

C++ template —— 实例化和模板实参演绎（四）

本篇讲解实例化和模板实参演绎

第10章 实例化

模板实例化是一个过程，它根据泛型的模板定义，生成（具体的）类型或者函数。本篇阐述如何组织源代码，以正确地使用模板。

当C++On-Demand实例化（隐式实例化或自动实例化）时，C++编译器遇到模板特化的使用例，它会自动所给的实参替换对应的模板参数，从而产生该模板的特化。这个过程是编译器自动进行的，并不需要客户端代码来引导（或者不需要模板定义来引导）。on-demand实例化表明：在使用模板（特化）的地方，编译器通常需要访问模板和某些模板成员的定义（也就是说，只有声明是不够的）。如下：

```
template <typename T> class C;           // (1) 这里只有声明

C<int>* p = 0;                          // (2) 正确：并不需要C<int>的定义

template<typename T>
class C
{
public:
    void f();                            // (3) 成员声明
};                                       // (4) 类模板定义结束

void g(C<int>& c)                        // (5) 只使用类模板声明
{
    c.f();                              // (6) 使用了类模板的定义，需要C::f() 的定义
};
```

下面是另一个需要进行（前面的）类模板实例化的表达式，因为编译器需要知道C<void>的大小：

```
C<void>* p = new C<void>;
```

在源码中，有时可能需要访问类模板成员，但在源码中这种需求并不总是显式可见的。例如，C++的重载解析规则会要求：如果候选函数的参数是class类型，那么该类型所对应的类就必须是可见的：

```
template <typename T>
class C
{
public:
    C(int);                             // 具有单参构造函数，可以被用于隐式类型转换
};

void candidate(C<double> const&);        // (1)
void candidate(int) {}                  // (2)

int main()
{
    candidate(42);                      // 前面两个函数声明都可以被调用
}
```

调用candidate(42)将会采用（2）处的重载声明。然而，编译器仍然可以实例化（1）处的声明，来检查产生的实例化能否成为该调用的一个有效候选函数。

10.2 延迟实例化

- 模板实例化只对确实需要的部分进行实例化。换句话说，编译器会延迟模板的实例化。
- 当隐式实例化类模板时，同时也实例化了该模板的每个成员声明，但并没有实例化相应的定义。
- 然而，存在一些例外的情况：首先，如果类模板包含了一个匿名的union，那么该union定义的成员声明也会被实例化了（匿名的union有它自身的特殊之处：它的成员可以被看成是外围类的成员。匿名成员可以看作是一种构造，用来说明某些类成员自身同一个存储器）。
- 另一种例外情况发生在虚函数身上：作为实例化类模板的结果，虚函数的定义可能被实例化了，但也可能还没有被实例化，这要依赖于具体的实现。实际上，许多实现都会实例化（虚函数）的定义，因为“实现虚函数调用机制的内部结构”要求虚函数（的定义）作为链接实体存在。
- 当实例化模板的时候，缺省的函数调用实参是分开考虑的。准确而言，只有这个被调用的函数（或成员函数）确实使用了缺省实参，才会实例化该实参。就是说，如果这个函数（或成员函数）不使用缺省调用实参，而是使用显式实参来进行调用，那么就不会实例化缺省实参。

10.3 C++的实例化模型

10.3.1 两阶段查找

上一篇中我们知道：当对模板进行解析的时候，编译器并不能解析依赖型名称。于是，编译器会在POI(point of instantiation,实例化点)再次查找这些依赖型名称。另一方面，非依赖型名称是在首次看到模板的时候就被进行查找，因此在第1次查找时就可以诊断错误信息，于是，就有了两阶段查找这个概念：第1阶段发生在模板的解析阶段，第2阶段发生在模板的实例化阶段。

在第1阶段，当使用普通查找规则（在适当的情况下也会使用ADL）对模板进行解析时，就会查找非依赖型名称。另外，非受限的依赖型名称（诸如函数调用中的函数名称，所以说它是依赖型的，是因为该名称具有一个依赖型实参）也会在这个阶段进行查找，但它的查找结果是不完整的（就是说查找还没结束），在实例化模板的时候，还会再次进行查找。

