C++ template —— 模板中的名称（三）

第9章 模板中的名称

C++(与C一样)是一种上下文相关语言：对于C++的一个构造，我们不能脱离它的上下文来理解它。模板也是一种构造，它必须处理多种上下文相关信息：
（1）模板出现的上下文；
（2）模板实例化的上下文；
（3）用来实例化模板的模板实参的上下文。

9.1 名称的分类

主要的命名概念：

（1）如果一个名称使用域解析运算符（即::）或者成员访问运算符（即 . 或 ->）来显式表明它所属的作用域，我们就称该名称为受限名称。

（2）如果一个名称（以某种方式）依赖于模板参数，我们就称它为依赖型名称。

名称的分类详见表9.1

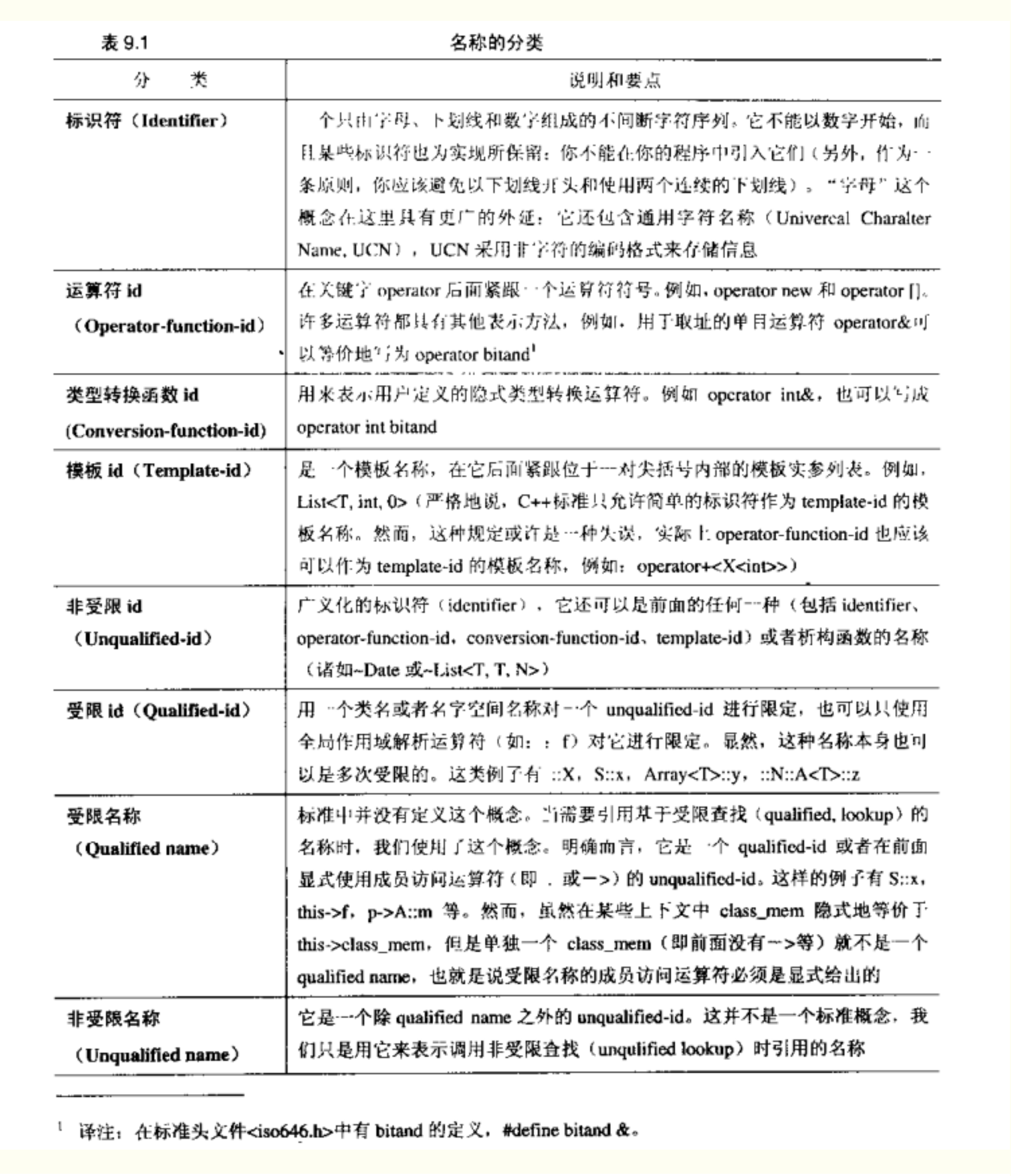
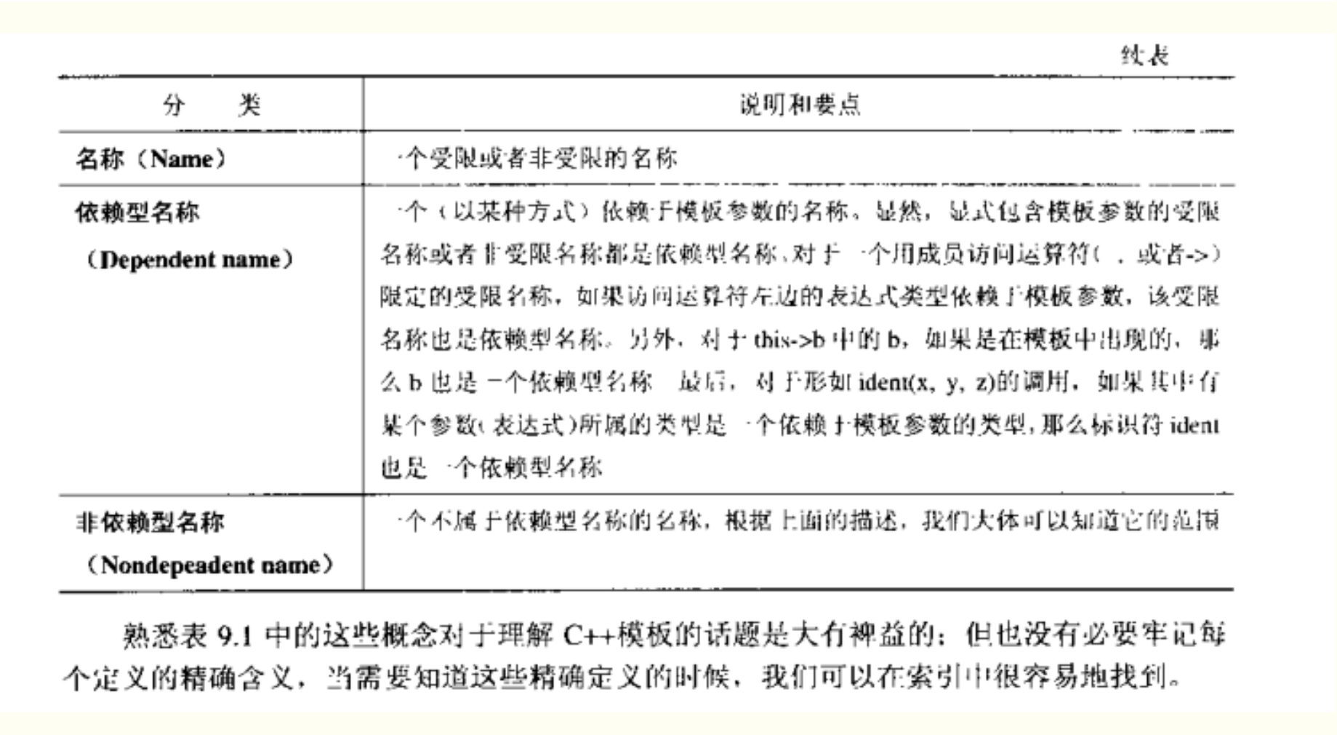


