C++ Meta Programming 和 Boost MPL(2)

标签： c++ 编程 class struct lisp 语言

2007-08-30 23:02 856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. 模板元编程的原理

1.1. 函数式编程

1.1.1. 函数式编程的概念

从上面的例子可以看出，由于模板元编程借助的是C++模板的特化能力，使它的设计方法迥异于普通的C++编程习惯。比如我们不能够在元函数中使用变量，编译期显然只可能接受静态定义的常量，也就因此，我们不能够使用传统意义上的循环；要实现任何分支选择，唯一的方法是调用其它元函数或者使用递归。这种编程风格正是具有重要理论意义的函数式编程（Functional Programming）。
Kenneth C. Louden指出函数式编程有如下的特点：

- 所有的过程都是函数，并且严格地区分输入值（参数）和输出值（结果）。
- 没有变量或赋值（变量被参数替代）。
- 没有循环（循环杯递归调用替代）。
- 函数的值只取决于它的参数的值而与求得参数值的先后或者调用函数的路径无关。
- 函数是一类值。

上述的最后一点是一个很重要的性质。所谓“函数是一类值（First Class Value）”指的是函数和值是同等的概念，一个函数可以作为另外一个函数的参数，也可以作为值使用。如果函数可以作为一类值使用，那么我们就可以写出一些函数，使得这些函数接受其它函数作为参数并返回另外一个函数。比如定义了f和g两个函数，用compose(f,g)的风格就可以生成另外一个函数，使得这个函数执行f(g(x))的操作，则可称compose为高阶函数（Higher-order Function）。

1.1.2. 函数式编程在C++中的常用方法

如果排除上述的最后一点，那么C语言已经能完整模拟出函数式编程的风格，但是在C语言中，函数却并不能作为一类值。也许我们会想到函数指针，但是试想如果我们的函数需要返回另一个函数的指针：

```
typedef int(*IntProc) (int);

IntProc compose(IntProc f, IntProc g)
{
    int tempProc(int x)
    {
        return f(g(x));
    }
    return tempProc;
}
```

这个例子是不能通过编译的，因为C语言禁止在函数中定义函数。

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字段 (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。

幸运的是，我们可以在C++中利用类和模板来解决这个问题，因为C++允许定义（）操作符，建立所谓的仿函数（Functor）。所以对象既可以作为值来传递和调用，又可以像函数一样用obj(x)的语法来使用了；另一方面，利用模板对返回值的控制，就可以避免上面无法定义内部函数的矛盾了。例如在GCC的 STL中有一个不属于C++标准的compose1函数，可以接受两个定义了（）操作符的对象作为函数参数，并返回一个能进行f(g(x))运算的对象：

```
template <class Operation1, class Operation2>
class unary_compose
  : public unary_function<typename Operation2::argument_type,
                          typename Operation1::result_type> {
protected:
        Operation1 op1;
        Operation2 op2;
public:
        unary_compose(const Operation1& x, const Operation2& y) :
         op1(x), op2(y) {}
        typename Operation1::result_type
        operator()(const typename Operation2::argument_type& x) const {
                return op1(op2(x));
  }
};

template <class Operation1, class Operation2>
inline unary_compose<Operation1, Operation2>
compose1(const Operation1& op1, const Operation2& op2) {
        return unary_compose<Operation1, Operation2>(op1, op2);
}
```

1.1.3. 模板元编程中的高阶函数

那么，使用C++的模板机制又是否可以满足元函数作为一类值使用呢？答案是肯定的，不过解答稍稍有点复杂，并不像上述compose1的解决方法一样优雅。

```
#include <iostream>
using namespace std;

template<int n>
struct f
{
      static const int value=n+1;
};

template <int n>
struct g
{
      static const int value=n+2;
};

template <template <int n> class op1, template <int n> class op2>
struct compose
{
      template <int x>
      struct return_type
      {
            static const int value=op1<op2<x>::value>::value;
      };
};
int main()
{
   typedef compose<f, g>::return_type<6> h;
     cout<<h::value<<std::endl; //6+2+1=9
```

关闭

```
}
```

在这里，f和g是两个元函数，compose接受f和g作为参数，生成了一个可以计算f(g(x))的新函数，看起来能够得出正确的答案，但是却仍然有两个问题。

首先，在compose的模板参数中，不能直接使用类似于template <class op1, class op2>的写法，原因是C++的模板机制严格区分模板和类，我们无法把模板直接作为另一个模板的参数，唯一可行的方法是使用“作为类模板的模板参数（Class Template Template Parameter）”，这样就把f和g的参数类型限制死了。不过我们似乎仍然可以勉强接受这个限制，事实上模板机制对非类型的模板参数本来就存在着限制，现在的C++标准禁止浮点数、类对象和内部链接对象（比如字符串和全局指针）作为模板参数，所以非类型的模板参数实际上只剩下布尔型和整型可用，写成类似composeint和composebool 的两个类仍然有可行性（模板参数是无法重载的）。

其次，同样是C++的模板机制严格区分模板和类的缘故，返回值return_type是一个模板而并不是一个元函数（或者说是类）。

上述两点都隐含着一个最大共同问题，C++对“作为类模板的模板参数”作了很严格的限制，所以一旦定义了以后，其模板参数的个数不能改变。当然，在STL里面我们似乎习惯了这个限制并用重新定义函数的方式来避开这个限制。但在作为函数式编程的模板元编程里面，似乎应该要求以更优雅的方式来解决（事实上即使是常规编程下的高阶函数，也有函数库提供了更好的组合方式[5]）。

现在我们仅仅用到了模板元编程的数值计算能力，还没有引入它对类型的处理能力，稍后在分析MPL时会重新讨论到这个问题，还会显示出高阶函数更大的使用障碍。幸而MPL提供了很好[illegible]，通过增加包装层和使用lambda演算的概念，高阶函数依然能用上，使得模板元编程能够编程的要求。

Krzysztof Czarnecki曾利用模板元编程实现了一个很简单的LISP，而LISP就是典型的函数式编程语言。

总之既然C++模板能够使用函数式编程，那么也就意味着这个C++语言的子集已经是图灵完备的，因为任何计算都可以使用函数来描述，理论上模板元编程应该能完成图灵机上的一切运算。

当然，理论上的完备并不意味着实用性。尽管在C++中能够在某种程度上使用函数式编程的风格，但是从实用性和效率来说，大概没有程序员使用纯粹的函数编程方式。不过，在进行模板元编程的时候，由于语法的特殊性，却不得不使用纯粹函数式编程。因此，模板元编程与传统的C++命令式编程相差很大，并不符合大多数C++程序员的习惯，不但会带来编写上的困难，还增加了许多程序理解上的难度。那么，为什么要使用模板元编程呢？首先我们应当了解模板元编程能够做些什么，其次，模板元编程有可能用在什么地方。

上一篇 C++ Meta Programming 和 Boost MPL(1)
下一篇 C++ Meta Programming 和 Boost MPL(3)

关闭