

C++ template —— 类型区分 (十一)

前面的博文介绍了模板的基础，深入模板特性，模板和设计的一些内容。从这篇开始，我们介绍一些高级模板设计，开发某些相对较小、并且互相独立的功能，而且对于这些简单功能而言，模板是最好的实现方法：


- (1) 一个用于类型区分的框架；
- (2) 智能指针
- (3) tuple
- (4) 仿函数

第19章 类型区分

本章主要介绍用模板实现对类型的辨识，判断其是内建类型、指针类型、class类型或者其他类型中的哪一种。

19.1 辨别基本类型

缺省情况下，我们一方面假定一个类型不是一个基本类型，另一方面我们为所有的基本类型都特化给模板：

// types/type1.hpp

// 基本模板：一般情况下T不是基本类型

template <typename T>

class IsFundat

{

public:

enum { Yes = 0, No = 1 };

};

// 用于特化基本类型的宏

#define MK_FUNDA_TYPE(T) \

template<> class IsFundat<T> { \

public: \

enum { Yes = 1, No = 0 } ; \

};

MK_FUNDA_TYPE(void)

MK_FUNDA_TYPE(bool)

MK_FUNDA_TYPE(char)

MK_FUNDA_TYPE(signed char)

MK_FUNDA_TYPE(unsigned char)

MK_FUNDA_TYPE(wchar_t)

MK_FUNDA_TYPE(signed short)

MK_FUNDA_TYPE(unsigned short)

MK_FUNDA_TYPE(signed int)

MK_FUNDA_TYPE(unsigned int)

MK_FUNDA_TYPE(signed long)

MK_FUNDA_TYPE(unsigned long)

#if LONG_LONG_EXISTS

MK_FUNDA_TYPE(signed long long)

MK_FUNDA_TYPE(unsigned long long)


#endif // LONG_LONG_EXISTS

MK_FUNDA_TYPE(float)

MK_FUNDA_TYPE(double)


MK_FUNDA_TYPE(long double)

#undef MK_FUNDA_TYPE



19.2 辨别组合类型

组合类型是指一些构造自其他类型的类型。简单的组合类型包括：普通类型、指针类型、引用类型和数组类型。它们都是构造自单一的基本类型。同时，class类型和函数类型也是组合类型，但这些组合类型通常会涉及到多种类型（例如参数或者成员的类型）。在此，我们先考虑简单的组合类型；另外，我们还将使用局部特化对简单的组合类型进行区分。接下来，我们将定义一个trait类，用于描述简单的组合类型；而class类型和枚举类型将在最后考虑。

// types/type2.hpp

template <typename T>

class CompoundT // 基本模板

{

public:

enum { IsPtrT = 0, IsRefT = 0, IsArrayT = 0,


IsFuncT = 0, IsPtrMemT = 0 };

typedef T BaseT;

typedef T BottomT;


typedef CompoundT<void> ClassT;

};



成员类型BaseT指的是：用于构造模板参数类型T的（直接）类型；而BottomT指的是最终去除指针、引用和数组之后的、用于构造T的原始类型。例如，如果T是int**，那么BaseT将是int*，而BottomT将会是int类型。对于成员指针类型，BaseT将会是成员的类型，而ClassT将会是成员所属的类的类型。例如，如果T是一个类型为int(X::*)(X)的成员函数指针，那么BaseT将会是函数类型int()，而ClassT的类型则为X。如果T不是成员指针类型，那么ClassT将会是CompoundT<void>（这个选择并不是必须的，也可以使用一个noclass来作为ClassT）。

其中，针对指针和引用的局部特化是相当直接的：

// types/type3.hpp

template <typename T>

class CompoundT<T*>

{

public:

enum { IsPtrT = 0, IsRefT = 1, IsArrayT = 0,

IsFuncT = 0, IsPtrMemT = 0 };

typedef T BaseT;

typedef typename CompoundT<T*>::BottomT BottomT;

typedef CompoundT<void> ClassT;

};

template <typename T>

class CompoundT<T*>

{

public:

enum { IsPtrT = 1, IsRefT = 0, IsArrayT = 0,

IsFuncT = 0, IsPtrMemT = 0 };

typedef T BaseT;

typedef typename CompoundT<T*>::BottomT BottomT;

typedef CompoundT<void> ClassT;

};



对于成员指针和数组，我们可能会使用同样的技术来处理。但是，在下面的代码中我们将发现，与基本模板相比，这些局部特化将会涉及到更多的模板参数：

// types/type4.hpp

#include <stddef.h>

template<typename T, size_t N>

class CompoundT<T[N]>

{

public:

enum { IsPtrT = 0, IsRefT = 0, IsArrayT = 1,

IsFuncT = 0, IsPtrMemT = 0 };

typedef T BaseT;

typedef typename CompoundT<T*>::BottomT BottomT;

typedef CompoundT<void> ClassT;

};

template<typename T>

class CompoundT<T[]>

{

public:

enum { IsPtrT = 0, IsRefT = 0, IsArrayT = 1,

IsFuncT = 0, IsPtrMemT = 0 };

typedef T BaseT;

typedef typename CompoundT<T*>::BottomT BottomT;

typedef CompoundT<void> ClassT;

};

template<typename T>

class CompoundT<T C::*>

{

public:

enum { IsPtrT = 0, IsRefT = 0, IsArrayT = 0,

IsFuncT = 0, IsPtrMemT = 1 };

typedef T BaseT;

typedef typename CompoundT<T*>::BottomT BottomT;

typedef C ClassT;

};




