

[About](#)[Projects](#)[Blog](#)[Contact](#)

知无涯之C++ typename的起源与用法

08 May 2014 • 6 min. read • 31 Comments

侯捷在Effective C++的中文版译序中提到:

C++的难学，还在于它提供了四种不同（但相辅相成）的程序设计思维模式：procedural-based, object-based, object-oriented, generics

对于较少使用最后一种泛型编程的我来说，程序设计基本上停留在前三种思维模式当中。虽说不得窥见高深又现代的泛型技术，但前三种思维模式已几乎满足我所遇到的所有需求，因此一直未曾深入去了解泛型编程。

目录

- 起因
- typename的常见用法
- typename的来源
- 一些关键概念
 - 限定名和非限定名
 - 依赖名和非依赖名
 - 类作用域
- 引入typename的真实原因
 - 一个例子
 - 问题浮现
 - 千呼万唤始出来
 - 不同编译器对错误情况的处理
 - 使用typename的规则
 - 其它例子
 - 再看常见用法
- 参考
- 写在结尾

起因

近日，看到这样一行代码：

```
typedef typename __type_traits<T>::has_trivial_destructor trivial_destructor;
```

虽说已经有多年的C++经验，但上面这短短一行代码却看得我头皮发麻。看起来它应该是定义一个类型别名，但是 `typedef` 不应该是像这样使用么， `typedef +原类型名+新类型名`：

```
typedef char* PCHAR;
```

可为何此处多了一个 `typename`？另外 `__type_traits` 又是什么？看起来有些眼熟，想起之前在Effective C++上曾经看过 `traits` 这一技术的介绍，和这里的 `__type_traits` 有点像。只是一直未曾遇到需要 `traits` 的时候，所以当时并未仔细研究。然而STL中大量的充斥着各种各样的 `traits`，一查才发现原来它是一种非常高级的技术，在更现的高级语言中已经很普遍。因此这次花了些时间去学习它，接下来还会有另一篇文章来详细介绍C++的 `traits` 技术。在这里，我们暂时忘记它，仅将它当成一个普通的类，先来探讨一下这个多出来的 `typename` 是怎么回事？

typename的常见用法

对于 `typename` 这个关键字，如果你熟悉C++的模板，一定会知道它有这样一种最常见的用法(代码摘自C++ Primer)：

```
// implement strcmp-like generic compare function
// returns 0 if the values are equal, 1 if v1 is larger, -1 if v1 is smaller
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

也许你会想到上面这段代码中的 `typename` 换成 `class` 也一样可以，不错！那么这里便有了疑问，这两种方式有区别么？查看C++ Primer之后，发现两者完全一样。那么为什么C++要同时支持这两种方式呢？既然 `class` 很早就已经有了，为什么还要引入 `typename` 这一关键字呢？问的好，这里面有一段鲜为人知的历史（也许只是我不知道:-)）。带着这些疑问，我们开始探寻之旅。

typename的来源

对于一些更早接触C++的朋友，你可能知道，在C++标准还未统一时，很多旧的编译器只支持 `class`，因为那时C++并没有 `typename` 关键字。记得我在学习C++时就曾在某本C++书籍上看过类似的注意事项，告诉我们如果使用 `typename` 时编译器报错的话，那么换成 `class` 即可。

一切归结于历史。

Stroustrup在最初起草模板规范时，他曾考虑到为模板的类型参数引入一个新的关键字，但是这样做很可能会破坏已经写好的很多程序（因为 `class` 已经使用了很长一段时间）。但是更重要的原因是，在当时看来，`class` 已完全足够胜任模板的这一需求，因此，为了避免引起不必要的麻烦，他选择了妥协，重用已有的 `class` 关键字。所以只到ISO C++标准出来之前，想要指定模板的类型参数只有一种方法，那便是使用 `class`。这也解释了为什么很多旧的编译器只支持 `class`。

但是对很多人来说，总是不习惯 `class`，因为从其本来存在的目的来说，是为了区别于语言的内置类型，用于声明一个用户自定义类型。那么对于下面这个模板函数的定义（相对于上例，仅将 `typename` 换成了 `class`）：

```
template <class T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

从表面上看起来就好像这个模板的参数应该只支持**用户自定义类型**，所以使用**语言内置类型**或者**指针**来调用该模板函数时总会觉得有一丝奇怪（虽然并没有错误）：

```
int v1 = 1, v2 = 2;
int ret = compare(v1, v2);

int *pv1 = NULL, *pv2 = NULL;
ret = compare(pv1, pv2);
```

