

C++ template —— 模板与继承 (八)

16.1 命名模板参数

许多模板技术往往让类模板拖着一长串类型参数；不过许多参数都设有合理的缺省值，如：

```
template <typename policy1 = DefaultPolicy1,
          typename policy2 = DefaultPolicy2,
          typename policy3 = DefaultPolicy3,
          typename policy4 = DefaultPolicy4>
class BreadSlicer
{
    ....
};
```

一般情况下使用缺省模板实参BreadSlicer<>就足够了。不过，如果必须指定某个非缺省的实参，还必须明白地指定在它之前的所有实参（即使这些实参正好是缺省类型，也不能偷懒）。跟这样的BreadSlicer<DefaultPolicy1, DefaultPolicy2, Custom>相比，BreadSlicer<Policy3 = Custom>显然更有吸引力。这也是本节要介绍的内容。

我们的考虑主要是设法将缺省类型值放到一个基类中，再根据需要通过派生覆盖掉某些类型值。这样，我们就不再直接指定类型实参了，而是通过辅助类完成，如BreadSlicer<Policy3_Is<Custom> >。既然用辅助类做模板参数，每个辅助类都可以描述上述4个policy中的任意一个，故所有模板参数的缺省值均相同：

```
template <typename PolicySetter1 = DefaultPolicyArgs,
          typename PolicySetter2 = DefaultPolicyArgs,
          typename PolicySetter3 = DefaultPolicyArgs,
          typename PolicySetter4 = DefaultPolicyArgs>
class BreadSlicer
{
    typedef PolicySelector<PolicySetter1, PolicySetter2,
                          PolicySetter3, PolicySetter4>
        Policies;
    // 使用Policies::P1, Policies::P2, .....来引用各个Policies
};
```

剩下的麻烦事就是实现模板PolicySelector。这个模板的任务是利用typedef将各个模板实参合并到一个单一的类型（即Discriminator），该类型能够根据指定的非缺省类型（如policy1-is的Policy），改写缺省定义的typedef成员（如Default Policies的DefaultPolicy1）。其中合并的事情可以让继承来干：

```
// PolicySelector<A, B, C, D>生成A, B, C, D作为基类
// Discriminator<>使Policy Selector可以多次继承自相同的基类
// PolicySelector不能直接从Setter继承
template <typename Base, int D>
class Discriminator : public Base{
};

template <typename Setter1, typename Setter2,
          typename Setter3, typename Setter4>
class PolicySelector : public Discriminator<Setter1, 1>,
                      public Discriminator<Setter2, 2>,
                      public Discriminator<Setter3, 3>,
                      public Discriminator<Setter4, 4>{
};
```

注意，由于中间模板Discriminator的引入，我们就可以一致处理各个Setter类型（不能直接从多个相同类型的基类继承，但可以借助中间类间接继承）。如前所述，我们还需要把缺省值集中到一个基类中：

```
// 分别命名缺省policies为P1, P2, P3, P4
class DefaultPolicies
{
    public:
        typedef DefaultPolicy1 P1;
        typedef DefaultPolicy2 P2;
        typedef DefaultPolicy3 P3;
        typedef DefaultPolicy4 P4;
};
```

不过由于会多次从这个基类继承，我们必须小心以避免二义性，故用虚拟继承：

```
// 一个为了使用缺省policy值的类
// 如果我们多次派生自DefaultPolicies，下面的虚拟继承就避免了二义性
class DefaultPolicyArgs : virtual public DefaultPolicies{
};
```

最后，我们只需要写几个模板覆盖掉缺省的policy参数：

```
template <typename Policy>
class Policy1_is : virtual public DefaultPolicies
{
    public:
        typedef Policy P1;           //改写缺省的typedef
};

template <typename Policy>
class Policy2_is : virtual public DefaultPolicies
{
    public:
        typedef Policy P2;           //改写缺省的typedef
};

template <typename Policy>
class Policy3_is : virtual public DefaultPolicies
{
    public:
        typedef Policy P3;           //改写缺省的typedef
};

template <typename Policy>
class Policy4_is : virtual public DefaultPolicies
{
    public:
        typedef Policy P4;           //改写缺省的typedef
};
```

最后，我们把模板BreadSlicer实例化为：

```
BreadSlicer<Policy3_is<CustomPolicy> > bc;
```

这时模板BreadSlicer中的类型Polices被定义为：

```
PolicySelector<Policy3_is<CustomPolicy>,
              DefaultPolicyArgs,
              DefaultPolicyArgs,
              DefaultPolicyArgs>
```

