模板中的名字查找问题

问题起源

先看下面很简单的一小段程序。

```
#include <iostream>

template <typename T>
struct Base
{
   void fun()
   {
       std::cout << "Base::fun" << std::endl;
   }
};

template <typename T>
struct Derived : Base<T>
{
   void gun()
   {
       std::cout << "Derived::gun" << std::endl;
       fun();
   }
};
```

这段代码在 GCC 下很意外地编译不过，原因竟然是找不到 fun 的定义，可是明明就定义在基类中了好吗！为什么视而不见呢？显然这和编译器对名字的查找方式有关，那这里面究竟有什么玄机呢？上述代码是写得不规范，还是 GCC 竟然存在这样愚蠢而又莫名其妙的 bug?

C++ 标准的要求

对于模板中引用的符号，C++ 的标准有这样的要求：

1. 如果名字不依赖于模板中的模板参数，则该符号必须定义在当前模板可见的上下文内。
2. 如果名字是依赖于模板中的模板参数，则该符号是在实例化该模板时，才对该符号进行查找。

也就是说，对于前面提到的例子，gun() 函数中调用 fun()，由于该 fun() 并不依赖于 Derived 的模板参数T，因此在编译器看来该调用就相当于 ::fun()，直接把它当成是一个外部的符号去查找，而此时外部又没有定义该函数，因此就报错了。要去除这种错误，解决的方法很简单，只要在调用 fun 的地方，人为地加上该调用对模板参数的依赖则可。

```
template <typename T>
struct Derived : Base<T>
{
   void gun()
   {
       std::cout << "Derived::gun" << std::endl;
       this->fun();// or Base<T>::fun();
   }
};
```

加上 this 之后，fun 就依赖于当前 Derived 类，也就间接依赖了模板参数T，因此名字的查找就会被推迟到该类被实例化时才去基类中查找。

两阶段名字查找

从前面的介绍，我们可以看到编译器对模板中引用的符号的查找是分为两个阶段的：

1. 符号不依赖于当前模板参数，该符号则被当作是外部的符号，直接在当前模板所在的域中去查找。
2. 符号如依赖于当前模板参数，则对该符号的查找被推迟到模板被实例化时。

为什么要这样区别对待呢？原因其实很简单，编译器在看到模板 Derived 的定义时，还不能确定它的基类最后是怎样的：Base 有可能会在后面被特化，使得最后被继承的具体基类中不一定还有 fun() 函数。

```
template <>
struct Base<int>
{
   void fun2()
   {
       std::cout << "Specialized, Base::fun2" << std::endl;
   }
};
```

因此编译器在看到模板类的定义时，还不能判断它的基类最后会被实例化成怎样，所以对依赖于模板参数的符号的查找只能推迟到该模板被实例化时才进行，而如果符号不依赖于模板参数，显然没有这个限制，因此可以在看到模板的定义时就直接进行查找，于是就出现了对不同符号的两阶段查找。

符号与类型问题

对于前面介绍中提到的符号，我们其实默认指的是变量，细心的读者可能会想到，在继承类中引用的符号，还可能会是类型，而由于模板特化的存在，在名字查找的第一阶段编译器也是没法判断出该符号最后到底是怎样的类型，甚至不能知道是不是一个类型。

```
template <typename T>
struct Base
{
   typedef char* baseT;
};

template <typename T>
struct Derived : Base<T>
{
   void gun()
   {
      Base<T>::baseT p = "abc";
   }
};
```

```
template <>
struct Base<int>
{
   typedef int baseT;
};

template <>
struct Base<float>
{
   int baseT;
};
```

如上例子，Derived 中 gun() 函数对 Base::baseT 的引用会造成编译器的迷惑，它在看到 Derided 的定义时，根本无从知道 Base::baseT 究竟是一个变量名，还是一个类型，以及什么类型？而它又不能直接把这一部分相关的代码全部都推迟到第二阶段再进行，因此在这儿它就会报错了：它可以不知道这个类型最后是什么类型，但它必须知道它究竟是不是类型，如果连这个都不知道，接下来相关的代码它都没法去解析了。因此，实际上，编译器在看到一个与模板参数相关的符号时，默认它都是当作一个变量来处理的，所以在上述的例子中，编译器在看到 Derived 的定义时，它直接把 Base::baseT 当成了一个变量来处理，所以就会报错了。

那么，我们要怎样才能让编译器知道其实 Base::baseT 是一个类型呢？ 必须得显式地告诉它，因此需要在引用 Base::baseT 时，显式地加入一个关键字：typename.

```
template <typename T>
struct Derived : Base<T>
{
   void gun()
   {
     typename Base<T>::baseT p = "abc";
   }
};
```

此时，编译器看到有 typename 显式地指明 baseT 是一个类型，它就不会再把它默认当成是一个变量了，从而使得名字查找的第一个阶段可以继续下去。

总结

模板中名字的查找会因为该名字是否依赖于模板参数而有所不同。

依赖于模板参数的名字(如函数的参数的类型是模板的参数)，其符号解析会在第二阶段进行，其查找方式有两个：

1. 在模板定义的域内可见的符号。（很严格）
2. 在实例化模板的域内通过 ADL 的方式查找符号。（也很严格，杜绝了不同 namespace 内部重复定义导致冲突的问题）。

而不依赖于模板参数的符号，则只会在定义模板的可见域内进行查找，语言的定义严格如上所述，但实际编译器的支持上，msvc 不支持两阶段的查找（vc 2010 以前），gcc 的实现在 4.7 以前也不完全符合标准，一个比较全面的符合规范的例子，请参看如下：

```
void f(char); // 第一个 f 函数

template<class T>
void g(T t) {
    f(1);    // 不依赖参数的符号，符号解释在第一阶段进行，找到 ::f(char)
    f(T(1)); // 依赖参数的符号：查找推迟
    f(t);    // 依赖参数的符号：查找推迟
}

enum E { e };
void f(E);   // 第二个 f 函数
void f(int); // 第三个 f 函数

void h() {
    g(32); // 实例化 g<int>，此时进行查找 f(T(1)) 和 f(t)
           // f(t) 的查找找到 f(char)，因为是通过非 ADL 方式查找的(T 是 int，ADL 失效)，而定义模板的域内只有 f(char)。
           // 同理，f(T(1)) 的查找也只找到 f(char)。

    g(e); // 实例化 g<E>，此时进行查找 f(T(1)) 和 f(t)，因为参数都是用户定义的类型，ADL 起效，因此两者均找到了 f(E)，

}

typedef double A;
template<class T> class B {
   typedef int A;
};

template<class T> struct X : B<T> {
   A a; // 此处 A 为 double
};
```

【引用】

http://gcc.gnu.org/onlinedocs/gcc/Name-lookup.html
http://womble.decadent.org.uk/c++/template-faq.html
http://en.cppreference.com/w/cpp/language/unqualified_lookup
https://gcc.gnu.org/gcc-4.7/porting_to.html
http://en.cppreference.com/w/cpp/language/adl

分类: c/c++, Language

标签: c++, name lookup, 名字查找, template

twoon
关注 - 5
粉丝 - 181
+加关注

« 上一篇： 基于ε-NFA的正则表达式引擎
» 下一篇： c++11 内存模型解读

posted on 2014-06-15 16:29 twoon 阅读(1859) 评论(5) 编辑 收藏 举报