19.3 辨别函数类型

书中提供了两种辨识函数类型的方法，这里只介绍第2种：

method2：使用SFINAE原则的解决方案：

一个重载函数模板的后面可以是一些显式模板实参；而且对于某些重载函数类型而言，该实参是有效的，但对于其他的重载函数类型，该实参则可能是无效的。实际上，后面使用重载解析对枚举类型进行辨别的技术也使用了这种方法。SFINAE原则在这里的主要用处是：（1）找到一种构造，该构造对函数类型是无效的，但是对于其他类型都是有效的；或者完全相反。由于前面我们已经能够辨别出几种类型了，所以我们在此可以不再考虑这些（已经可以辨别的）类型。（2）因此，针对上面这种要求，数组类型就是一种有效的构造：因为数组的元素是不能为void值、引用或者函数的。故而可以编写如下代码：

template <typename T>

class IsFunctionT

{

private:

typedef char One;

typedef struct { char a[2]; } Two;

template <typename U> static One test (...);


template <typename U> static Two test (U (*)[1]); // 不理解，下面的IsFunctionT<T*>::test<T*>(0)怎么匹配

public:


enum { Yes = sizeof(IsFunctionT<T*>::test<T*>(0)) == 1 };

enum { No = !Yes };

};



借助于上面这个模板定义，只有对于那些不能作为数组元素类型的类型，IsFunctionT::Yes才是非零值（即为1）。另外，我们应当知道该方法也有一个不足之处：并非只有函数类型不能作为数组元素类型，引用类型和void类型同样也不能作为数组元素类型。（3）幸运的是，我们可以通过为引用类型提供局部特化，以及为void类型提供显式特化，来解决这个不足：

template <typename T>

class IsFunctionT<T*>

{

public:

enum { No = 0 };

enum { Yes = !Yes };

};

template <>

class IsFunctionT<void>

{

public:

enum { Yes = 0 };

enum { No = !Yes };

};

template <>

class IsFunctionT<void const>


{

public:

enum { Yes = 0 };

enum { No = !Yes };

};



至此，我们可以重新改写基本的CompoundT模板如下：

// types/type6.hpp

template <typename T>

class IsFunctionT

{

private:

typedef char One;

typedef struct { char a[2]; } Two;

template <typename U> static One test (...);

template <typename U> static Two test (U (*)[1]);

public:

enum { Yes = sizeof(IsFunctionT<T*>::test<T*>(0)) == 1 };

enum { No = !Yes };

};

template <typename T>

class IsFunctionT<T*>

{

public:

enum { Yes = 0 };

enum { No = !Yes };

};

template <>

class IsFunctionT<void>

{

public:

enum { Yes = 0 };

enum { No = !Yes };

};

template <>

class IsFunctionT<void const>

{

public:

enum { Yes = 0 };

enum { No = !Yes };

};

// 对于void volatile 和 void const volatile类型也是一样的

...

template <typename T>

class CompoundT // 基本模板

{

public:

enum { IsPtrT = 0, IsRefT = 0, IsArrayT = 0,

IsFuncT = IsFunctionT<T*>::Yes, IsPtrMemT = 0 };

typedef T BaseT;

typedef T BottomT;

typedef CompoundT<void> ClassT;

};



19.4 运用重载解析辨别枚举类型


重载解析是一个过程，它会根据函数参数的类型，在多个同名函数中选择一个合适的函数。接下来我们将看到，即使没有进行实际的函数调用，我们也可以声明一个返回类型为T的函数，然后通过调用这个函数来创建一个T。由于处于sizeof表达式内部，因此该函数实际上并不需要具有函数定义。事实上，更加巧妙的是：对于一个class类型T，重载解析是有可能选择一个针对整型的enum_check()声明的，但前提是该class必须定义一个到整型的自定义转型（有时也称为UDC）函数。到此，问题已经解决了。因为我们在ConsumeUDC模板中已经强制定义了一个到T的自定义转型，该转型运算符同时也为sizeof运算符生成了一个类型为T的实参。下面我们详细分析下：

- (1) 最开始的实参是一个临时的ConsumeUDC<T>对象；
- (2) 如果T是一个基本整型，那么将会借助于（ConsumeUDC的）转型运算符来创建一个enum_check()的匹配，该enum_check()以T为实参；
- (3) 如果T是一个枚举类型，那么将会借助于（ConsumeUDC的）转型运算符，先把类型转化为T，然后调用（从枚举类型到整型）的类型提升，从而能够匹配一个接收转型参数的enum_check()函数（通常而言是enum_check(int)）；
- (4) 如果T是一个class类型，而且已经为该class自定义了一个到整型的转型运算符，那么这个转型运算符将不会被考虑。因为对于以匹配为目的的自定义转型而言，最多只能调用一次；而且在前面的已经使用了一个从ConsumeUDC<T>到T的自定义转型，所以也就不允许再次调用自定义转型。也就是说，对enum_check()函数而言，class类型最终还是未能转型为整型。
- (5) 如果最终还是不能让类型T于整型互相匹配，那么将会选择enum_check()函数的省略号版本。

最后，由于我们这里只是为了辨别枚举类型，而不是基本类型或者指针类型，所有我们使用了前面已经开放的IsFundat和CompoundT类型，从而能够排除这些令IsEnumT<T*>::Yes成为非零的其他类型，最后使得只有枚举类型的IsEnumT::Yes才等于1。

19.5 辨别class类型

使用排除原理：如果一个类型不是一个基本类型，也不是枚举类型和组合类型，那么该类型就只能是class类型。

template <typename T>

class IsClassT

{

public:

enum {

Yes = IsFundat<T*>::No &&

IsEnumT<T*>::No &&

!CompoundT<T*>::IsPtrT &&

!CompoundT<T*>::IsArrayT &&


!CompoundT<T*>::IsPtrMemT &&

!CompoundT<T*>::IsFuncT

};

enum { No = !Yes };

};



分类: C++ Template

好文要顶

关注我

收藏该文

