C++ template —— 表达式模板（十）

表达式模板解决的问题是：
对于一个数值数组类，它需要为基于整个数组对象的数值操作提供支持，如对数组求和或放大：

```
Array<double> x(1000), y(1000);
...
x = 1.2 * x + x * y;
```

对效率要求苛刻的数值计算器，会要求上面的表达式以最高效的方式进行求值。想既高效又以这种比较紧凑的运算符写法来实现，表达式模板可以帮助我们实现这种需求。

谈到表达式模板，自然联想到前面的template metaprogramming。一方面：表达式模板有时依赖于深层的嵌套模板实例化，而这种实例化又和我们在template metaprogramming中遇到的递归实例化非常相似；另一方面：最初开发这两种实例化技术都是为了支持高性能的数组操作，而这又从另一个侧面说明了metaprogramming和表达式模板是息息相关的。当然，这两种技术还是互补的。例如，metaprogramming主要用于小的，大小固定的数组，而表达式模板则适用于能够在运行期确定大小、中等大小的数组。

18.1 临时变量和分割循环
书中本节讲解了使用普通的运算符重载方法，临时变量的构造和多次的读写操作带来的低效，以及使用包含计算的赋值运算符（如+=，*=等），即便可以避免过多的临时变量构造，却依旧进行了多次读写以致并没有带来明显的效率提高，同时引入并不雅观的代码。
有兴趣详见书籍；

18.2 在模板实参中编码表达式
前面的问题中，我们注意到：直到看到了整个表达式的时候（在我们的例子中，即在调用赋值运算符的时候），才对表达式的各个部分进行求值。下面我们要做的，就是把表达式：

```
1.2*x + x*y;
```

转化为一个具有如下类型的对象：

```
A_Add< A_Mult<A_Scalar<double>, Array<double> >,
             A_Mult<Array<double>, Array<double> > >
```

18.2.1 表达式模板的操作数
首先我们来实现A_Add和A_Mult，A_Scalar类：
---------------------- 标识1 ------------------------

```
// exprtmpl/exprops1.hpp

#include <stddef.h>
#include <cassert>
// 包含了一个辅助class trait template，从而可以根据不同的情况，判断究竟是以"传值"的方式还是以"传引用"的方式来引用对应的"表达式模板节点"：
#include "exprops1a.hpp"

// 表示两个操作数之和的对象的所属类
template <typename T, typename OP1, typename OP2>
class A_Add
{
    private:
        // 这里使用辅助类A_Traits的原因参见 "标识4"
        typename A_Traits<OP1>::ExprRef op1;        // 第1个操作数
        typename A_Traits<OP2>::ExprRef op2;        // 第2个操作数

    public:
        // 构造函数，用于初始化指向操作数的引用
        A_Add(OP1 const& a, OP2 const& b) : op1(a), op2(b) { }

        // 在求值的时候计算和
        // 但最外层的变量调用[]运算符的时候，会一层一层调用到最原始类型的[]运算符
        T operator[] (size_t idx) const
        {
            return op1[idx] + op2[idx];
        }

        // size代表最大的容量（大小）
        size_t size() const
        {
            assert (op1.size() == 0 || op2.size() == 0 || op1.size() == op2.size() );
            return op1.size() != 0 ? op1.size() : op2.size();
        }
};

// 表示两个操作数之积的对象的所属类
template <typename T, typename OP1, typename OP2>
class A_Mult
{
    private:
        typename A_Traits<OP1>::ExprRef op1;        // 第1个操作数
        typename A_Traits<OP2>::ExprRef op2;        // 第2个操作数

    public:
        // 构造函数，用于初始化对象指向操作数的引用
        A_Mult(OP1 const& a, OP2 const& b) : op1(a), op2(b) { }

        // 在求值的时候计算乘积
        // 但最外层的变量调用[]运算符的时候，会一层一层调用到最原始类型的[]运算符
        T operator[] (size_t idx) const
        {
            return op1[idx] * op2[idx];
        }

        // size代表最大的容量（大小）
        size_t size() const
        {
            assert (op1.size() == 0 || op2.size() == 0 || op1.size() == op2.size() );
            return op1.size() != 0 ? op1.size() : op2.size();
        }
};

exprtmpl/exprscalar.hpp
// 用于表示放到倍数的对象的所属类
template <typename T>
class A_Scalar
{
    private:
        T const& s;      // scalar的值
    public:
        // 构造函数，用于初始值
        A_Scalar (T const& v) : s(v) { }

        // 对于索引（下标）操作而言，每个元素的值都等于scalar（放大倍数）的值
        T operator[] (size_t) const {
            return s
        }

        // scalar 的大小（即元素个数）为0
        size_t size() const{
            return 0;
        }
};
```

