Item 11:优先考虑使用deleted函数而非使用未定义的私有声明

条款11:优先考虑使用deleted函数而非使用未定义的私有声明

如果你写的代码要被其他人使用，你不想让他们调用某个特殊的函数，你通常不会声明这个函数。无声明，不函数。简简单单！但有时C++会给你自动声明一些函数，如果你想防止客户调用这些函数，事情就不那么简单了。

上述场景见于特殊的成员函数，即当有必要时C++自动生成的那些函数。Item 17 详细讨论了这些函数，但是现在，我们只关心拷贝构造函数和拷贝赋值运算符重载。This chapter is largely devoted to common practices in
C++98 that have been superseded by better practices in C++11, and in C++98, if you want to suppress use of a member function, it's almost always the copy constructor, the assignment operator, or both.

在C++98中防止调用这些函数的方法是将它们声明为私有成员函数。举个例子，在C++ 标准库*iostream*继承链的顶部是模板类`basic_ios`。所有`istream`和`ostream`类都继承此类(直接或者间接)。拷贝`istream`和`ostream`是不合适的，因为要进行哪些操作是模棱两可的。比如一个`istream`对象，代表一个输入值的流，流中有一些已经被读取，有一些可能马上要被读取。如果一个`istream`被拷贝，需要像拷贝将要被读取的值那样也拷贝已经被读取的值吗？解决这个问题最好的方法是不定义这个操作。直接禁止拷贝流。

要使`istream`和`ostream`类不可拷贝，`basic_ios`在C++98中是这样声明的(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
};
```

将它们声明为私有成员可以防止客户端调用这些函数。故意不定义它们意味着假如还是有代码用它们就会在链接时引发缺少函数定义(missing function definitions)错误。

在C++11中有一种更好的方式，只需要使用相同的结尾：用`= delete`将拷贝构造函数和拷贝赋值运算符标记为`deleted`函数。上面相同的代码在C++11中是这样声明的:

```
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;
    …
};
```

删除这些函数(译注：添加"= delete")和声明为私有成员可能看起来只是方式不同，别无其他区别。其实还有一些实质性意义。`deleted`函数不能以任何方式被调用，即使你在成员函数或者友元函数里面调用`deleted`函数也不能通过编译。这是较之C++98行为的一个改进，后者不正确的使用这些函数在链接时才被诊断出来。

通常，`deleted`函数被声明为*public*而不是*private*.这也是有原因的。当客户端代码试图调用成员函数，C++会在检查`deleted`状态前检查它的访问性。当客户端代码调用一个私有的`deleted`函数，一些编译器只会给出该函数是*private*的错误(译注：而没有诸如该函数被`deleted`修饰的错误)，即使函数的访问性不影响它的使用。所以值得牢记，如果要将老代码的"私有且未定义"函数替换为`deleted`函数时请一并修改它的访问性为*public*，这样可以让编译器产生更好的错误信息。

`deleted`函数还有一个重要的优势是任何函数都可以标记为`deleted`，而只有*private*只能修饰成员函数。假如我们有一个非成员函数，它接受一个整型参数，检查它是否为幸运数：

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

C++有沉重的C包袱，使得含糊的、能被视作数值的任何类型都能隐式转换为`int`，但是有一些调用可能是没有意义的：

```
if (isLucky('a')) … // 字符'a'是幸运数?
if (isLucky(true)) … // "true"是?
if (isLucky(3.5)) … // 难道判断它的幸运之前还要先截尾成3?
```

如果幸运数必须真的是整数，我们该禁止这些调用通过编译。
其中一种方法就是创建`deleted`重载函数，其参数就是我们想要过滤的类型：

```
bool isLucky(int number); // 原始版本
bool isLucky(char) = delete; // 拒绝char
bool isLucky(bool) = delete; // 拒绝bool
bool isLucky(double) = delete; // 拒绝float和double
```

(上面double重载版本的注释说拒绝float和double可能会让你惊讶，但是请回想一下：将`float`转换为`int`和`double`，C++更喜欢转换为`double`。使用`float`调用`isLucky`因此会调用`double`重载版本，而不是`int`版本。好吧，它也会那么去尝试。事实是调用被删除的`double`重载版本不能通过编译。不再惊讶了吧。)

虽然`deleted`寒暑假不能被使用，它它们还是存在于你的程序中。也即是说，重载决议会考虑它们。这也是为什么上面的函数声明导致编译器拒绝一些不合适的函数调用。

```
if (isLucky('a')) … //错误! 调用deleted函数
if (isLucky(true)) … // 错误!
if (isLucky(3.5f)) … // 错误!
```

另一个`deleted`函数用武之地（private成员函数做不到的地方）是禁止一些模板的实例化。
假如你要求一个模板仅支持原生指针（尽管第四章建议使用智能指针代替原生指针）

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

在指针的世界里有两种特殊情况。一是`void*`指针，因为没办法对它们进行解引用，或者加加减减等。另一种指针是`char*`，因为它们通常代表C风格的字符串，而不是正常意义下指向单个字符的指针。这两种情况要特殊处理，在`processPointer`模板里面，我们假设正确的函数应该拒绝这些类型。
也即是说，`processPointer`不能被`void*`和`char*`调用。
要想确保这个很容易，使用`delete`标注模板实例：

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

现在如果使用`void*`和`char*`调用`processPointer`就是无效的，按常理说`const void*`和`const void*`也应该无效，所以这些实例也应该标注`delete`：

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

如果你想做得更彻底一些，你还要删除`const volatile void*`和`const volatile char*`重载版本，另外还需要一并删除其他标准字符类型的重载版本：`std::wchar_t`,`std::char16_t`和`std::char32_t`。

有趣的是，如果的类里面有一个函数模板，你可能想用`private`（经典的C++98惯例）来禁止这些函数模板实例化，但是不能这样做，因为不能给特化的模板函数指定一个不同（于函数模板）的访问级别。如果`processPointer`是类`Widget`里面的模板函数， 你想禁止它接受`void*`参数，那么通过下面这样C++98的方法就不能通过编译：
compile:

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

问题是模板特例化必须位于一个命名空间作用域，而不是类作用域。`delete`不会出现这个问题，因为它不需要一个不同的访问级别，且他们可以在类外被删除（因此位于命名空间作用域）：

```
class Widget {
public:
    …
    template<typename T>
    void processPointer(T* ptr)
    { … }
    …
};
template<>
void Widget::processPointer<void>(void*) = delete; // 还是public，但是已经被删除了
```

事实上C++98的最佳实践即声明函数为*private*但不定义是在做C++11 delete函数要做的事情。作为模仿者，C++98的方法不是十全十美。它不能在类外正常工作，不能总是在类中正常工作，它的罢工可能直到链接时才会表现出来。所以请坚定不移的使用`delete`函数。

记住：

- 比起声明函数为private但不定义，使用delete函数更好
- 任何函数都能`delete`，包括非成员函数和模板实例

译注：
+本条款`delete`，`deleted`,删除视情况使用，都表示一个意思.`删除函数`和`delete函数`也是如此

- 函数模板意指未特化前的源码，模板函数则倾向于模板实例化后的函数