C 类型转换时定义非成员函数 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/245/2021_2022_C___E7_B1_BB_E5_9E_8B_E8_c97_245956.htm 《C 箴言：声明为非成员函数的时机》阐述了为什么只有 non-member functions（非成员函数）适合于应用到所有 arguments（实参）的 implicit type conversions（隐式类型转换），而且它还作为一个示例使用了一个 Rational class 的 operator* function。我建议你在阅读本文之前先熟悉那个示例，因为本文进行了针对《C 箴言：声明为非成员函数的时机》中的示例做了一个无伤大雅（模板化 Rational 和 operator*）的扩展讨论： template class Rational { public: Rational(const Tamp. denominator = 1). // are now passed by reference const T numerator() const. // see《C 箴言：避免返回对象内部构件的句柄》for why return const T denominator() const. // values are still passed by value, ... // Item 3 for why they’re const }. template const Rational operator*(const Rationalamp. rhs) { ... } 就像在《C 箴言：声明为非成员函数的时机》中，我想要支持 mixed-mode arithmetic（混合模式运算），所以我们要让下面这些代码能够编译。我们指望它能，因为我们使用了和 Item 24 中可以工作的代码相同的代码。仅有的区别是 Rational 和 operator* 现在是 templates（模板）： Rational oneHalf(1, 2). // this example is from 《C 箴言：声明为非成员函数的时机》, // except Rational is now a template Rational result = oneHalf * 2. // error! won’t compile 编译失败的事实暗示对于模板化 Rational 来说，有某些东西和 non-template（非模板）版本不同，而且确实存在。在《C 箴言：声明为非成员函数的时机》中，编

译器知道我们想要调用什么函数（取得两个 Rationals 的 operator*），但是在这里，编译器不知道我们想要调用哪个函数。作为替代，它们试图断定要从名为 operator* 的 template（模板）中实例化出（也就是创建）什么函数。它们知道它们假定实例化出的某个名为 operator* 的函数取得两个 Rational 类型的参数，但是为了做这个实例化，它们必须断定 T 是什么。问题在于，它们做不到。 在推演 T 的尝试中，它们会察看被传入 operator* 的调用的 arguments（实参）的类型。在当前情况下，类型为 Rational（oneHalf 的类型）和 int（2 的类型）。每一个参数被分别考察。 使用 oneHalf 的推演很简单。operator* 的第一个 parameter（形参）被声明为 Rational 类型，而传入 operator* 的第一个 argument（实参）(oneHalf) 是 Rational 类型，所以 T 一定是 int。不幸的是，对其它参数的推演没那么简单。operator* 的第二个 parameter（形参）被声明为 Rational 类型，但是传入 operator* 的第二个 argument（实参）(2) 的 int 类型。在这种情况下，让编译器如何断定 T 是什么呢？你可能期望它们会使用 Rational 的 non-explicit constructor（非显式构造函数）将 2 转换成一个 Rational，这样就使它们推演出 T 是 int，但是它们不这样做。它们不这样做是因为在 template argument deduction（模板实参推演）过程中从不考虑 implicit type conversion functions（隐式类型转换函数）。从不。这样的转换可用于函数调用过程，这没错，但是在你可以调用一个函数之前，你必须知道哪个函数存在。为了知道这些，你必须为相关的 function templates（函数模板）推演出 parameter types（参数类型）（以便你可以实例化出合适的函数）。但是在 template argument deduction（模板

实参推演）过程中不考虑经由 constructor（构造函数）调用的 implicit type conversion（隐式类型转换）。《C 箴言：声明为非成员函数的时机》不包括 templates（模板），所以 template argument deduction（模板实参推演）不是一个问题，现在我们在 C 的 template 部分，这是主要问题。 在一个 template class（模板类）中的一个 friend declaration（友元声明）可以指涉到一个特定的函数，我们可以利用这一事实为受到 template argument deduction（模板实参推演）挑战的编译器解围。这就意味着 class Rational 可以为 Rational 声明作为一个 friend function（友元函数）的 operator*。class templates（类模板）不依靠 template argument deduction（模板实参推演）（这个过程仅适用于 function templates（函数模板）），所以 T 在 class Rational 被实例化时总是已知的。通过将适当的 operator* 声明为 Rational class 的一个 friend（友元）使其变得容易： template class Rational { public: ... friend // declare operator* const Rational operator*(const Rationalamp. rhs). // below for details) }. template // define operator* const Rational operator*(const Rationalamp. rhs) { ... } 现在我们对 operator* 的混合模式调用可以编译了，因为当 object oneHalf 被声明为 Rational 类型时，class Rational 被实例化，而作为这一过程的一部分，取得 Rational parameters（形参）的 friend function（友元函数）operator* 被自动声明。作为已声明函数（并非一个 function template（函数模板）），在调用它的时候编译器可以使用 implicit conversion functions（隐式转换函数）（譬如 Rational 的 non-explicit constructor（非显式构造函数）），而这就是它们如何使得混合模式调用成功的。 唉，在这里的上