令人感到奇怪的原因是，`class` 在类和模板中表现的意义看起来存在一些不一致，前者针对用户自定义类型，而后者包含了语言内置类型和指针。也正因为如此，人们似乎觉得当时没有引入一个新的关键字可能是一个错误。

这是促使标准委员会引入新关键字的一个因素，但其实还有另外一个更加重要的原因，和文章最开始那行代码相关。

一些关键概念

在我们揭开真实原因的面纱之前，先保持一点神秘感，因为为了更好的理解C++标准，有几个重要的概念需要先行介绍一下。

限定名和非限定名

限定名(qualified name)，故名思义，是限定了命名空间的名称。看下面这段代码，`cout` 和 `endl` 就是限定名：

```
#include <iostream>

int main() {
    std::cout << "Hello world!" << std::endl;
}
```

`cout` 和 `endl` 前面都有 `std::`，它限定了 `std` 这个命名空间，因此称其为限定名。

如果在上面这段代码中，前面用 `using std::cout;` 或者 `using namespace std;`，然后使用时只用 `cout` 和 `endl`，它们的前面不再有空间限定 `std::`，所以此时的 `cout` 和 `endl` 就叫做非限定名(unqualified name)。

依赖名和非依赖名

依赖名(dependent name)是指依赖于模板参数的名称，而非依赖名(non-dependent name)则相反，指不依赖于模板参数的名称。看下面这段代码：

```
template <class T>
class MyClass {
```

```
int i;
vector<int> vi;
vector<int>::iterator vitr;

T t;
vector<T> vt;
vector<T>::iterator viter;
};
```

因为是内置类型，所以类中前三个定义的类型在声明这个模板类时就已知。然而对于接下来的三行定义，只有在模板实例化时才能知道它们的类型，因为它们都依赖于模板参数 `T`。因此，`T`，`vector<T>` 和 `vector<T>::iterator` 称为依赖名。前三个定义叫做非依赖名。

更为复杂一点，如果用了 `typedef T U; U u;`，虽然 `T` 没再出现，但是 `U` 仍然是依赖名。由此可见，不管是直接还是间接，只要依赖于模板参数，该名称就是依赖名。

类作用域

在类外部访问类中的名称时，可以使用类作用域操作符，形如 `MyClass::name` 的调用通常存在三种：静态数据成员、静态成员函数和嵌套类型：

```
struct MyClass {
    static int A;
    static int B();
    typedef int C;
}
```

`MyClass::A`，`MyClass::B`，`MyClass::C` 分别对应着上面三种。

引入typename的真实原因

结束以上三个概念的讨论，让我们接着揭开 `typename` 的神秘面纱。

一个例子

在Stroustrup起草了最初的模板规范之后，人们更加无忧无虑的使用了 `class` 很长一段时间。可是，随着标准化C++工作的到来，人们发现了模板这样一种定义：

```
template <class T>
void foo() {
    T::iterator * iter;
    // ...
}
```

这段代码的目的是什么？多数人第一反应可能是：作者想定义一个指针 `iter`，它指向的类型是包含在类作用域 `T` 中的 `iterator`。可能存在这样一个包含 `iterator` 类型的结构：

```
struct ContainsAType {
    struct iterator { /*...*/ };
    // ...
};
```

然后像这样实例化 `foo`：

```
foo<ContainsAType>();
```

这样一来，`iter` 那行代码就很明显了，它是一个 `ContainsAType::iterator` 类型的指针。到目前为止，咱们猜测的一点不错，一切都看起来很好。

问题浮现

在类作用域一节中，我们介绍了三种名称，由于 `MyClass` 已经是一个完整的定义，因此编译期它的类型就可以确定下来，也就是说 `MyClass::A` 这些名称对于编译器来说也是已知的。

可是，如果是像 `T::iterator` 这样呢？`T` 是模板中的类型参数，它只有等到模板实例化时才会知道是哪种类型，更不用说内部的 `iterator`。通过前面类作用域一节的介绍，我们可以知道，`T::iterator` 实际上可以是以下三种中的任何一种类型：

- 静态数据成员
- 静态成员函数
- 嵌套类型

前面例子中的 `ContainsAType::iterator` 是嵌套类型，完全没有问题。可如果是静态数据成员呢？如果实例化 `foo` 模板函数的类型是像这样的：

```
struct ContainsAnotherType {  
    static int iterator;  
    // ...  
};
```

然后如此实例化 `foo` 的类型参数：

```
foo<ContainsAnotherType>();
```

那么，`T::iterator * iter;` 被编译器实例化为 `ContainsAnotherType::iterator * iter;`，这是什么？前面是一个静态成员变量而不是类型，那么这便成了一个乘法表达式，只不过 `iter` 在这里没有定义，编译器会报错：

error C2065: 'iter' : undeclared identifier

但如果 `iter` 是一个全局变量，那么这行代码将完全正确，它是表示计算两数相乘的表达式，返回值被抛弃。

同一行代码能以两种完全不同的方式解释，而且在模板实例化之前，完全没有办法来区分它们，这绝对是滋生各种bug的温床。这时C++标准委员会再也忍不住了，与其到实例化时才能知道到底选择哪种方式来解释以上代码，委员会决定引入一个新的关键字，这就是 `typename`。

千呼万唤始出来

我们来看看C++标准：

A name used in a template declaration or definition and that is dependent on a template-parameter is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword `typename`.