18.2.2 Array 类型
---------------------- 标识2 ------------------------
既然能够使用轻量级的表达式模板来对表达式进行编码，下面我们创建一个Array:它既能够针对占用实际内存的数组，同时也适用于表达式模板。

```
// Rep类型要么是SArray,但前提是Array必须是一个占用实际存储空间的数组；要么是一个用于编码表达式的嵌套template-id,如A_Add和A_Mult。
// exprtmpl/exprarray.hpp
#include <stddef.h>
#include <cassert>
#include "sarray1.hpp"

template <typename T, typename Rep = SArray<T> >
class Array
{
    private:
        Rep expr_rep;       // （访问）数组的数据

    public:
        // 创建具有初始化大小的数组
        explicit Array (size_t s) : expr_rep(s) { }

        // 根据其他可能的表示来创建数组
        Array (Rep const& rb) ; expr_rep(rb) { }

        // 针对相同类型的赋值运算符
        Array& operator = (Array const& b) {
            assert (size() == b.size() );
            for (size_t idx = 0; idx < b.size(); ++idx)
            {
                expr_rep[idx] = b[idx];
            }
            return *this;
        }

        // 针对不同类型的赋值运算符
        template <typename T2, typename Rep2>
        Array& operator = (Array<T2, Rep2> const& b) {
            assert (size() == b.size() );
            for (size_t idx = 0; idx < b.size(); ++idx)
            {
                expr_rep[idx] = b[idx];
            }
            return *this;
        }

        // size 是所表示数据的大小
        size_t size() const{
            return expr_rep.size();
        }

        // 分别针对常量和变量的索引（下标）运算符
        T operator[] (size_t idx) const {
            assert(idx < size() );
            return expr_rep[idx];
        }

        T& operator[] (size_t idx) const {
            assert(idx < size() );
            return expr_rep[idx];
        }

        // 返回数组现在所表示的对象
        Rep const& rep() const {
            return expr_rep;
        }

        Rep& rep(){
            return expr_rep;
        }
};
```

18.2.3 运算符
---------------------- 标识3 ------------------------
到目前为止，我们只是实现了用于代表运算符的、针对数值Array模板的运算符操作（诸如A_Add），但仍然没有实现运算符本身（诸如+）。下面我们来做这件事：

```
// exprtmpl/exprops2.hpp
// 两个数组相加
template <typename T, typename R1, typename R2>
Array<T, A_Add<T, R1, R2> >
operator + (Array<T, R1> const& a, Array<T, R2> const& b){
    return Array<T, A_Add<T, R1, R2> >(A_Add<T, R1, R2>(a.rep(), b.rep() ));
}

// 两个数组相乘
template <typename T, typename R1, typename R2>
Array<T, A_Mult<T, R1, R2> >
operator * (Array<T, R1> const& a, Array<T, R2> const& b){
    return Array<T, A_Mult<T, R1, R2> >(A_Mult<T, R1, R2>(a.rep(), b.rep() ));
}

// scalar 和 数组相乘
template <typename T, typename R2>
Array<T, A_Mult<T, A_Scalar<T>, R2> >
operator * (T const& s, Array<T, R2> const& b){
    return Array<T, A_Mult<T, A_Scalar<T>, R2> >(A_Mult<T, A_Scalar<T>, R2>(A_Scalar<T>(s), b.rep() ));
}

// 数组和scalar相乘
// scalar和数组相加
// 数组和scalar相加
...
```

18.2.4 总结
针对前面的例子代码，我们来进行一个自顶向下的回顾：

```
int main()
{
    Array<double> x(1000), y(1000);
    ...
    x = 1.2*x + x*y;
}
```

（1）首先，编译器解析最左边的*运算符,它是一个scalar-array运算符：

```
template <typename T, typename R2>
Array<T, A_Mult<T, A_Scalar<T>, R2> >      // 返回类型
operator * (T const& s, Array<T, R2> const& b){
    return Array<T, A_Mult<T, A_Scalar<T>, R2> >             // 类型转换
            (A_Mult<T, A_Scalar<T>, R2>        // 模板参数
            (A_Scalar<T>(s), b.rep() ));        // 构造函数参数
}
```

其中操作数的类型是double和Array<double, SArray<double> >。因此，实际的结果类型是：

```
Array<double, A_Mult<double, A_Scalar<double>, SArray<double> > >
```

而结果值是一个构造自double值1.2的A_Scalar<double>对象，和一个表示对象x的SArray<double>对象。

（2）接下来，将会对第2个乘法进行求值：x*y是一个array-array操作：

```
template <typename T, typename R1, typename R2>
Array<T, A_Mult<T, R1, R2> >
operator * (Array<T, R1> const& a, Array<T, R2> const& b){
    return Array<T, A_Mult<T, R1, R2> >
            (A_Mult<T, R1, R2>(a.rep(), b.rep() ));
}
```