第2阶段发生在模板被实例化的时候，我们也将此时发生的地点（或者源代码的某个位置）为一个实例化点POI。依赖型受限名称就是在此阶段进行查找的（查找的目标是：运用模板实参代替参数之后所获得的特定实例化体）；另外，非受限的依赖型名称在此阶段也会再次执行ADL查找。

10.3.2 POI

1. 非类型POI

（非类型POI、非类型实例、非类型实体、非类型特化.....这里“非类型”可以理解为（个人理解）不是类的类型。）

从上面我们知道，C++编译器会在模板客户端代码中的某些位置访问模板实体的声明或者定义。于是，当某些代码特化实例化模板特化，而且为了生成这个完整的特化，需要实例化相应模板的定义时，就会在源代码中产生一个实例化点（POI）。我们应该清楚，POI是位于源代码中的一个点，在该点会插入替换后的模板实例。例如：

```
class MyInt
{
public:
    MyInt(int i);
};

MyInt operator - (MyInt const&);

bool operator > (MyInt const& a, MyInt const& b);
typedef MyInt Int;
template <typename T>
void f(T i)
{
    if(i > 0)
    {
        g(-i);
    }
}
// (1)
void g(Int)
{
    // (2)
    f(Int>(42)); // 调用点
    // (3)
}
// (4)
```

当C++编译器看到调用f<Int>(42)时，它知道需要用MyInt替换T来实例化模板f:即生成一个POI。（2）处和（3）处是临近调用点的两个地方，但它们不能作为POI，因为C++并不允许我们把::f<Int>(Int)的定义在这里插入。另外，（1）处和（4）处的本质区别在于：在（4）处，函数g(Int)是可见的，而（1）处则不是；因此在（4）处函数g(-i)可以被解析。然而，如果我们假定（1）处作为POI，那么调用g(-i)将不能被解析，因为g(Int)在（1）处是不可见的。幸运的是，对于指向非类型特化的引用，C++把它的POI定义在“包含这个引用的定义或声明之后的最近名字空间域”中。在我们的例子中，这个位置是（4）。你可能会疑惑我们为什么在例子中使用类型MyInt，而不直接使用简单的int类型。这主要是因为：在POI执行的第2次查找（指g(-i)）只是使用了ADL，而基本类型int并没有关联名字空间，因此，如果使用了int类型，就不会发生ADL查找，也就不能找到函数g。

- 对于类特化，这个（POI）位置是不一样的。如下：

```
template <typename T>
class S
{
public:
    T m;
};
// (5)

unsigned long h()
{
    // (6)
    return (unsigned long)sizeof(S<int>);
    // (7)
}
// (8)
```

如前所述，我们知道位置（6）和（7）不能作为POI，因为名字空间域类S<int>的定义不能出现在这两个位置（模板是不能出现在函数作用域域内部的）。如果无效，因为我们采用前面非类型实例的规则，那么POI应该在（8）处。但这样的做法，表达式sizeof(S<int>)会是无效的，因为要等到在编译到（8）之后，我们才能确定S<int>的大小，而代表sizeof(S<int>)位于（8）之前。因此，对于指向产生自模板的类实例的引用，它的POI只能定义在“包含这个实例引用的定义或声明之前的最近名字空间域”中。在我们的这个例子中，是指位置（5）。

注：书中还介绍了某些附带的实例化以及二次POI，详见书籍；

同一篇中我们通常会包含同个实例的多个POI，对于类模板实例而言，在每个翻译单元中，只有首个POI会被保留，而其他的POI则被忽略。对于非类型实例而言，所有的POI都会被保留。然而，对于上面的任何一种情况，ODR原则都会要求：对保留的任何个POI处所出现的同种实例化体，都必须是等价的。

10.3.3 包含模型与分离模型

当遇到POI的时候，（编译器要求）相应模板的定义必须是（基于某种方式）可见的。对于类特化而言，这就意味着：在同一个翻译单元中，类模板的定义必须要在它的POI之前就已经是可见的。对于非类型的POI而言，也可能采取上述定义，来确认和解释其他的表达式和声明。就这一点而言，在多个翻译单元中包含同个类定义的多个实例化体，和存在另一种实现方法：使用export关键字来声明非类型模板，而在另一个翻译单元中定义该非类型模板，这就是我们前面所谈到的分离模型。

