kimi's Blog 我的C++窝

C++ Meta Programming 和 Boost MPL(3)

标签： c++ class struct 数据结构 编程 数学计算

2007-08-30 23:06 1468人阅读 评论(0) 收藏 举报

分类： C++ (22)

目录(?) [+]

本系列全部转载自kuibyshev.bokee.com

1. 模板元编程的基本用途

1.1. 数值计算

上文提及的所有关于模板元编程的例子都是在编译时的数值计算，这是模板元编程的最简单直接的使用方法。数值计算主要利用模板的特化和局部特化能力进行递归演算，模板类被视为元函数，利用类中的一个静态常量保存结果。由于C++模板对非类型的参数有限制，一般只有整型和布尔型可以参加运算。元函数的结果一般放在一个静态的常量中，但对于整型而言还有一个更好的选择，可以放置在枚举中，其唯一的优点是静态常量是左值必须在运行期占有内存地址，也就是分配了内存空间，而枚举值不能作为左值，所以不占用内存空间，显得有些微的优势。这样，阶乘计算的例子可以改写如下：

```
#include <iostream>

template <unsigned n>
struct factorial
{
    enum {
        value = n * factorial<n-1>::value
    };
};
template<>
struct factorial<0>
{
    enum {
        value = 1
    };
};
int main()
{
   std::cout<<factorial<6>::value<<std::endl; //6!=720
}
```

无论是编译期的递归还是运行期内存的节省，对比起模板元编程在数值计算上的不足，都显得有点得不偿失。主要有以下四点：

- 运算范围仅限于整型和布尔型，用途有限。
- 递归式的编程难于实行和理解。
- C++编译器对编译期的递归都是有一定的限制的，C++标准建议进行17层递归实例化，这无疑不能满足稍复杂的程序。
- 大量消耗编译器资源的同时，降低了编译效率。

一个STL风格的动态二维 (856)
C++ Meta Programming (855)
C# - Ini文件类 (639)

评论排行

为什么C++ (2)
智能指针的标准之争：B (2)
一个STL风格的动态二维 (2)
你应当如何学习C++(以及 (2)
boost.shared_ptr源码整理 (1)
我看国内的C++教育以及 (1)
C++ Meta Programming (1)
《C++0x漫谈》系列之： (0)
《C++0x漫谈》系列之： (0)
C#中const与readonly字 (0)

推荐文章

* CSDN日报20170429 ——《程序修行从“拔刀术”到“万剑诀”》
* 抓取网易云音乐歌曲热门评论生成词云
* Android NDK开发之从环境搭建到Demo级十步流
* 个人的中小型项目前端架构浅谈
* 基于卷积神经网络(CNN)的中文垃圾邮件检测
* 四无年轻人如何逆袭

最新评论

C++ Meta Programming 和 Boos
code_pipeline: 大哥，这个代码错了 template< unsigned n > struct fa...
为什么C++
don: 总觉得刘某就是针对linus在说。
为什么C++
don: C++肯定还是有用的，这点毫无疑问
我看国内的C++教育以及我的建i
coolage31: 多态与虚拟是其核心，这也是所有面向对象的核心。 你的心得总结真的写的很好~但这两者也真的很难吃透现...
一个STL风格的动态二维数组
laibach0304: 恩，没错 完全原创
一个STL风格的动态二维数组
jxlczjp77: 老兄这时你自己写的吗？看着恐怖，这么长.^_^
智能指针的标准之争：Boost vs.
laibach0304: 呵呵,欢迎光临. 我也准备尝试新的东西了,现在还在观望中,但是C++我还是会坚持下去的. 不提马光...
智能指针的标准之争：Boost vs.
guokeno1: 我来也! 不错,老兄你研究的东西估计马光志抓狂也弄不明白了..... 我现在不怎么用C++了,...
你应当如何学习C++(以及编程)
laibach0304: 呵呵，转载的文章，没仔细排版了，见谅
你应当如何学习C++(以及编程)
BillEasy: 看起来太累.字体大小格式应该整理下。