下文中，“ 成功 ”是一个可笑的词，因为尽管代码可以编译，但是不能连接。但是我们过一会儿再处理它，首先我想讨论一下用于在 Rational 内声明 operator* 的语法。 在一个 class template（类模板）内部，template（模板）的名字可以被用做 template（模板）和它的 parameters（参数）的缩写，所以，在 Rational 内部，我们可以只写 Rational 代替 Rational。在本例中这只为我们节省了几个字符，但是当有多个参数或有更长的参数名时，这既能节省击键次数又能使最终的代码显得更清晰。我把这一点提前，是因为 operator* 被声明为取得并返回 Rationals，而不是 Rationals。它就像如下这样声明 operator* 一样合法： template class Rational { public: ... friend const Rational operator*(const Rationalamp. rhs). ... }. 然而，使用缩写形式更简单（而且更常用）。 现在返回到连接问题。混合模式代码编译，因为编译器知道我们想要调用一个特定的函数（取得一个 Rational 和一个 Rational 的 operator*），但是那个函数只是在 Rational 内部声明，而没有在此处定义。我们的意图是让 class 之外的 operator* template（模板）提供这个定义，但是这种方法不能工作。如果我们自己声明一个函数（这就是我们在 Rational template（模板）内部所做的事），我们就有责任定义这个函数。当前情况是，我们没有提供定义，这也就是连接器为什么不能找到它。 让它能工作的最简单的方法或许就是将 operator* 的本体合并到它的 declaration（定义）中： template class Rational { public: ... friend const Rational operator*(const Rationalamp. rhs) { return Rational(lhs.numerator() * rhs.numerator(), // same impl lhs.denominator() * rhs.denominator()). // as in } //《C 箴言：声

明为非成员函数的时机》}. 确实，这样就可以符合预期地工作：对 operator* 的混合模式调用现在可以编译，连接，并运行。万岁！ 关于此技术的一个有趣的观察结论是 friendship 的使用对于访问 class 的 non-public parts（非公有构件）的需求并没有起到什么作用。为了让所有 arguments（实参）的 type conversions（类型转换）成为可能，我们需要一个 non-member function（非成员函数）（《C 箴言：声明为非成员函数的时机》 依然适用）；而为了能自动实例化出适当的函数，我们需要在 class 内部声明这个函数。在一个 class 内部声明一个 non-member function（非成员函数）的唯一方法就是把它做成一个 friend（友元）。那么这就是我们做的。反传统吗？是的。有效吗？毫无疑问。 就像《C 箴言：理解inline化的介入和排除》阐述的，定义在一个 class 内部的函数被隐式地声明为 inline（内联），而这也包括像 operator* 这样的 friend functions（友元函数）。你可以让 operator* 不做什么事情，只是调用一个定义在这个 class 之外的 helper function（辅助函数），从而让这样的 inline declarations（内联声明）的影响最小化。在本文的这个示例中，没有特别指出这样做，因为 operator* 已经可以实现为一个 one-line function（单行函数），但是对于更复杂的函数体，这样做也许是合适的。"have the friend call a helper"（“ 让友元调用辅助函数 ”）的方法还是值得注意一下的。 Rational 是一个 template（模板）的事实意味着那个 helper function（辅助函数）通常也是一个 template（模板），所以典型情况下在头文件中定义 Rational 的代码看起来大致如下： template class Rational. // declare // Rational // template template // declare const Rational

doMultiply(const Rationalamp. rhs). // template template class Rational { public: ... friend const Rational operator*(const Rationalamp. rhs) // Have friend { return doMultiply(lhs, rhs). } // call helper ... }. 多数编译器基本上会强迫你把所有的 template definitions（模板定义）都放在头文件中，所以你可能同样需要在你的头文件中定义 doMultiply。（就像 Item 30 阐述的，这样的 templates（模板）不需要 inline（内联）。）可能看起来就像这样： template // define const Rational doMultiply(const Rationalamp. rhs) // template in { // header file, return Rational(lhs.numerator() * rhs.numerator(), // if necessary lhs.denominator() * rhs.denominator()). } 当然，作为一个 template（模板），doMultiply 不支持混合模式乘法，但是它不需要。它只被 operator* 调用，而 operator* 支持混合模式运算！本质上，函数 operator* 支持为了确保被相乘的是两个 Rational objects 而必需的各种 type conversions（类型转换），然后它将这两个 objects 传递给一个 doMultiply template（模板）的适当的实例化来做实际的乘法。配合行动，不是吗？ Things to Remember 在写一个提供了 class template（类模板），而这个 class template（类模板）提供了一个函数，这个函数指涉到支持所有 parameters（参数）的 implicit type conversions（隐式类型转换）的 template（模板）的时候，把这些函数定义为 class template（类模板）内部的 friends（友元）。