10.3.4 跨翻译单元查找

模板中的名称分两阶段查找：

第1阶段发生在解析模板（也就是说，C++编译器第1次看到模板定义）的时候。在这个过程中，会使用普通查找规则和ADL规则对非依赖型名称进行查找。另外，非受限的依赖型函数名称（这里的依赖型是指函数的实参是依赖型的）会使用普通查找规则进行查找，但只是把查找结果保存起来，并不会试图进行重载解析过程——这是在第2阶段的查找完成之后才进行的。

第2阶段发生在产生POI（实例化点）的时候。在这一点上，会使用普通查找规则和ADL规则来查找依赖型受限名称。而依赖型非受限名称（它已经在第1阶段使用普通查找规则查找了一次）则只（注意，只）使用ADL规则进行查找，然后把ADL的查找结果结合第1阶段普通查找所获得的结果，组成一个候选函数集合，然后借助于重载解析，从该集合中选出（最佳的）被调用的函数。

下面用一个例子来解析所描述的一些概念：

- 关于包含模型的简单例子：

```
template <typename T>
void f1(T x)
{
    g1(x); // (1)
}

void g1(int)
{
}

int main()
{
    f1(7); // 错误，找不到g1
           // (2):f<int>(int)的POI
}
```

调用f1(7)将会产生f1<int>(int)的一个POI，它紧跟main()函数的后面（即（2）处）。在这个实例中，关键的问题是函数g1的查找。当第1次看到模板f1的定义时，编译器注意到非受限名称g1是一个依赖型名称，因为它的参数名称依赖于外部函数f的模板参数（即实参x的类型依赖于模板参数T）。因此，编译器会在（1）处使用普通查找规则来查找g1，然而在（1）处并不能看到g1（g1的定义在f1之后），从而第1阶段找不到g1。在（2）处，即f1的POI，会在关联名字空间和关联类中再次查找g1，但由于g1的唯一实参类型是int，而int并没有关联名字空间的关联类，从而第2阶段也找不到g1。因此，尽管在f1的POI处（即（2）处）可以使用普通查找规则找到g1（这只是一个假象而已），但是根据我们前面的分析，该例子实际上并不能找到g1。

书中提供了另一个例子，说明了：分离模型如何导致跨翻译单元的重载二义性问题。详见书籍。

10.4 几种实现方案

书中介绍了几种主流的C++（编译器）实现对包含模型的一些支持方法。

注：

当在多个翻译单元中使用类模板特化的时候，编译器会在每个（应用该类模板特化的）翻译单元都重复类模板的实例化过程。这通常都不会产生问题，因为类定义并不会直接生产低层次的代码；C++实现也只是在内部使用这些类定义，来确认和解释其他的表达式和声明。就这一点而言，在多个翻译单元中包含同个类定义的多个实例化体，和在多个翻译单元中多次包含同个类定义（通常是借助包含头文件来实现），两者之间并没有本质上的区别。

然而，如果你实例化的是一个（非内联）函数模板，而不是一个类模板，上面的情况就不同了。如果提供了普通非内联函数的多个定义，那么你将会违反ODR原则（一处定义原则）。

10.5 显式实例化（尽量不用）

为模板特化显式地生成POI是可行的，我们获得获得这种特化的构造称为显式实例化指示符。从语法上讲，它由关键字template和后面的特化声明组成，所声明的特化就是即将由实例化获得的特化。例如：

```
template <typename T>
void f(T) throw(T)
{
    // 4个有效的显式实例化体
    template void f<int>(int) throw(int);
    template void f<float> throw(float);
    template void f<long> throw(long);
    template void f<char>;
}
```

类模板的成员也可以利用这种方式来进行显式实例化：

```
template<typename T>
class S
{
public:
    void f(){}
};

template void S<int>::f();
template class S<void>;
```