因此，用模板元编程作编译期数值计算并不在实践中经常使用，但它作为一个“中心设施”在MPL库中发挥着重要的作用[6]。

1.2. 解开循环（Loop Unrolling）

当计算两个向量进行点乘，而维数不定时，例如：

```
int a[]={1,3,5,7};
int b[]={2,4,6,8};
```

考虑下面计算点乘的代码：

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

这里的代码很平常，但对于性能要求极高并大量使用点乘的应用程序，也许想再节省一点开销，如果能减少循环的计数，对性能也有比较可观的提升。这样代码应该展开以直接计算：

```
T result=a[0]*b[0]+a[1]*b[1]+a[2]*b[2]+a[3]*b[3];
```

但是我们希望泛化这个表达式，以便应用于不同维数的向量计算，这里，模板元编程正好可以发挥出它编译时计算和生成代码的能力。我们可以把代码改写成：

```
template <int DIM, class T>
struct DotProduct {
      static T result(T* a, T* b) {
            return *a * *b + DotProduct<DIM-1, T>::result(a+1, b+1);
      }
};
//局部特化，用于结束递归
template <class T>
struct DotProduct<1, T> {
      static T result(T* a, T* b) {
            return *a * *b;
      }
};
//包装函数
template <int DIM, class T>
inline T doc_product(T* a, T* b) {
      return DotProduct<DIM, T>::result(a, b);
}
```

这种方法是定义了一个类模板DotProduct作为元函数，通过递归调用不断展开表达式，还定义了一个局部特化的版本，使它在维数递减到1时能够终结递归。

我们还留意到一个习惯，元函数都用struct而不是用class定义，这是因为struct中的成员可见性默认为public，在编写元函数时可以省却public:这个声明。

注意包装函数的接口已经改变，利用普通方法的函数是这样使用的：

```
doc_product(4, a, b);
```

现在的写法是：

```
doc_product<4>(a, b);
```

为什么不能使用相同的接口呢？原因是模板参数必须在编译时确定，所以DIM必须是一个常量，而不可能是一个变量。所以这是对此种编程技术的一个重大限制。当一般都能在编译时确定。

Todd Veldhuizen在1995年第一次提出了这项技术，并且把这种技术运用到高性能数学计算的Blitz++库中。此外，在Blitz++库中还大量运用到一种称为“表达式模板（Expression Template）”的技术，同样是为了减少线性代数计算中的循环次数和临时对象的开销。表达式模板尽管不属于模板元编程的范畴（因为它不是依赖编译时计算进行优化的），但它与模板元编程具有异曲同工的妙用，特别在高性能数学计算中能够发挥极大的用途。Todd Veldhuizen指出，通过这一系列的优化手段，C++在科学计算上的性能已经达到甚至超过Fortran的性能。

1.3. 类型处理

对类型的处理是模板元编程最重要和最具有现实意义的应用。由于模板可以接受类型参数，也可以通

关闭

过typedef或定义内嵌类建立模板类的成员类型，再加以强大的模板特化的能力，使得类型计算几乎能有着数值计算所有的全部能力。

1.3.1. 类型分支选择

利用模板局部特化的能力，编译时的类型选择可以很容易做到：

```
//默认值，如果C为true就把第二个类型作为返回值
template<
      bool C
    , typename T1
    , typename T2
    >
struct if
{
    typedef T1 type;
};
//局部特化，如果C为false就把第二个类型作为返回值
template<
      typename T1
    , typename T2
    >
struct if<false,T1,T2>
{
    typedef T2 type;
};
```

不过，有某些旧式编译器并不支持模板局部特化，这种情况下增加一层包装就可以巧妙地转为使用全局特化。

```
template< bool C >
struct if_impl
{
    template< typename T1, typename T2 > struct result
    {
        typedef T1 type;
    };
};

template<>
struct if_impl<false>
{
    template< typename T1, typename T2 > struct result
    {
        typedef T2 type;
    };
};

template<
      bool C
    , typename T1
    , typename T2
    >
struct if
{
    typedef typename if_impl< C >
        ::template result<T1,T2>::type type;
};
```