对于用于模板定义的依赖于模板参数的名称，只有在实例化的参数中存在这个类型名，或者这个名称前使用了 `typename` 关键字来修饰，编译器才会将该名称当成是类型。除了以上这两种情况，绝不会被当成是类型。

因此，如果你想直接告诉编译器 `T::iterator` 是类型而不是变量，只需用 `typename` 修饰：

```
template <class T>
void foo() {
    typename T::iterator * iter;
    // ...
}
```

这样编译器就可以确定 `T::iterator` 是一个类型，而不再需要等到实例化时期才能确定，因此消除了前面提到的歧义。

不同编译器对错误情况的处理

但是如果仍然用 `ContainsAnotherType` 来实例化 `foo`，前者只有一个叫 `iterator` 的静态成员变量，而后者需要的是一个类型，结果会怎样？我在Visual C++ 2010和g++ 4.3.4上分别做了实验，结果如下：

Visual C++ 2010仍然报告了和前面一样的错误：

error C2065: 'iter' : undeclared identifier

虽然我们已经用关键字 `typename` 告诉了编译器 `iterator` 应该是一个类型，但是用一个定义了 `iterator` 变量的结构来实例化模板时，编译器却选择忽略了此关键字。出现错误只是由于 `iter` 没有定义。

再来看看g++如何处理这种情况，它的错误信息如下：

```
In function 'void foo() [with T = ContainsAnotherType]': instantiated from here error: no type named 'iterator' in 'struct ContainsAnotherType'
```

g++在 `ContainsAnotherType` 中没有找到 `iterator` 类型，所以直接报错。它并没有尝试以另外一种方式来解释，由此可见，在这点上，g++更加严格，更遵循C++标准。

使用typename的规则

最后这个规则看起来有些复杂，可以参考MSDN：

- `typename`在下面情况下**禁止**使用：
 - 模板定义之外，即`typename`只能用于模板的定义中
 - 非限定类型，比如前面介绍过的 `int` , `vector<int>` 之类
 - 基类列表中，比如 `template <class T> class C1 : T::InnerType` 不能在 `T::InnerType` 前面加`typename`
 - 构造函数的初始化列表中
- 如果类型是依赖于模板参数的限定名，那么在它之前**必须**加`typename`(除非是基类列表，或者在类的初始化成员列表中)
- 其它情况下`typename`是**可选**的，也就是说对于一个不是依赖名的限定名，该名称是可选的，例如 `vector<int> vi;`

其它例子

对于不会引起歧义的情况，仍然需要在前面加 `typename` ，比如：

```
template <class T>
void foo() {
    typename T::iterator iter;
```

```
// ...  
}
```

不像前面的 `T::iterator * iter` 可能会被当成乘法表达式，这里不会引起歧义，但仍需加 `typename` 修饰。

再看下面这种：

```
template <class T>  
void foo() {  
    typedef typename T::iterator iterator_type;  
    // ...  
}
```

是否和文章刚开始的那行令人头皮发麻的代码有些许相似？没错！现在终于可以解开 `typename` 之迷了，看到这里，我相信你也一定可以解释那行代码了，我们再看一眼：

```
typedef typename __type_traits<T>::has_trivial_destructor trivial_destructor;
```

它是将 `__type_traits<T>` 这个模板类中的 `has_trivial_destructor` 嵌套类型定义一个叫做 `trivial_destructor` 的别名，清晰明了。

再看常见用法

既然 `typename` 关键字已经存在，而且它也可以用于最常见的指定模板参数，那么为什么不废除 `class` 这一用法呢？答案其实也很明显，因为在最终的标准出来之前，所有已存在的书、文章、教学、代码中都是使用的是 `class`，可以想像，如果标准不再支持 `class`，会出现什么情况。

对于指定模板参数这一用法，虽然 `class` 和 `typename` 都支持，但就个人而言我还是倾向使用 `typename` 多一些，因为我始终过不了 `class` 表示用户定义类型这道坎。另外，从语义上来说，`typename` 比 `class` 表达的更为清楚。C++ Primer也建议使用 `typename`：