C++标准规定，在同一个程序中，每个特定的模板特化最多只能存在一处显式实例化。而且，如果某个模板特化已经被显式实例化了，那么就不能对它进行显式特化，反之亦然。（唯一的显式实例化或显式特化）

第11章 模板实参演绎

11.1 演绎的过程

- 针对一个函数调用，演绎过程会“比较”调用实参的类型“和”函数模板对应的参数化类型（即T），然后针对要被演绎的一个或多个参数，分别推导出正确的替换。我们应该记住：每个“实参-参数对”的分析都是独立的；因此，如果最后所得出的结论发生矛盾，那么演绎过程将失败。
- 即使所有被演绎的模板参数都可以一致性地确定（即不发生矛盾），演绎过程也可能失败。这种情况是：在函数声明中，进行替换的模板实参可能会导致无效的构造。如下：

```
template <typename T>
typename T::Element* at(T const& a, int i)
{
    return a[i];
}

void f(int* p)
{
    int x = at(p, 7);
}
```

在此，T被演绎成int*（只有一个参数类型与T有关，当然也就不会发生矛盾），然而，在返回类型T::ElementT中，用int*来替换T之后，显然会导致一个无效的C++构造，从而也是这个演绎过程失败。

- 接下来描述实参-参数对是如何进行匹配的。我们使用下面的概念来进行描述：匹配类型A（来自实参的类型）和参数化类型P（来自参数的声明）。如果被声明的参数是一个引用声明（即T&），那么P就是所引用的类型（即T），而A仍然是实参的类型。否则的话，P就是所声明的参数类型，而A则是实参的类型；如果这个实参的类型是数组或者函数类型，那么还会发生decay转型，转化为对应的匹配诸如int(&)[sizeof(S<T>)]类型的参数化类型，而A是被赋值（或者初始化）的指针（即下面的pf）所代表的函数类型。例如：

```
template<typename T> void f(T);           // F就是T

template<typename T> void g(T&);         // F仍然是T

double x[20];
int const seven = 7;

f(x); // 非引用参数（针对f）：T（decay转化）是double*
g(x); // 引用参数（针对g）：T是double[20]
f(seven); // 非引用参数：T是int（忽略const限定符）
g(seven); // 引用参数：T是int const
f(7); // 非引用参数：T是int
g(7); // 引用参数：T是int=>错误：不能把7传递给int&
```

11.2 演绎的上下文

对于比T复杂很多的参数化类型，也可以与给定的实参进行匹配。如下：

```
template<typename T>
void f1(T*)

template<typename E, int N>
void f2(E(&)[N]); // 数组、引用参数

template<typename T1, typename T2, typename T3>
void f3(T1 (T2::*)(T3*)) ; // 类的成员函数指针

class S
{
public:
    void f(double*);
};

void g(int*** ppp)
{
    bool b[42]; // 演绎T=int***
    f1(ppp); // 演绎E=bool, N为42.
    f3(&S::f); // 演绎T=void,T2=S,T3= double.
}
```

复杂的类型声明都是产生自（比它）基本的构造（例如指针、引用、数组、函数声明子（declarators）；成员指针声明子、template-id等）；匹配过程是从最顶层的构造开始，然后不断递归各种组成元素（即子构造）。我们可以认为：大多数的类型声明构造都可以使用这种方式进行匹配，这些构造也被称为演绎的上下文。然而，某些构造就不能作为演绎的上下文，如：

- 受限的类型名称。例如，一个诸如Q<T>::X的类型名称不能被用来演绎模板参数T。
- 除了非类型参数之外，模板参数还必须包含其他成分的非类型参数化形式。例如，诸如S<1+1>的类型名称就不能用来演绎（如果是S<I>就可以）。另外，一个不能演绎的上下文并没有自动地表明：所对应的程序就是错误的，或者前面分析的参数不能再次进行类型演绎。

11.3 特殊的演绎情况

存在两种特殊情况，其中用于演绎的实参-参数对（A，P）并不是分别来自于函数调用的实参和函数模板的参数。第1种情况出现在取函数模板地址的时候。在这种情况下，P是函数模板声明子的参数化类型（即下面的f的类型），而A是被赋值（或者初始化）的指针（即下面的pf）所代表的函数类型。例如：

```
template<typename T>
void f(T, T);

void (*pf)(char, char) = &f;
```