由类模板Discriminator的帮助，我们得到了图16.1所示的类层次。从中可以看出，所有的模板实参都是基类，而它们有共同的虚基类DefaultPolicies，正是这个共同的虚基类定义了P1, P2, P3和P4的缺省类型；不过，其中一个派生类Policy3_Is<>重定义了P3。根据优势规则，重定义的类型隐藏了基类中的定义，这里没有二义性问题。

在模板BreadSlicer中，我们可以使用诸如Policies::P3等限定名称来引用这4个policy，例如：

```
template <... >
class BreadSlicer
{
    ...
    public:
        void print(){
            Policies::P3::doPrint();
        }
    ...
};
```

16.2 空基类优化

C++类常常为“空”，这就意味着在运行期其内部表示不耗费任何内存。这常见于只包含类型成员、非虚成员函数和静态数据成员类，而非静态数据成员、虚函数和虚基类则的确在运行期耗费内存。

即使是空类，其大小也不会是0。在某些对于对齐要求更严格系统上也会有差异。

16.2.1 布局原则

C++的设计者们不允许类的大小为0，其原因很多。比如由它们构成的数组，其大小必然也是0，这会导致指针运算中普遍使用的性质失效。

虽然不能存在“0大小”的类，但C++标准规定，当空类作为基类时，只要不会与同一类型的另一个对象或子对象分配在同一地址，就不需要为其分配任何空间。我们通过实例来看看这个所谓的空基类优化（empty base class optimization, EBCO）技术：

```
// inherit/ebco1.cpp
#include <iostream>

class Empty
{
    typedef int Int;           // typedef 成员并不会使类成为非空
};

class EmptyToo : public EmptyToo
{
};

class EmptyThree : public EmptyToo
{
};

int main()
{
    std::cout << "sizeof(Empty) :           " << sizeof(Empty) << '\n';
    std::cout << "sizeof(EmptyToo) :        " << sizeof(EmptyToo) << '\n';
    std::cout << "sizeof(EmptyThree) :      " << sizeof(EmptyThree) << '\n';
}
```

如果编译器支持空基类优化，上述程序所有的输出结果相同，但均不为0（见图16.2）。也就是说，在类EmptyToo中的类Empty没有分配空间。注意，带有优化空基类的空类（没有其他基类），其大小亦为0；这也是类EmptyThree能够与类Empty具有相同大小的原因所在。然而，在不支持EBCO的编译器上，结果就大相径庭（见图16.3）。

想想在空基类优化下，下列的结果如何？

```
// inherit/ebco2.cpp
#include <iostream>
class Empty
{
    typedef int Int;           // typedef 成员并没有使一个类变成非空
};

class EmptyToo : public Empty
{
};

class NonEmpty : public Empty, public EmptyToo
{
};

int main()
{
    std::cout << "sizeof(Empty) :           " << sizeof(Empty) << '\n';
    std::cout << "sizeof(EmptyToo) :        " << sizeof(EmptyToo) << '\n';
    std::cout << "sizeof(NonEmpty) :        " << sizeof(NonEmpty) << '\n';
}
```

也许你会大吃一惊，类NonEmpty并非真正的“空”类，但的确它和它的基类都没有任何成员。不过，NonEmpty的基类Empty和EmptyToo不能分配到同一地址空间，否则EmptyToo的基类Empty会和NonEmpty的基类Empty撞在同一地址空间上。换句话说，两个相同类型的子对象偏移量相同，这是C++对象布局规则不允许的。有人可能会认为可以把两个Empty对象分别放在偏移0和1字节处，但整个对象的大小也不能仅为1.因为在一个包含两个NonEmpty的数组中，第一个元素和第二个元素的Empty子对象也不能撞在同一地址空间（见图16.4）。

对空基类优化进行限制的根本原因在于，我们需要能比较两个指针是否指向同一对象，由于指针几乎总是用地址作内部表示，所以我们必须保证两个不同的地址（即两个不同的指针值）对应两个不同的对象。虽然这种约束看起来并不非常重要，但是在实际应用中的许多类都是继承自一组定义公共typedefs的基类，当这些类作为子对象出现在同一对象中时，问题就凸现出来了，此时优化应该被禁止。