表 9.1 名称的分类

分类	说明和要点
标识符（Identifier）	一个只由字母、下划线和数字组成的不间断字符序列。它不能以数字开始，而且某些标识符也为实现所保留：你不能在你的程序中引入它们（另外，作为一条原则，你应该避免以下划线开头和使用两个连续的下划线）。"字母"这个概念在这里具有更广的外延：它还包含通用字符名称（Univercal Charalter Name, UCN），UCN 采用非字符的编码格式来存储信息
运算符 id（Operator-function-id）	在关键字 operator 后面紧跟一个运算符符号。例如，operator new 和 operator []。许多运算符都具有其他表示方法，例如，用于取址的单目运算符 operator&可以等价地写为 operator bitand[1]
类型转换函数 id（Conversion-function-id）	用来表示用户定义的隐式类型转换运算符。例如 operator int&，也可以写成 operator int bitand
模板 id（Template-id）	是一个模板名称，在它后面紧跟位于一对尖括号内部的模板实参列表。例如，List<T, int, 0>（严格地说，C++标准只允许简单的标识符作为 template-id 的模板名称。然而，这种规定或许是一种失误，实际上 operator-function-id 也应该可以作为 template-id 的模板名称，例如：operator+<X<int>>）
非受限 id（Unqualified-id）	广义化的标识符（identifier），它还可以是前面的任何一种（包括 identifier、operator-function-id、conversion-function-id、template-id）或者析构函数的名称（诸如~Date 或~List<T, T, N>）
受限 id（Qualified-id）	用一个类名或者名字空间名称对一个 unqualified-id 进行限定，也可以只使用全局作用域解析运算符（如：：f）对它进行限定。显然，这种名称本身也可以是多次受限的。这类例子有 ::X，S::x，Array<T>::y，::N::A<T>::z
受限名称（Qualified name）	标准中并没有定义这个概念。当需要引用基于受限查找（qualified, lookup）的名称时，我们使用了这个概念。明确而言，它是一个 qualified-id 或者在前面显式使用成员访问运算符（即 . 或—>）的 unqualified-id。这样的例子有 S::x，this->f，p->A::m 等。然而，虽然在某些上下文中 class_mem 隐式地等价于 this->class_mem，但是单独一个 class_mem（即前面没有—>等）就不是一个 qualified name，也就是说受限名称的成员访问运算符必须是显式给出的
非受限名称（Unqualified name）	它是一个除 qualified name 之外的 unqualified-id。这并不是一个标准概念，我们只是用它来表示调用非受限查找（unqulified lookup）时引用的名称

---

[1] 译注：在标准头文件<iso646.h>中有 bitand 的定义，#define bitand &。



续表

分类	说明和要点
名称（Name）	一个受限或者非受限的名称
依赖型名称（Dependent name）	一个（以某种方式）依赖于模板参数的名称。显然，显式包含模板参数的受限名称或者非受限名称都是依赖型名称。对于一个用成员访问运算符（ . 或者->）限定的受限名称，如果访问运算符左边的表达式类型依赖于模板参数，该受限名称也是依赖型名称。另外，对于 this->b 中的 b，如果是在模板中出现的，那么 b 也是一个依赖型名称。最后，对于形如 ident(x, y, z)的调用，如果其中有某个参数(表达式)所属的类型是一个依赖于模板参数的类型，那么标识符 ident 也是一个依赖型名称
非依赖型名称（Nondepeadent name）	一个不属于依赖型名称的名称，根据上面的描述，我们大体可以知道它的范围

熟悉表 9.1 中的这些概念对于理解 C++模板的话题是大有裨益的；但也没有必要牢记每个定义的精确含义，当需要知道这些精确定义的时候，我们可以在索引中很容易地找到。

9.2 名称查找

这里是讨论一些主要概念.