元函数if是模板元编程中最简单但运用得最多的基础设施。

1.3.2. 类型的数据结构

把类型作为普通数据一样管理，这初看起来有点匪夷所思：普通数据可以运用struct或者array来组织

关闭

，但C++并没有为类型提供一个专用的数据结构，可以利用的唯一设施是模板的参数列表。比如我们可以定义一个类型的“数组”如下

```
template <class a, class b, class c, class d, class e>
struct type_array;
```

但是为了使它真正像数组一样使用，还需要在其中定义一系列的typedef，比如某一下标的类型的提取等，类似于：

```
typedef a type1;
typedef b type2;
……
```

在这里，数组长度也无法动态变化是一个重大的缺陷。当然，有时候数组仍然是有用的，MPL就提供了一个框架为任何自定义的类型数据结构提供支持，下文会有所提及。现在先介绍一种更自动的类型组织方法——Typelist。

（2）Typelist

上面提到过模板元编程是函数式的编程，参照其他一些函数式编程语言对数据的组织，很容易得到一些启发。比如在Scheme（一种LISP的变体）中，基本的数据结构是表，其他数据结构都必须用表的形式来表达。一个表可以这样定义：

（”A” (“B” () (“C” () ())) (“D” () ()) ）

这个表可以表示出一个二叉搜索树：

A
B
C
D

通过简单的表的递归，就可以构造出各种数据结构。注意到C++模板的实现体也是一种类型，利用类型的递归，同样的风格也可以吸收到模板元编程中。

Typelist的定义是很简单的：

```
template <class T, class U>
struct Typelist
{
    typedef T Head;
    typedef U Tail;
};
```

另外我们需要定义一个特殊的标记：

```
struct Nulltype;
```

这个无定义的类不能产生对象，它的存在仅仅为了提供一个结束的标记。现在我们定义一个Typelist，并在这一节里面反复使用：

```
typedef Typelist<int,
   Typelist<float,
      Typelist<long, Nulltype> > >
typelist;
```

这样的结构比起“数组”有什么优点呢？由于它的结构是递归的，我们可以很容易写一个通用的元函数提取它某一个位置的类型，我们甚至可以插入、修改和删除元素，然后返回一个新的Typelist。

（3）提取Typelist中的类型

如果需要按照位置来提取Typelist类型，可以定义这样一个元函数：

```
//声明
template <class List, unsigned int i> struct typeat;
//局部特化，当处理到需要提取的位置时，Head就是要返回的类型
template <class Head, class Tail>
struct typeat<Typelist<Head, Tail>, 0>
{
    typedef Head result;
};
//如果未到需要提取的位置，在下一个位置继续递归
template <class Head, class Tail, unsigned int i>
struct typeat< Typelist<Head, Tail>, i>
{
    typedef typename typeat<Tail, i-1>::result result;
};
```

这里使用了局部特化，对于不能支持局部特化的编译器，可以类似上面的if元函数的处理手法，适当

关闭

修改这里的代码。
这个元函数按照以下方式调用：

```
typedef typeat<typelist, 0>::result result;
```

如果试图定义一个变量：

```
result a=1.2f;
```

编译器会抱怨无法把一个float类型转换为result类型，因为上面定义的typelist的第一个类型是int。而把下标0改为1以后，则可以编译通过，这证明元函数可以在编译时正确选择出所需的类型。

（3） 修改Typelist中的元素

实例化以后的模板已经成为一种类型，所以是不可能进行修改的，要达到修改的效果，唯一的方法是返回一种新的类型，使它带有新的Typelist。
比如，如果要在一个已有的Typelist中添加一个类型，可以定义一个这样的元函数：

```
//声明
template <class List, class T> struct append;
//如遇到把空类型加入空类型，只需要返回 一个空类型
template <>
struct append<Nulltype, Nulltype>
{
    typedef Nulltype result;
};
//如果把一个非空类型加入空类型，那么就可以直接返回
//一个只有一个元素的Typelist
template <class T> struct append <Nulltype, T>
{
     typedef  Typelist<T, Nulltype> result;
};
//如果把一个Typelist加入空类型，那么就可以
//直接返回这个Typelist
template <class Head, class Tail>
struct append <Nulltype, Typelist<Head, Tail> >
{
    typedef Typelist<Head, Tail> result;
};
//当未到达Typelist尾部（遇到空类型）时，递归调用append元函数
template <class Head, class Tail, class T>
struct append <Typelist<Head, Tail>, T>
{
    typedef Typelist<Head,
        typename append < Tail, T>::result>
    result;
};
```

