C++ template —— template metaprogram（九）

metaprogramming含有“对一个程序进行编程”的意思。换句话说，编程系统将会执行我们所写的代码，来生成新的代码，而这些新代码才真正实现了我们所期望的功能。通常而言，metaprogramming这个概念意味着一种反射的特性：metaprogramminig组件只是程序的一部分，而且它也只生成一部分代码或者程序。

使用metaprogramming的目的是为了实现更多的功能，并且是花费的开销（代码大小，维护的开销等来衡量）更小。另一方面，metaprogramming的最大特点在于：某些用户自定义的计算可以在程序翻译期进行。而这通常都能够在性能或接口简单性方面带来好处；甚至为两方面同时带来好处。

本篇讲解的metaprogramming概念要依赖于前面关于trait和类型函数的讨论。

17.1 metaprogram的第一个实例（递归模板）

模板实例化机制是一种基本的递归语言机制，可以用于在编译期执行复杂的计算。因此，这种随着模板实例化所出现的编译期计算通常就被称为template metaprogramming。

看一个简单的例子：如何在编译期计算3的幂

```
//meta/pow3.hpp
#ifndef POW3_HPP
#define POW3_HPP

// 用于计算3的N次方的基本模板
template <int N>
class Pow3
{
    public:
        enum { result = 3 * Pow3<N-1>::result };
};

// 用于结束递归的全局特化
template<>
class Pow3<0>
{
    public:
        enum { return = 1 };
};

#endif      // POW3_HPP
```

在这里，Pow3<>模板（包含它的特化）就被称为一个template metaprogramming。它描述一些可以在翻译期（编译期）进行求值的计算，而这整个求值过程属于模板实例化过程的一部分。

17.2 枚举值和静态常量

在原来的C++编译器中，在类声明的内部，枚举值是声明“真常值”（也称为常量表达式）的唯一方法。然而，现在C++的标准化过程引入了在类内部进行静态常量初始化的概念。我们可以作如下修改上面的例子：

```
//meta/pow3b.hpp
#ifndef POW3_HPP
#define POW3_HPP

// 用于计算3的N次方的基本模板
template <int N>
class Pow3
{
    public:
        static int const result = 3 * Pow3<N-1>::result;
};

// 用于结束递归的全局特化
template<>
class Pow3<0>
{
    public:
        static int const result = 1;
};

#endif      // POW3_HPP
```

新的例子中我们使用静态常量成员而不是枚举值。然而，该版本存在一个缺点：静态成员变量只能是左值。因此，如果你具有一个如下的声明：

```
void foo(int const&);
```

而且你把上一个metaprogram的结果传递进去，即：

```
foo(Pow3<7>::result);
```

那么编译器将必须传递Pow3<7>::result的地址，而这会强制编译器实例化静态成员的定义，并为该定义 分配内存。于是，该计算将不再局限于完全的“编译期”效果。然而，枚举值却不是左值（也就是说，它们并没有地址）。因此，当你通过引用传递枚举值的时候，并不会使用任何静态内存，就像是以文字常量的形式传递这个完成计算的值一样。所以，下面的所有例子，我们使用枚举值而不是静态常量。

17.3 第二个例子：计算平方根

```
// meta/sqrt1.hpp

#ifndef SQRT_HPP
#define SQRT_HPP

// 用于计算sqrt(N)的基本模板
template <int N, int LO = 0, int HI = N>
class Sqrt
{
    // 计算中点
    enum { mid = (LO + HI +1) / 2};

    // 借助二分查找一个较小的result
    enum { return = (N<mid*mid) ? Sqrt<N, LO, mid-1>::result : Sqrt<N, mid, HI>::result };
};

// 局部特化，适用于LO等于HI
template<int N, int M>
class Sqrt<N, M, M>
{
    public:
        enum { result = M };
};

#endif    // SQRT_HPP
```

现在考虑当编译器试图计算下面表达式的时候：

```
(16<=8*8) ? Sqrt<16, 1, 18>::result : Sqrt<16, 9, 16>::result
```