1. 受限名称的名称查找是在一个受限作用域内部进行的，该受限作用域由一个限定的构造所决定。如果该作用域是一个类，那么查找范围可以到达它的基类；但不会考虑它的外围作用域。如下例子：

```
int x;

class B
{
    public:
        int i;
};

class D : public B
{
};

void f(D* pd)
{
    pd->i = 3;  // 找到B::i
    D::x = 2;   // 错误：并不能找到外围作用域中的::x
}
```

名称:
1. 受限不受限: 在自己类中找 vs 从内到外逐层查找

* 非受限特例: 非限在子类类的成员函数中被定义]:
如果非受限名称出现在一个[类的成员函数, 而非成员数据]中被定义, 显然, 意思是说, 这个非限名称, 很可能不是本类的, 要首先在基类中查找这个名称.

2. 依赖不依赖:
(1) 非依赖名称一碰到(定义时)就查询, 而不延迟到模板实例化时;
(2) 非依赖名称(不含模板参数T的名称) 不会去 依赖性基类(含模板参数T的基类base<T>)中找

3. 非受限 + 非依赖基类:
* 非受限特例: 非限名称的父类是独立/非依赖型的 * :
如果非限名称出现在子类中, 无论是否出现在子类的成员函数还是子类的成员数据中, 只要基类是非依赖型的, 则先 在非依赖型基类中找, 再去子类和其他类中找.

2. 非受限名称的查找则相反，它可以（由内到外）在所有外围类中逐层地进行查找（但在某个类内部定义的成员函数定义中，它会先查找该类和基类的作用域，然后才查找外围类的作用域）。这种查找方式也被称为普通查找。如下：

3. 对于非受限名称的查找，最近增加了一项新的查找机制——除了前面的普通查找——就是说非受限名称有时可以使用依赖于参数的查找（argument-dependent lookup,ADL）。在阐述ADL的细节之前，让我们先通过max()模板来说明这种机制的动机：

```
template <typename T>
inline T const& max(T const& a, T const& b)
{
    return a < b ? b : a;
}
```

假设我们现在要让"在另一个名字空间中定义的类型"使用这个模板函数：

```
namespace BigMath{
    class BigNumber
    {
        ...
    };
    bool operator < (BigNumber const&, BigNumber const&);
    ...
}
using BigMath::BigNumber;

void g(BigNumber const& a, BigNumber const& b)
{
    ...
    BigNumber x = max(a, b);
    ...
}
```

问题是max()模板并不知道BigMath名字空间，因此普通查找也找不到"应用于BigNumber类型值的operator<"。ADL正是解决这种限制的特殊规则。

9.2.1 Argument-Dependent Lookup(ADL)

ADL只能应用于非受限名称。在函数调用中，这些名称看起来像是非成员函数。对于成员函数名称或者类型名称，如果普通查找能找到该名称，那么将不会应用ADL。如果把被调用函数的名称（如max）用圆括号括起来，也不会使用ADL。

否则，如果名称后面的括号里面有（一个或多个）实参表达式，那么ADL将会查找这些实参的associated class（关联类）和associated namespace（关联名字空间）。

对于给定类型，对于由associated class（关联类）和associated namespace（关联名字空间）所组成的集合的准确定义，可以通过下列规则来确定：
（1）对于基本类型，该集合为空集。
（2）对于指针和数组类型，该集合是所引用类型（譬如对于指针而言，它所引用的类型是"指针所指对象"的类型）的associated class和associated namespace。
（3）对于枚举，associated namespace指的是枚举声明所在的namespace。对于类成员，associated class指的是它所在的类。
（4）对于class类型（包含联合类型），associated class集合包括：该class类型本身、它的外围类型、直接基类和间接基类。associated namespace集合是每个associated class所在的namespace。如果这个类是一个类模板实例化体，那么还包含：模板类型实参本身的类型、声明模板的模板实参所在的class和namespace。
（5）对于函数类型，该集合包括所有参数类型和返回类型的associated class和associated namespace。
（6）对于类X的成员指针类型，除了包括成员相关的associated namespace和associated class，该集合还包括与X相关的associated namespace和associated class。

至此，ADL会在所有的associated class和associated namespace中依次地查找，就好像依次地直接使用这些名字空间进行限定一样。唯一的例外情况是：它会忽略using-directives（using指示符）。