使用关键字**typename**代替关键字**class**指定模板类型形参也许更为直观，毕竟，可以使用内置类型（非类类型）作为实际的类型形参，而且，**typename**更清楚地指明后面的名字是一个类型名。但是，关键字**typename**是作为标准C++的组成部分加入到C++中的，因此旧的程序更有可能只用关键字**class**。

参考

1. C++ Primer
2. Effective C++
3. A Description of the C++ typename keyword
4. 维基百科**typename**
5. 另外关于 `typename` 的历史，Stan Lippman写过一篇文章，Stan Lippman何许人，也许你不知道他的名字，但看完这些你一定会发出，“哦，原来是他！”：他是 *C++ Primer*, *Inside the C++ Object Model*, *Essential C++*, *C# Primer* 等著作的作者，另外他曾是 Visual C++的架构师。
6. 在StackOverflow上有一个非常深入的回答，感谢@Emer 在本文评论中提供此链接。

写在结尾

一个简单的关键字就已经充满曲折，这可以从一个角度反映出一门语言的发展历程，究竟要经历多少决断、波折与妥协，最终才发展成为现在的模样。在一个特定的时期，由于历史、技术、思想等各方面的因素，设计总会向现实做出一定的让步，出现一些“不完美”的设计，为了保持向后兼容，有些“不完美”的历史因素被保留了下来。现在我可以理解经常为人所诟病的Windows操作系统，Intel芯片，IE浏览器，Visual C++等，为了保持向后兼容，不得不在新的设计中仍然保留这些“不完美”，虽然带来的是更多的优秀特性，但有些人却总因为这些历史因素而唾弃它们，也为自己曾有一样的举动而羞愧不已。但也正是这些“不完美”的出现，才让人们在后续的设计中更加注意，站在前人的肩膀上，做出更好，更完善的设计，于是科技才不断向前推进。

然而也有一些敢于大胆尝试的例子，比如C++ 11，它的变化之大甚至连Stroustrup都说它像一门新语言。对于有着30余年历史的“老”语言，不仅没有被各种新贵击溃，反而在不断向晚辈们借鉴，吸纳一些好的特性，老而弥坚，这十分不易。还有Python 3，为了清理2.x版本

中某些语法方面的问题，打破了与2.x版本的向后兼容性，这种牺牲向后兼容换取进步的做法固然延缓了新版本的接受时间，但我相信这是向前进步的阵痛。Guido van Rossum的这种破旧立新的魄力实在让人钦佩，至于这种做法能否最终为人们所接受，一切交给历史来检验。

(全文完)

feihu

2014.05.08 于 Shenzhen

31条评论

libfeihu

1 登录 ▾

♥ 推荐 6

🔗 分享

评分最高 ▾



加入讨论...

通过以下方式登录

或注册一个 DISQUS 帐号 (?)

姓名



Diu · 1个月前

写的太好了。怀着非常感激的心情阅读完毕。感谢作者伟大的工作（于我真的是伟大了）

^ | v · 回复 · 分享 ▾



feihu 管理员 ➔ Diu · 1个月前

真的是言重了，没想到这样一篇算是晦涩，偏门的文章会引来大家的热烈讨论

^ | v · 回复 · 分享 ▾



wapeyang · 2个月前

确实是好文，文笔清晰，觉的技术类文章就应该这样写 抛出问题 引介历史背景 对比现在 分析 解决，，多谢作者

^ | v · 回复 · 分享 ▾

**bob bright** • 4个月前

首先感谢楼主分享精神。

楼主这种循序渐进，简单介绍环境，引出问题，分析问题，解决问题，之后从深度和背景探讨了该话题，让我肃然起敬。

^ | v • 回复 • 分享 >

**feihu** 管理员 ➔ bob bright • 4个月前

谢谢鼓励，能给大家带来帮助，这篇文章的目的也就达到了

^ | v • 回复 • 分享 >

**风海流** • 5个月前

感谢楼主分享，赞赏楼主在技术上保有追根溯源的精神！

^ | v • 回复 • 分享 >

**peng yan** • 1年前

在看c++Primer 模版那一块 正好遇到这个不太理解。看完博主的文章之后 明白了。感谢！

^ | v • 回复 • 分享 >

**name** • 1年前

先不说文章内容的质量如何，单单文章的文笔就值得我佩服！

^ | v • 回复 • 分享 >

**Monster** • 1年前

写的真心好啊，简单易懂！（为了评论，专门注册了一下）

^ | v • 回复 • 分享 >

**feihu** 管理员 ➔ Monster • 1年前

谢谢夸奖，请多支持:-)