这个append元函数不仅能插入一个类型，也可以把一个Typelist添加到另一个Typelist尾部。如果这样使用的话：

```
typedef append<typelist, double>::result newlist;
```

那么就会得到一个新的newlist，里面共有4个类型。
利用很类似的方法，还可以为Typelist编写删除或者修改某个元素的元函数，这可以参阅[18]的论述。

（4） 小结

用合适的数据结构组织类型，我们可以得到这样的一些好处：

- 编译时可以根据情况自动决定选择的类型。
- 由于模板中没有数据成员，它们在运行时
- 可以运用于代码的自动生成，或者实现某种设计模式。

下面分析Boost的MPL库时，还会遇到更多和更复杂的类型数据结构。

1.4. 自动生成代码

上面曾经提到的解开循环技术其实已经是一种代码生成了，但这个独立出来的小节专门用于说明代码生成在设计模式中的应用，这些精巧的设计承接着上文提到的Typelist，也是来源于Alexandrescu的Loki库[18]。
首先，引入一个类模板GenScatterHierarchy，这个模板通过递归的定义进行多重继承，从而构造出一种复杂而散乱的体系。

```
//模板定义
```

关闭

```
    template <class TList, template <class> class Unit>
class GenScatterHierarchy;
//当未到达Typelist尾部时，继承Head和Tail各一次
template <class T1, class T2, template <class> class Unit>
class GenScatterHierarchy<Typelist<T1, T2>, Unit>
   : public GenScatterHierarchy<T1, Unit>
   , public GenScatterHierarchy<T2, Unit>
{
};
//遇到单个Unit类型时，直接继承Unit
    template <class AtomicType, template <class> class Unit>
class GenScatterHierarchy : public Unit<AtomicType>
{
};
   //遇到NullType时，不继承
template <template <class> class Unit>
class GenScatterHierarchy<NullType, Unit>
{
};
```

另外还需要定义一个Holder来持有一个类型的对象，这也是为了包装基本类型，以便它们可以被继承。

```
template <class T> struct Holder
{
     T value;
};
```

现在可以定义一个GenScatterHierarchy的实现体了：

```
typedef GenScatterHierarchy<
    Typelist<int, Typelist<string, Typelist<Widget, NullType> > >,
    Holder>
WidgetInfo;
```

这里的Typelist嵌套定义显得有点繁琐，在Loki库里面有定义相对应的宏来简化使用，这里为了展示的方便而展开了。WidgetInfo的继承体系如下图所示：

为什么要使用这样复杂的一个体系呢？这种体系对于程序设计有什么实质性的帮助呢？首先，Typelist是一个可扩展的结构，不但可以定制任意长度的Typelist，在更改Typelist以后，WidgetInfo不需要作任何改变就可以自动适应并产生新的代码。其次，WidgetInfo继承了有Typelist长度个的Holder实体，它们拥有相同的接口，可以加入一个成员模板函数用统一的接口来操纵这些实体。再次，分别继承的Holder实体并不会互相干扰，WidgetInfo可以根据需要上调成它们中的任何一种。

Alexandrescu在Loki库中把这种体系发挥得淋漓尽致[18]，他以这个体系（以及其它辅助方法）构造出一个非常灵活的Abstract Factory模式，避免了Abstract Factory通常耦合度较高的缺点。

- C++ Meta Programming 和 ... 2007-08-30 阅读 908
- C++ Meta Programming 和 ... 2007-08-30 阅读 898
- 我看国内的C++教育以及我... 2007-08-28 阅读 602
- C++ Meta Programming 和 ... 2007-08-30 阅读 856
- 智能指针的标准之争：Boost... 2007-08-28 阅读 872
- 泛型快速排序 2007-08-28 阅读 448

更多文章

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

关闭