9.2.2 友元名称插入 考虑下面代码：

```
template <typename T>
class C
{
    ...
    friend void f();
    friend void f(C<T> const&);
    ....
};

void g(C<int>* p)
{
    f();      // f()在此是可见的吗？不可见，不能利用ADL，因此是一个无效调用
    f(*p);        // f(C<int> const&)在此是可见的吗？可见，因为友元函数所在的类属于ADL的关联类
集合
}
```

这里的问题是:如果友元声明在外围类中是可见的，那么实例化一个类模板可能会使一些普通函数（例如f()）的声明也成为可见的。一些程序员会认为这样很出乎意料。因此C++标准规定：通常而言，友元声明在外围（类）作用域中是不可见的。

但同时，C++标准还规定：如果友元函数所在的类属于ADL的关联类集合，那么我们在这个外围类是可以找到该友元声明的。

9.2.3 插入式类名称

如果在类本身的作用域中插入该类的名称，我们就称该名称为插入式类名称。它可以被看作位于该类作用域中的一个非受限名称，而且是可访问的名称。

类模板也可以具有插入式类名称。然而，它们和普通插入式类名称有些区别：它们的后面可以紧跟模板实参（在这种情况下，它们也被称为插入式类模板名称）。但是，如果后面没有紧跟模板实参，那么它们代表的就是用参数来代表实参的类（例如，对于局部特化，还可以用特化实参代表对应的模板实参）。这同时说明了下面的情况：

```
template <template<typename> class TT> class X{ };

template <typename T> class C
{
    C* a;        // 正确：等价于C<T>* a
    C<void> b;      // 正确
    X<C> c;       // 错误：后面没有模板实参列表的非受限名称C不被看作模板
    X<::C> d;    // 错误：<: 是 [ 的另一种标记（表示）
    X< ::C> e;       // 正确：在 < 和 ::之间的空格是必需的
};
```

从上面代码我们可以知道如何使用非受限名称来引入插入式名称（即C），如果这些非受限名称的后面没有紧跟模板实参列表，那么是不会被看成模板名称的。

9.3 解析模板

大多数程序设计语言的编译都包含两个最基本的步骤：符号标记——和解析。扫描过程把源代码当作字符串序列读入，然后根据该序列生成一系列标记。接下来，解析器会递归地减少标记，或者把前面已经找到的模式结合成更高层次的构造，从而在标记序列中不断对应已知模式。

9.3.1 非模板中的上下文相关性
C++编译器会使用一张符号表把扫描器和解析器结合起来，解决上下文相关性的问题。当解析某个声明的时候，该声明就会添加到表中。当扫描器找到一个标识符时，它会在符合表中进行查找，如果发现该标识符是一个类型，就会注释这个所获得的标记（标识符）。

9.3.2 依赖型类型名称
有关模板名称的问题主要是：这些名称不能有效地确定。尤其是模板中不能引用其他模板的名称，因为其他模板的内容可能会由于显式特化而使原来的名称失效。

C++的语言定义通过下面规定来解决这个问题：通常而言，依赖型受限名称并不会代表一个类型，除非在该名称的前面有关键字typename前缀。总之，当类型名称具有以下性质时，就应该在该名称前面添加typename前缀：
（1）名称出现在一个模板中；
（2）名称是受限的；
（3）名称不是用于指定基类继承的列表中，也不是位于引入构造函数的成员初始化列表中；
（4）名称依赖于模板参数。

而且，只有当前面3个条件满足的情况下，才能使用typename前缀。如下例子：

```
template <typename1 T>
struct S : typename2 X<T>::Base
{
    S() : typename3 X<T>::Base(typename4 X<T>::Base(0) ) {}
    typename5 X<T> f()
    {
        typename6 X<T>::C *p;      // 指针p的声明
        X<T>::D* q;    // 乘积
    }
    typename7 X<int>::C *s;
};

struct U
{
    typename8 X<int>::C *pc;
};
```