^ | v • 回复 • 分享 >

**羽茶仙** • 1年前

感谢作者分享你的理解，写的相当易懂，确实好文！尤其是把一步步的思考和发展的追因溯源的讲了出来，让人有醍醐灌顶之感。一点小小

的建议，如果可以的话可否把中文名和其相对应的英文名一起对照着说出来。因为有些时候确实觉得中文的翻译不如英文来的准确。

^ | v · 回复 · 分享 ›



feihu 管理员 → 羽茶仙 · 1年前

恩，非常好的建议，以后的这种尽量做到中英文放在一起对照展示出来

^ | v · 回复 · 分享 ›



tao hou · 2年前

比cpp reference写得清楚多了。不过感觉“限定名”“name”这些词汇定义依然有点模糊，好像是指类型而非变量名称。

^ | v · 回复 · 分享 ›



Asetsuna · 2年前

探讨很深入的文章，赞。

^ | v · 回复 · 分享 ›



Ryan · 2年前

一直都是像记公式一样记typename怎么用，看了您写的这篇解释终于理解了，十分感谢

^ | v · 回复 · 分享 ›



Emer · 2年前

不错不错,楼主可以出书了~

^ | v · 回复 · 分享 ›



feihu 管理员 → Emer · 2年前

谢谢，这里仅仅是简单的技术分享，难登大大雅之堂^_^

^ | v · 回复 · 分享 ›



Emer → feihu · 2年前

我是先看了stackoverflow上的回答,再看你的,你写的需要的读者知识起点更低,不过stackoverflow提到了自动lookup,而且放在一个暂时的buffer里,实例化了才会定下类型,所以有的时候不会出问题.

1 ^ | v · 回复 · 分享 ›



feihu 管理员 → Emer · 2年前



能把你说的StackOverflow上的回答地址贴出来看看么?

^ | v · 回复 · 分享 ›



Emer → feihu · 2年前

<http://stackoverflow.com/qu...>

^ | v · 回复 · 分享 ›



feihu 管理员 → Emer · 2年前

已经加上此链接，感谢提供。你研究的很深入啊:-)

^ | v · 回复 · 分享 ›



feihu 管理员 → Emer · 2年前

果然还有很多东西没有了解，这个回答很详细，估计是为了减轻我们这些可怜的用户们的负担，VC和gcc都没有采用这种方式吧。想想，如果真的那样实现了，会变得更难理解了。不过为了保持完整，我把这个链接加到文章的附录部分

^ | v · 回复 · 分享 ›



Emer → feihu · 2年前

我就是一半路出家的初学者，我一直认为工具类的东西，够用就行了，碰到不懂得再学习，不然前看后忘。最近是在研究boost的graph包，里面老是typename，所以才看了下。

你的文笔很清晰，我觉得可以往这方面走啊，肯定很有优势。

^ | v · 回复 · 分享 ›



feihu 管理员 → Emer · 2年前

很巧，我也算是个半路出家的。同意你的观点，这种学习方式最有效，也最快。写这些东西因为总是受到看过的一些技术书籍的影响，比如侯捷的书，深入浅出，让一个即使是初学者也不至于太难懂，所以也希望自己写的东西让别人也有这种感受吧。一点点积累！

^ | v · 回复 · 分享 ›



Emer → feihu · 2年前

traits的你写了么？写好的话分享给我啊，这东西现在很流行

^ | v · 回复 · 分享 ›



feihu 管理员 → Emer • 2年前

是嘛？之前写完这一篇时是计划写的，不过后来想想有些太偏，所以便搁置了。和typename一样，这东西对一般C++使用者来说还是过于艰涩了些

^ | v • 回复 • 分享 ›

— | 旗



looyup • 2年前

确实厉害

^ | v • 回复 • 分享 ›

— | 旗



skyeyvapor • 3年前

Bravo!

^ | v • 回复 • 分享 ›

— | 旗



wulongict • 3年前

写得太好了，深入浅出！

^ | v • 回复 • 分享 ›

— | 旗



feihu 管理员 → wulongict • 3年前

谢谢:-)

^ | v • 回复 • 分享 ›

— | 旗



JellyThink • 3年前

Nice。非常谢谢楼主的分享，写的非常详细。

^ | v • 回复 • 分享 ›

— | 旗

✉ 订阅 在您的网站上使用 Disqus添加 Disqus添加 隐私



libfeihu is the home of feihu, a programmer from Shenzhen,
China.

douban | weibo

© 2017 libfeihu

Made with Jekyll — Theme by orderedlist