如何在 C++中获得完整的类型名称

木头云 2014-05-16 00:37:02 17644 收藏 22
分类专栏： C/C++ 文章标签： C++11 泛型 lambda 类型
版权

C/C++
专栏收录该内容
20 篇文章 0 订阅
订阅专栏
Wrote by mutouyun . (http://darkc.at/cxx-get-the-name-of-the-given-type/)

地球人都知道 C++里有一个 typeid 操作符可以用来获取一个类型/表达式的名称：
std::cout << typeid(int).name() << std::endl;
但是这个 name()的返回值是取决于编译器的，在 vc 和 gcc 中打印出来的结果如下：

int // vc
i // gcc
一个稍微长一点的类型名称，比如：

class Foo {};
std::cout << typeid(Foo*[10]).name() << std::endl;

打出来是这个效果：
class Foo * [10] // vc
A10_P3Foo // gcc
（话说 gcc 您的返回结果真是。。）
当然了，想在 gcc 里得到和微软差不多显示效果的方法也是有的，那就是使用 __cxa_demangle：

char* name = abi::__cxa_demangle(typeid(Foo*[10]).name(), nullptr, nullptr, nullptr);
std::cout << name << std::endl;
free(name);
显示效果：

Foo* [10]
先不说不同编译器下的适配问题，来看看下面这个会打印出啥：

// vc
std::cout << typeid(const int&).name() << std::endl;

// gcc

```
char* name = abi::__cxa_demangle(typeid(const int&).name(), nullptr, nullptr, nullptr);
std::cout << name << std::endl;
free(name);
```

显示效果：

```
int // vc
int // gcc
```

可爱的 cv 限定符和引用都被丢掉了=.=

如果直接在 typeid 的结果上加上被丢弃的信息，对于一些类型而言（如函数指针引用）得到的将不是一个正确的类型名称。

想要获得一个类型的完整名称，并且获得的名称必须要是一个正确的类型名称，应该怎样做呢？

一、如何检查 C++中的类型

我们需要一个泛型类，用特化/偏特化机制静态检查出 C++中的各种类型，并且不能忽略掉类型限定符（type-specifiers）和各种声明符（declarators）。

先来考虑一个最简单的类模板：

```
template <typename T>
struct check
{
	// ...
};
```

假如在它的基础上特化，需要写多少个版本呢？我们可以稍微实现下试试：

```
template <typename T> struct check<T &>;
template <typename T> struct check<T const &>;
template <typename T> struct check<T volatile &>;
template <typename T> struct check<T const volatile &>;

template <typename T> struct check<T &&>;
template <typename T> struct check<T const &&>;
template <typename T> struct check<T volatile &&>;
template <typename T> struct check<T const volatile &&>;

template <typename T> struct check<T *>;
template <typename T> struct check<T const *>;
template <typename T> struct check<T volatile *>;
template <typename T> struct check<T const volatile *>;
template <typename T> struct check<T * const>;
template <typename T> struct check<T * volatile>;
template <typename T> struct check<T * const volatile>;
```

```
template <typename T> struct check<T []>;
template <typename T> struct check<T const []>;
template <typename T> struct check<T volatile []>;
template <typename T> struct check<T const volatile []>;
template <typename T, size_t N> struct check<T [N]>;
template <typename T, size_t N> struct check<T const [N]>;
template <typename T, size_t N> struct check<T volatile [N]>;
template <typename T, size_t N> struct check<T const volatile [N]>;

// ......
```

这还远远没有完。有同学可能会说了，我们不是有伟大的宏嘛，这些东西都像是一个模子刻出来的，弄一个宏批量生成下不就完了。

实际上当我们真的信心满满的动手去写这些宏的时候，才发现适配上的细微差别会让宏写得非常痛苦（比如&和*的差别，[]和[N]的差别，还有函数类型、函数指针、函数指针引用、函数指针数组、类成员指针、……）。当我们一一罗列出需要特化的细节时，不由得感叹 C++类型系统的复杂和纠结。

但是上面的理由并不是这个思路的致命伤。
不可行的地方在于：我们可以写一个多维指针，或多维数组，类型是可以嵌套的。总不可能为每一个维度都特化一个模板吧。

不过正由于类型其实是嵌套的，我们可以用模板元编程的基本思路来搞定这个问题：

```
template <typename T> struct check<T const> : check<T>;
template <typename T> struct check<T volatile> : check<T>;
template <typename T> struct check<T const volatile> : check<T>;

template <typename T> struct check<T & > : check<T>;
template <typename T> struct check<T &&> : check<T>;
template <typename T> struct check<T * > : check<T>;

// ......
```