注：
typename1引入模板参数，因此不适用前面的规则；
typename2和typename3属于规则（3）所禁止的用法；
typename4必不可少；
typename5属于规则（2）所禁止的用法；
typename6如果是期望声明一个指针，那么这个typename就是必需的；
typename7是可选的，因为它符合前面的3条规则，但不符合第4条规则；
typename8是禁止的，因为它并不是在模板中使用。

9.3.3 依赖型模板名称
如果一个模板名称是依赖型名称，我们将会遇到与上一小节类似的问题。通常而言，C++编译器会把模板名称后面的<看作模板参数列表的开始；但如果该<不是位于模板名称后面，那么编译器将会把它当作小于号处理。和类型名称一样，要让编译器知道所引用的依赖型名称是一个模板，需要在该名称前面插入template关键字，否则的话编译器将假定它不是一个模板名称：

```
template <typename T>
class Shell
{
    public:
        template<int N>
        class In
        {
            public:
                template<int M>
                class Deep
                {
                    public:
                        virtual void f();
                };
        };
};

template<typename T, int N>
class Weird
{
    public:
        void case1(typename Shell<T>::template In<N>::template Deep<N>* p){
            p->template Deep<N>::f();      // 禁止虚函数调用（具体原因后面针对限定符部分讲解）
        }
        void case2(typename Shell<T>::template In<N>::template Deep<N>* p){
            p.template Deep<N>::f();      // 禁止虚函数调用
        }
};
```

这个多少有些复杂的例子给出了何时需要在运算符（::, ->和 . ，用于限定一个名称）的后面使用关键字template。更明确的说法是：如果限定符号前面的名称（或者表达式）的类型要依赖于某个模板参数，并且紧接在限定符后面的是一个template-id（就是指一个后面带有尖括号内部实参列表的模板名称），那么就应该使用关键字template。例如，在下面的表达式中：

```
p.template Deep<N>::f()
```

p的类型要依赖于模板参数T。然而，C++编译器并不会查找Deep来判断它是否是一个模板；因此我们必须显式指定Deep是一个模板名称，这可以通过插入template前缀来实现。如果没有这个前缀的话，p.Deep<N>::f()将会被解析为((p.Deep) < N) > f()，这显然并不是我们所期望的。我们还应该看到：在一个受限名称内部，可能需要多次使用关键字template，因为限定符本身可能还会受限于外部的依赖型限定符（我们可以从前面例子中case1和case2的参数中看到这一点）。

9.3.4 using-declaration 中的依赖型名称

using-declaration 会从两个位置（即类和名字空间）引入名称。如果引入的是名字空间，将不会涉及到上下文问题，因为并不存在名字空间模板。实际上，从类中引入名称的using-declaration 的能力是有限的：只能把基类中的名称引入到派生类中。如下：

```
class BX
{
    public:
        void f(int);
        void f(char const*);
        void g();
};

class DX : private BX
{
    public:
        using BX::f;
};
```

私有继承中，通过using-declaration 访问基类的成员，但是这违背了C++早期的访问级别声明机制，所以可能以后不会包含这个机制。

现在，当using-declaration是从依赖型类（模板）中引入名称的时候，我们虽然知道这个引入的名称，但并不知道该名称究竟是一个类型名称、模板名称、还是一个其他的名称：

```
template <typename T>
class BXT
{
    public:
        typedef T Mystery;
        template <typename U>
        struct Magic;
};

template <typename T>
class DXTT : private BXT<T>
{
    public:
        using typename BXT<T>::Mystery;
        Mystery* p;      // 如果上面不使用typename，将会是一个语法错误
};
```

而且，如果我们期望使用using-declaration 所引入的依赖型名称是一个类型，我们必须插入关键字typename来显式指定。另一方面，比较奇怪的是，C++标准并没有提供一种类似的机制，来指定依赖型名称是一个模板。如下：

```
template <typename T>
class DXTM : private BXT<T>
{
    public:
        using BXT<T>::template Magic;  // 错误：非标准的
        Magic<T>* plink; //语法错误：Magic并不是一个已知模板
};
```

