非0和NULL

你看这样对不对：字面值0是一个int不是指针。
如果C++发现在当前上下文只能使用指针，它会很不情愿的把0解释为指针，但是那是最后的退路。
一般来说C++的解析策略是把0看做int而不是指针。

实际上，NULL也是这样的。但在NULL的实现细节有些不确定因素，
因为实现被允许给NULL一个除了**int**之外的整型类型（比如**long**）。
这不常见，但也算不上问题所在。这里的问题不是NULL没有一个确定的类型，而是0和NULL都不是指针类型。

在C++98中，对指针类型和整型进行重载意味着可能导致奇怪的事情。
如果给下面的重载函数传递0或NULL，它们绝不会调用指针版本的重载函数：

```
void f(int);        //三个f的重载函数
void f(bool);
void f(void*);

f(0);               //调用f(int)而不是f(void*)

f(NULL);            //可能不会被编译，一般来说调用f(int),绝对不会调用f(void*)
```

而f(NULL)的不确定行为是由NULL的实现不同造成的。
如果NULL被定义为**0L**（指的是**0**为**long**类型），这个调用就具有二义性，因为从**long**到**int**的转换或从**long**到**bool**的转换或**0L**到**void*** 的转换都会被考虑。
有趣的是源代码表现出的意思（我指的是使用NULL调用f）和实际想表达的意思（我指的是用整型数据调用f）是相矛盾的。
这种违反直觉的行为导致C++98程序员都将避开同时重载指针和整型作为编程准则*[0]*。
在C++11中这个编程准则也有效，因为尽管我这个条款建议使用**nullptr**，可能很多程序员还是会继续使用**0**或**NULL**，哪怕**nullptr**是更好的选择。

nullptr的优点是它不是整型。
老实说它也不是一个指针类型，但是你可以把它认为是通用类型的指针。
nullptr的真正类型是**std::nullptr_t**，在一个完美的循环定义以后，**std::nullptr_t**又被定义为**nullptr**。
std::nullptr_t可以转换为指向任何内置类型的指针，这也是为什么我把它叫做通用类型的指针。

使用**nullptr**调用f将会调用**void***版本的重载函数，因为**nullptr**不能被视作任何整型：

```
f(nullptr);         //调用重载函数f的f(void*)版本
```

使用**nullptr***代替**0**和**NULL**可以避开了那些令人奇怪的函数重载决议，这不是它的唯一优势。
它也可以使代码表意明确，尤其是当和**auto**一起使用时。
举个例子，假如你在一个代码库中遇到了这样的代码：

```
auto result = findRecord( /* arguments */ );
if (result == 0) {
    …
}
```

如果你不知道findRecord返回了什么（或者不能轻易的找出），那么你就不太清楚到底result是一个指针类型还是一个整型。
毕竟，**0**也可以像我们之前讨论的那样被解析。
但是换一种假设如果你看到这样的代码：

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

这就没有任何歧义：**result**的结果一定是指针类型。

当模板出现时**nullptr**就更有用了。
假如你有一些函数只能被合适的已锁互斥量调用。
每个函数都有一个不同类型的指针：

```
int    f1(std::shared_ptr<Widget> spw);  // 只能被合适的
double f2(std::unique_ptr<Widget> upw);  // 已锁互斥量调
bool   f3(Widget* pw);                   // 用
```

如果这样传递空指针：

```
std::mutex f1m, f2m, f3m;          // 互斥量f1m，f2m，f3m，各种用于f1，f2，f3函数
using MuxGuard =                   // C++11的typedef，参见Item9
    std::lock_guard<std::mutex>;
     …
{
    MuxGuard g(f1m);               // 为f1m上锁
    auto result = f1(0);           // 向f1传递控制空指针
}                                  // 解锁
…
{
    MuxGuard g(f2m);               // 为f2m上锁
    auto result = f2(NULL);        // 向f2传递控制空指针
}                                  // 解锁
…
{
  MuxGuard g(f3m);                 // 为f3m上锁
  auto result = f3(nullptr);       // 向f3传递控制空指针
}                                  // 解锁
```

令人遗憾前两个调用没有使用**nullptr**，但是代码可以正常运行，这也许对一些东西有用。
但是重复的调用代码——为互斥量上锁，调用函数，解锁互斥量——更令人遗憾。它让人很烦。
模板就是被设计于减少重复代码，所以让我们模板化这个调用流程：

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

如果你对函数返回类型** (auto ... -> decltype(func(ptr)) **感到困惑不解，Item3可以帮助你。**
在C++14中代码的返回类型还可以被简化为decltype(auto)**:

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

可以写这样的代码调用**lockAndCall**模板（两个都算）：

```
auto result1 = lockAndCall(f1, f1m, 0);          // 错误!
…
auto result2 = lockAndCall(f2, f2m, NULL);       // 错误!
…
auto result3 = lockAndCall(f3, f3m, nullptr);    // 没问题
```

代码虽然可以这样写，但是就像注释中说的，前两个情况不能通过编译。
在第一个调用中存在的问题是当0被传递给**lockAndCall**模板，模板类型推导会尝试去推导实参类型，
0的类型总是**int**，所以**int**版本的实例化中的func会被int类型的实参调用。
这与**f1**期待的参数**std::shared_ptr**不符。
传递**0**本来想表示空指针，结果**f1**得到的是和它相差十万八千里的**int**。
把**int**类型看做**std::shared_ptr**类型自然是一个类型错误。
在模板**lockAndCall**中使用**0**之所以失败是因为得到的是**int**但实际上模板期待的是一个
std::shared_ptr

第二个使用**NULL**调用的分析也是一样的。当**NULL**被传递给**lockAndCall**，形参ptr被推导为整型*[1]*，
然后当ptr——一个int或者类似int的类型——传递给f2的时候就会出现类型错误。当ptr被传递给f3的时候，
隐式转换使**std::nullptr_t**转换为**Widget***，因为**std::nullptr_t**可以隐式转换为任何指针类型。

模板类型推导将**0**和**NULL**推导为一个错误的类型，这就导致它们的替代品**nullptr**很吸引人。
使用**nullptr**，模板不会有什么特殊的转换。
另外，使用**nullptr**不会让你受到同重载决议特殊对待**0**和**NULL**一样的待遇。
当你想用一个空指针，使用**nullptr**，不用**0**或者**NULL**。

记住

- 优先考虑nullptr而非0和NULL
- 避免重载指针和整型

译注

[0] 请务必注意结合上下文使用这条规则

[1] 由于依赖于具体实现所以不一定是整数类型，所以用整型泛指int,long等类型