在上面的代码中，P就是void(T, T),而A是void(char, char)。用char替换T，该演绎过程是成功的。另外，pf被初始化为“特化f<char>”的地址。

另一种特殊情况和转型运算符模板一起出现。如：

```
class S
{
public:
    template<typename T, int N> operator T[N](&);
};
```

在这种情况下，实参-参数对（A，P）涉及到我们试图进行转型的实参和转型运算符的返回类型。下面的代码清楚地说明了这种情况：

```
void f(int (&)[20]);

void g(S s)
{
    f(s);
}
```

在此，我们试图把S转型为int(&)[20];因此，类型A为int[20]，而类型P为T[N]。于是，用类型int替换T，用20替换N之后，该演绎就是成功的。

11.4 可接受的实参转型

通常，模板演绎过程会试图找到函数模板参数的一个匹配，以参数化类型P等同于类型A。然而，当找不到这种匹配的时候，下面的几种变化就是可接受的：

- 如果原来声明的参数是一个引用参数，那么被替换的P类型可以比A类型多一个const或者volatile限定符。
- 如果A类型是指针类型或者成员指针类型，那么它可以用限定符转型（就是说，添加const或者volatile限定符），转化为被替换的P类型；
- 当演绎过程不涉及取函数模板地址的时候，被替换的P类型可以是A类型的基类；或者当A是指针类型时，P可以是一个指针类型，它所指向的类型是A所指向的类型的基类。如：

```
template<typename T>
class B
{
};

template<typename T>
class D : public B<T>
{
};

template<typename T> void f(B<T>*>);
void g(D<long> dl)
{
    f(&dl); // 成功演绎：用long替换T
}
```

只有在精确匹配不存在的情况下，才会出现这种宽松的匹配。即使这样，只有在前面添加的几种转型中能找到一种替换，并且借助这种替换可以匹配A类型和P类型，演绎过程才能是成功的。

11.5 类模板参数

模板实参演绎只能应用于函数模板和成员函数模板，是不能应用于类模板的。另外，对于类模板的构造函数，也不能根据实参来演绎类模板参数。如：

```
template <typename T>
class S
{
public:
    S(T b) : a(b) {}
private:
    T a;
};

S s(12); // 错误：不能从构造函数的调用实参12演绎类模板参数T
```

11.6 缺省调用实参

和普通函数一样，在函数模板中也可以指定缺省的函数调用实参。例如：

```
template<typename T>
void init(T* loc, T const& val = T())
{
    *loc = val;
}
```

如例子所示，缺省调用实参是可以依赖于模板参数的。但是，只有在没有提供显式实参的情况下，才会实例化这种依赖型的缺省实参——这也使得下面例子有效的一条规则：

```
class S
{
public:
    S(int, int);
};



S s(0, 0);
int main()
{
    // 因为T=S, 所以T()就是无效的了。于是缺省调用实参T()也就不需要进行实例化，因为已经提供了一个显式参数
    init(&s, S(7, 42));
}
```

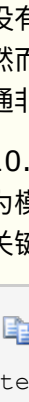

对于缺省调用实参而言，即使不是依赖型的，也不能用于演绎模板实参。这意味着下面的C++程序是无效的：

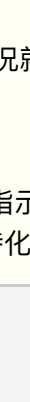
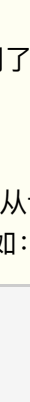
```
template<typename T>
void f(T x = 42){}

int main()
{
    f<int>(); // 正确：T= int
    f(); // 错误：不能根据缺省调用实参42来演绎T
}
```

分类：C++ Template

好文顶顶 关注我 收藏该文  

 关注 - 63  粉丝 - 96

[+加关注](#)  推荐  反对

« 上一篇: [C++ template —— 模板中的名称（三）](#)

» 下一篇: [C++ template —— 模板特化（五）](#)

posted @ 2016-01-25 14:30 小天_y 阅读(2100) 评论(0) 编辑 收藏 举报

[刷新评论](#) [刷新页面](#) [返回顶部](#)