这应该是标准规范的一个疏忽。

9.3.5 ADL和显式模板实参
考虑下面例子：

```
namespace N{
    class X
    {
        ...
    };
    template<int I> void select(X*);
}

void g(N::X* xp)
{
    select<3>(xp);     // 错误：没有ADL
}
```

在这个例子中，调用select<3>(xp)的时候，我们可能会期望通过ADL来找到模板select()；然而，实际情况并不是这样的。因为编译器在不知道<3>是一个模板实参列表之前，是无法断定xp是一个函数调用实参的；反过来，如果要判断<3>是一个模板实参列表，我们需要先知道select()是一个模板。这种是先有鸡还是先有蛋的问题没法解决，因此编译器只能把上面表达式解析成(select<3)>(xp)，但这并不是我们所期望的，也是毫无意义的。

9.4 派生和类模板

类模板可以继承也可以被继承。

9.4.1 非依赖型基类

在一个类模板中，一个非依赖型基类是指：无需知道模板实参就可以完全确定类型的基类。就是说，基类名称使用非依赖型名称来表示的。如下：

```
template<typename X>
class Base
{
    public:
        int basefield;
        typedef int T;
};
class D1 : public Base<Base<void> >   // 实际上不是模板
{
    public:
        void f() { basefield = 3; }
};

template<typename T>
class D2 : public Base<double>       // 非依赖型基类
{
    public:
        void f() {basefield = 7; }   // 正常访问继承成员
        T strange;      // T是Base<double>::T，而不是模板参数
};
```

模板出现前, 普通类都是不含模板参数的独立类, 类中的名称也多是非限名称.
所以, 独立基类+非限名称≈普通类+成员数据名或函数名

非依赖性基类, 不含模板参数的基类, 自己是个啥自个儿就能定下来的基类,
比如这里的基类 Base<double>.
对于这个自立的基类, 如果其子类中也有很独立的非受限名称, 则先会在"这个独立的基类"中找这个"非受限的"名称.

受限/非限名称, 独立/依赖名称 中的"名称":
变量名? 函数名? 类型名? 任何名?

(1) 类型名: 非限的T是独立基类的Base<double>::T, 是 int 的别名, 不是子类模板的模板参数T
Base不能说独立不独立, 甚至不是个类, 独不独立是说Base<double>或Base<T>.

(2) 变量名: 非限的strange 现在独立基类 Base<double>中没找到, 然后在子类中找[反正非受限名称的搜索范围大嘛], 嗯, 就是T后面的这个strange, 子类新定义的整型变量.

(3) 变量名: basefield, 非限名称, 先去独立基类中找找看, 嗯, 找到了, 基类定义它是一个整型变量.

(4) 函数名: 出现在子类中的非限名称 f, 好像是子类定义了个函数, 但应该 先在基类中找, 没找到. 那就是刚刚定义的一个函数.

模板中的非依赖型基类的性质和普通非模板类中的基类的性质很相似，但存在一个很细微的区别：对于模板中的非依赖型基类而言，如果在它的派生类中查找一个非受限名称，那就会先查找这个非依赖型基类，然后才查找模板参数列表。这就意味着：在前面的例子中，类模板D2的成员strange的类型一直都会是Base<double>::T中对应的T类型（个人理解：因为首先查找了非依赖型基类Base<double>，所以得到的T的类型就一直是Base<double>::T的类型。如果是普通非模板类的话，那么会首先在派生类自己中查找，也即，可以找到如D2<int>::T的类型。还是不太理解，待求证？？）。例如，下面的函数是无效的C++代码：

```
void g(D2<int*>& d2, int* p)
{
    d2.strange = p;   // 错误，类型不匹配
}
```

这一违背直观查找的特性是我们要格外注意的。

9.4.2 依赖型基类

在前面的例子中，基类是完全确定的，它并不依赖于模板参数。这就意味着：一看到模板的定义，C++编译器就可以在这些基类中查找非依赖型名称。而另一种候选方法（C++标准并不允许这种方法）会延迟这类名称的查找，只有等到进行模板实例化时，才真正查找这类名称。这种候选方法的缺点是：它同时也将诸如漏写某个符号导致的错误信息，延迟到实例化的时候产生。因此，C++标准规定：对于模板中的非依赖型名称，将会在看到的第一时间进行查找。有了这个概念之后，让我们考虑下面的例子：