这时候，编译器会实例化"?:"运算符两边的模板，这会产生数量庞大的实例化体，总数大约是N的两倍。这并不是我们所期望的，因为对于大多数编译器而言，模板实例化通常都会是一个代价高昂的过程，特别对于内存开销而言。所以我们放弃使用"?:"运算符，而是使用我们在前面xxxxx博文讲解过的IfThenElse模板：

```
// meta/sqrt2.hpp

#include "ifthenelse.hpp"

// 用于主要递归步骤的基本模板
template <int N, int LO = 0, int HI = N>
class Sqrt
{
    // 计算中点
    enum { mid = (LO + HI +1) / 2};

    // 借助二分查找一个较小的result
    typedef typename IfThenElse<(N<mid*mid), Sqrt<N, LO, mid-1>, Sqrt<N, mid, HI> >::ResultT S
ubT;

    enum { result = SubT::result };
};

// 局部特化，适用于LO等于HI
template<int N, int S>
class Sqrt<N, S, S>
{
    public:
        enum { result = S };
};
```

可以把IfThenElse看成一个简易装置（实际上是模板），它能根据给定布尔常量的值，在两个类型中选择出其中一个。记住：为一个类模板实例定义一个typedef并不会导致C++编译器实例化该实例的实体。

17.4 使用归纳变量

详见书籍，不作笔记

17.5 计算完整性

Pow3<>和Sqrt这两个例子说明：一个template metaprogram可以包含下面几部分：

（1）状态变量：也就是模板参数。

（2）迭代构造：通过递归

（3）路径选择：通过使用条件表达式或者特化

（4）整型（即枚举里面的值应该为整型）算法

模板实例化通常都要消耗巨大的编译器资源，而且扩展的递归实例化也会很快地降低编译器的效率，甚至耗光所有的可用资源。

C++标准建议最多只进行17层的递归实例化，但实际开发中又很容易就超过这个限制。然而，在某些情况下，metaprogram又是实现高效率模板的一个不可替代的工具。

17.6 递归实例化和递归模板实参

书中在本节向我们介绍了一个例子，表明当使用递归模板实参的时候，编译器为每个类型保存一个mangled name将会变得非常大。故而，在其他条件都相同的情况下，在组织递归实例化的时候，我们仍然（趋向于）避免在模板实参中使用递归嵌套的实例化。

17.7 使用metaprogram来展开循环 这是本篇博文一个实用的应用程序，用于展开数值计算的循环：

```
// meta/loop1.hpp

#ifndef LOOP1.HPP
#define LOOP1.HPP

template <typename T>
inline T dot_product(int dim, T* a, T* b)
{
    T result = T();
    for (int i = 0; i < dim; ++i)
    {
        result += a[i]*b[i];
    }
    return result;
}

#endif // LOOP1.HPP
```

上面程序的问题在于：对于许多迭代，编译器通常都会优化这种循环（即迭代），而在这个例子中，这种优化却会带来反面的效果（why？）。

如果实用了旨在执行千万次点乘计算的程序库组件，那么差别可能就会很大了。template metaprogramming为我们解决了这个问题。程序修改如下：

```
// meta/loop2.hpp

#ifndef LOOP2.HPP
#define LOOP2.HPP

// 基本模板
template <int DIM, typename T>
class DotProduct
{
    public:
        static T result (T* a, T* b){
            return *a * *b + DotProduct<DIM-1, T>::result(a+1, b+1);
        }
};

// 作为结束条件的局部特化：一元vector的情况
template <typename T>
class DotProduct<1, T>
{
    public:
        static T result (T* a, T* b){
            return *a * *b;
        }
};

// 辅助函数
template <int DIM, typename T>
inline T dot_product(T* a, T* b)
{
    return DotProduct<DIM, T>::result(a, b);
}

#endif // LOOP2.HPP
```

可以如下调用：

```
dot_product<3>(a, b);
```

这个表达式将实例化一个辅助函数模板，而在此函数模板内部将会直接调用：

```
DotProduct<3, int>::result(a, b);       // 模板实参分别是：非类型模板参数，通过函数模板实参演绎得到的模板
参数
```

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：C++ template —— 模板与继承（八）
» 下一篇：C++ template —— 表达式模板（十）

posted @ 2016-02-22 15:37 小天_y 阅读(1225) 评论(0) 编辑 收藏 举报