而两个操作数的类型都是Array<double, SArray<double> >, 因此结果类型为：

```
Array<double, A_Mult<double, SArray<double>, SArray<double> > >
```

这一次，A_Mult所封装的两个参数对象都引用了一个SArray<double>表示：即一个用于表示x对象，另一个用于表示y对象。

（3）最后，才对+运算符进行求值。这次还是array-array操作：

```
template <typename T, typename R1, typename R2>
Array<T, A_Add<T, R1, R2> >
operator * (Array<T, R1> const& a, Array<T, R2> const& b){
    return Array<T, A_Add<T, R1, R2> >
            (A_Add<T, R1, R2>(a.rep(), b.rep() ));
}
```

其中用double来替换T,R1则用：

```
A_Mult<double, A_Scalar<double>, SArray<double> >
```

进行替换，而R2则替换为：

```
A_Mult<double, SArray<double>, SArray<double> >
```

因此，赋值运算符右边的表达式最终的类型为：

```
Array<double,
            A_Add<double,
                    A_Mult<double, A_Scalar<double>, SArray<double> >,
                    A_Mult<double, SArray<double>, SArray<double> > > >
```

这个类型将与Array模板的赋值运算符模板进行匹配：

```
template <typename T, typename Rep = SArray<T> >
class Array
{
    public:
        ...

        // 针对相同类型的赋值运算符
        Array& operator = (Array const& b) {
            assert (size() == b.size() );
            for (size_t idx = 0; idx < b.size(); ++idx)
            {
                expr_rep[idx] = b[idx];
            }
            return *this;
        }

        ...
};
```

其中赋值运算符将会运用右边Array（即b）的下标运算符来计算目标数组x的每一个元素，其中右边Array的实际类型为：

```
A_Add<double,
                    A_Mult<double, A_Scalar<double>, SArray<double> >,
                    A_Mult<double, SArray<double>, SArray<double> > >
```

如果我们仔细跟踪这个下标操作，那么对于一个给定的下标x，将会得到：

b[idx]实际展开成：(1.2*x[idx]) + (x[idx]*y[idx])

也即： x = 1.2*x + x*y;

表达式首先根据"标识3"规则进行展开，然后，再进一步根据"标识1"进行展开，在"标识1"中，针对每一个原始类型执行实际的数值操作（包括下标运算符操作）。

---------------------- 标识4 ------------------------
标识1中的运算符类使用了一个辅助类A_Traits，来定义操作数成员，是必要的：

```
 typename A_Traits<OP1>::ExprRef op1;        // 第1个操作数
  typename A_Traits<OP2>::ExprRef op2;        // 第2个操作数
```

原因在于：通常而言，我们可以把这些操作数声明为引用类型，因为大多数局部节点是在顶层表达式进行绑定的，因此它们的生命周期能够延续到完整表达式的求值。但是，唯一的例外是A_Scalar节点，它是在运算符函数内部进行绑定的，所以并不能一直存在到完整表达式的求值。因此，为了使放到倍数的成员能够一直存在到完整表达式求值，我们需求对scalar操作数进行"传值拷贝"，而不是"传引用拷贝"。也即：

（1）通常情况下是常数引用：

```
OP1 const& op1;     // 指向第1个操作数的引用
OP2 const& op2;     // 指向第2个操作数的引用
```

（2）对于scalar值，则是普通值：

```
OP1 op1;        // 以传值拷贝的方式引用第1个操作数
OP2 op2;        // 以传值拷贝的方式引用第2个操作数
```

故而，trait class 定义如下：它定义了一个针对大多数常数引用的基本模板，但同时定义了一个针对scalar的特化：

```
// exprtmpl/expropsla.hpp

/* 用于选择如何引用"表达式模板节点"的辅助trait class
*    - 通常情况下：传引用
*  - 对于scalar：传值
*/

template <typename T> class A_Scalar;

// 基本模板
template <typename T>
class A_Traits
{
    public:
        typedef T cosnt& ExprRef;      // 所引用的类型typedef成一个常量引用
};

// 针对scalar的局部特化
template <typename T>
class A_Traits<A_Scalar<T> >
{
    public:
        typedef A_Scalar<T> ExprRef;            // 所引用的类型实际是一个普通值
};
```

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 —— template metaprogram（九）
» 下一篇： C++ template —— 类型区分（十一）

posted @ 2016-02-22 15:45 小天_y 阅读(2221) 评论(0) 编辑 收藏 举报

刷新评论 刷新页面 返回顶部