第一时间: 定义中碰到了就开始查找, 不等到模板实例化时

```
template <typename T>
class DD : public Base<T>
{
    public:
        void f() { basefield = 0; }     // (1)problem……
};

template<>    // 显式特化
class Base<bool>
{
    public:
        enum { basefield = 42 };     // (2)tricky
};

void g(DD<bool>& d)
{
    d.f();       // (3)oops ?
}
```

查找的主要是指: 非限名称 或 非依赖性名称, 因为没有限制域嘛, 算谁的呢? 因为不含模板参数, 不知道和谁一起参数化.

不含模板参数的名称, 即, 我本身就没有T, 为啥要到含T的基类中找呢?

子类DD对象, d, 的f: d.f()
f是非依赖的, 不含T的, 所以不会去含T的基类中找, 即, 绝对不是 基类<T>::f.

不是非依赖性, 而是有依赖的, 所以先找子类DD<bool>

在（1）处我们发现代码中引用了非依赖型名称basefield，必须马上对它进行查找。假设我们在模板Base中查找到它，并根据Base类的声明把basefield绑定为int变量。然而，我们随后使用显式特化改写了Base的泛型定义，在特化中改变了成员basefiel的含义，而（1）处basefield的含义在这之前已确定下来了（即绑定为一个int变量）；这也是错误的根源。因此，当我们在（3）处实例化DD::f的定义时，我们会发现过早地在（1）处绑定了非类型名称；然而根据（2）处对DD<bool>的特殊指定，basefield应该是一个不可修改的常量，因此编译器在（3）处将会给出一个错误的信息。
为了（巧妙地）解决这个问题，标准C++声明：非依赖型名称不会在依赖型基类中进行查找（但仍然是在看到的时候马上进行查找）。因此，标准的C++编译器将会在（1）处给出一个诊断信息。为了纠正这里的代码，我们可以让basefield也成为依赖型名称，因为依赖型名称只有在实例化时才会进行查找；而且在实例化时，基类的特化是已知的。例如，在（3）处，编译器知道DD<bool>的基类是Base<bool>,而且Base<bool>是程序员进行显式特化的。在这个例子中，我们可以借助如下的修改方案是basefield成为一个依赖型名称：

```
// 修改方案1
template <typename T>
class DD1 : public Base<T>
{
    public:
        void f() { this->basefield = 0; }     // 查找被延迟了
};

// 修改方案2：利用受限名称来引入依赖性
template <typename T>
class DD2 : public Base<T>
{
    public:
        void f() { Base<T>::basefield = 0; }
};
```

如果是使用这个解决方法，我们需要格外小心，因为如果（原来的）非受限的非依赖型名称是被用于虚函数调用的话，那么这种引入依赖性的限定将会禁止虚函数调用，从而也会改变程序的含义(详见下一篇)。因此，当遇到第2种解决方案不适用的情况，我们可以使用方案1。

最后提供第3个修改方案如下：

```
// 修改方案3：重复的限定让代码不雅观，可以在派生类中只引入依赖型基类
template <typename T>
class DD3 : public Base<T>
{
    public:
        using Base<T>::basefield;       // (1)依赖型名称现在位于作用域
        void f() { basefield = 0; }     // 正确
};
```

更多有关模板名称查找和ADL机制的内容参见本系列下一篇博文xxxx。

分类: C++ Template

好文要顶 关注我 收藏该文

小天_y
关注 - 63
粉丝 - 96
+加关注

0 推荐 0 反对

« 上一篇： 关于烂代码的那些事（下）
» 下一篇： C++ template —— 实例化和模板实参演绎（四）

posted @ 2016-01-22 10:29 小天_y 阅读(2193) 评论(0) 编辑 收藏 举报