16.2.2 成员作基类

书中介绍了将成员作基类的技术。但对于数据成员，则不存在类似空基类优化的技术，否则遇到指向成员的指针时就会出问题。

将成员变量实现为（私有）基类的形式，在模板中考虑这个问题特别有意义，因为模板参数常常可能就是空类（虽然我们不可以依赖这个规则）。

16.3 奇特的递归模板模式（Curiously Recurring Template Pattern, CRTP）

奇特的递归模板模式（Curiously Recurring Template Pattern, CRTP）这个奇特的名字代表了类实现技术中一种通用的模式，即派生类将本身作为模板参数传递给基类。最简单的情形如下：

```
template <typename Derived>
class CuriousBase
{
    ....
};

class Curious : public CuriousBase<Curious>           // 普通派生类
{
    ....
};
```

在第一个实例中，CRTP有一个非依赖型基类：类Curious不是模板，因此免于与依赖型基类的名字可见性问题纠缠。不过，这并非CRTP的本质特征，请看：

```
template <typename Derived>
class CuriousBase
{
    ....
};

template <typename T>
class CuriousTemplate : public CuriousBase<CuriousTemplate<T> >           // 派生类也是模板
{
    ...
};
```

从这个示例出发，不难再举出使用模板的模板参数的方式：

```
template <template<typename> class Derived>
class MoreCuriousBase
{
    ....
};

template <typename T>
class MoreCurious : public MoreCuriousBase<MoreCurious>
{
    ....
};
```

CRTP的一个简单应用是记录某个类的对象构造的总个数。数对象个数很简单，只需要引入一个整数类型的静态数据成员，分别在构造函数和析构函数中进行递增和递减操作。不过，要在每个类里都这么写就很繁琐了。有了CRTP，我们可以先写一个模板：

```
// inherit/objectcounter.hpp
#include <stddef.h>

template <typename CountedType>
class ObjectCounter
{
    private:
        static size_t cout;           // 存在对象的个数

    protected:
        //缺省构造函数
        ObjectCounter(){
            ++ObjectCounter<CountedType>::count;
        }
        // 拷贝构造函数
        ObjectCounter(ObjectCounter<CountedType> const&){
            ++ObjectCounter<CountedType>::count;
        }
        // 析构函数
        ~ObjectCounter(){
            --ObjectCounter<CountedType>::count;
        }

    public:
        // 返回存在对象的个数:
        static size_t live(){
            return ObjectCounter<CountedType>::count;
        }
};

// 用0来初始化count
template <typename CountedType>
size_t ObjectCounter<CountedType>::count = 0;
```

如果想要数某个类的对象存在的个数，只需让该类从模板ObjectCounter派生即可。以一个字符串类为例：

```
//inherit/testcounter.cpp
#include "objectcounter.hpp"
#include <iostream>

template <typename CharT>
class MyString : public ObjectCounter<MyString<CharT> >
{
    ....
};

int main()
{
    MyString<char> s1, s2;
    MyString<wchar_t> ws;

    std::cout << "number of MyString<char> : " << MyString<char>::live() << std::endl;
    std::cout << "number of MyString<wchar_t> : " << MyString<wchar_t>::live() << std::endl;
}
```

一般地，CRTP适用于仅能用作成员函数的接口（如构造函数、析构函数和小标运算符operator[]等）的实现提取出来。

16.4 参数化虚拟性

C++允许通过模板直接参数化3种实体：类型、常数（nontype）和模板。同时，模板还能间接参数化其他属性，比如成员函数的虚拟性。

```
// inherit/virtual.cpp
#include <iostream>
class NotVirtual
{
};

class Virtual
{
    public:
        virtual void foo(){
        }
};

template <typename VBase>
class Base : private VBase
{
    public:
        // foo()的虚拟性依赖于它在基类VBase（如果存在基类的话）中声明
        void foo(){
            std::cout << "Base::foo() " << '\n';
        }
};

template <typename V>
class Derived : public Base<V>
{
    public:
        void foo(){
            std::cout << "Derived::foo() " << '\n';
        }
};

int main()
{
    Base<NotVirtual>* p1 = new Derived<NotVirtual>;
    p1->foo();           // 调用Base::foo()


    Base<Virtual>* p2 = new Derived<Virtual>;
    p2->foo();           // 调用Derived::foo()
}
```

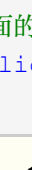
分类: [C++ Template](#)


好文要顶

关注我

收藏该文





 小天_Y
关注 - 63
粉丝 - 96

1

0

 推荐

 反对

« 上一篇: [\[转\]2015有得有悟,2016笨鸟起飞](#)

» 下一篇: [C++ template —— template metaprogram \(九\)](#)