一个简单的继承，就让特化变得 simple 很多。因为当我们萃取出一个类型，比如 T *，之后的 T 其实是携带上了除*之外所有其他类型信息的一个类型。那么把这个 T 再重复投入 check 中，就会继续萃取它的下一个类型特征。

可以先用指针、引用的萃取来看看效果：

```
#include <iostream>
```

```
template <typename T>
struct check
{
	check(void)  { std::cout << typeid(T).name(); }
	~check(void) { std::cout << std::endl; }
};

#define CHECK_TYPE__(OPT) \
	template <typename T> \
	struct check<T OPT> : check<T> \
	{ \
		check(void) { std::cout << " "#OPT; } \
	};

CHECK_TYPE__(const)
CHECK_TYPE__(volatile)
CHECK_TYPE__(const volatile)
CHECK_TYPE__(&)
CHECK_TYPE__(&&)
CHECK_TYPE__(*)

int main(void)
{
	check<const volatile void * const*&>();
	system("pause");
	return 0;
}
```

输出结果（vc）：

```
void const volatile * const * &
```

很漂亮，是不是？当然，在 gcc 里这样输出，void 会变成 v，所以 gcc 下面要这样写 check 模板：

```
template <typename T>
struct check
{
	check(void)
	{
		char* real_name = abi::__cxa_demangle(typeid(T).name(), nullptr, nullptr, nullptr);
		std::cout << real_name;
		free(real_name);
	}
```

```
    ~check(void) { std::cout << std::endl; }
};
```

二、保存和输出字符串

我们可以简单的这样修改 check 让它同时支持 vc 和 gcc：

```
template <typename T>
struct check
{
    check(void)
    {
#   if defined(__GNUC__)
        char* real_name = abi::__cxa_demangle(typeid(T).name(), nullptr, nullptr, nullptr);
        std::cout << real_name;
        free(real_name);
#   else
        std::cout << typeid(T).name();
#   endif
    }
    ~check(void) { std::cout << std::endl; }
};
```

但是到目前为止，check 的输出结果都是无法保存的。比较好的方式是可以像 typeid(T).name() 一样返回一个字符串。这就要求 check 能够把结果保存在一个 std::string 对象里。

当然了，我们可以直接给 check 一个“std::string& out”类型的构造函数，但是这样会把输出的状态管理、字符的打印逻辑等等都揉在一起。因此，比较好的设计方法是实现一个 output 类，负责输出和维护状态。我们到后面就会慢慢感觉到这样做的好处在哪里。

output 类的实现可以是这样：

```
class output
{
    bool is_compact_ = true;

    template <typename T>
    bool check_empty(const T&) { return false; }
    bool check_empty(const char* val)
    {
        return (!val) || (val[0] == 0);
    }

    template <typename T>
    void out(const T& val)
    {
        if (check_empty(val)) return;
        if (!is_compact_) sr_ += " ";
```

```
            using ss_t = std::ostringstream;
            sr_ += static_cast<ss_t&>(ss_t() << val).str();
            is_compact_ = false;
        }

        std::string& sr_;

public:
        output(std::string& sr) : sr_(sr) {}

        output& operator()(void) { return (*this); }

        template <typename T1, typename... T>
        output& operator()(const T1& val, const T&... args)
        {
            out(val);
            return operator()(args...);
        }

        output& compact(void)
        {
            is_compact_ = true;
            return (*this);
        }
};
```

这个小巧的 output 类负责自动管理输出状态（是否增加空格）和输出的类型转换（使用 std::ostringstream）。
上面的实现里有两个比较有意思的地方。
一是 operator()的做法，采用了变参模板。这种做法让我们可以这样用 output：

```
output out(str);
out("Hello", "World", 123, "!");
```

这种写法比 cout 的流操作符舒服多了。
二是 operator()和 compact 的返回值。当然，这里可以直接使用 void，但是这会造成一些限制。
比如说，我们想在使用 operator()之后马上 compact 呢？若让函数返回自身对象的引用，就可以让 output 用起来非常顺手：

```
output out(str);
out.compact()("Hello", "World", 123, "!").compact()("?");
```

check 的定义和 CHECK_TYPE__宏只需要略作修改就可以使用 output 类：

```
template <typename T>
struct check
{
	output out_;
	check(const output& out) : out_(out)
	{
#	if defined(__GNUC__)
		char* real_name = abi::__cxa_demangle(typeid(T).name(), nullptr, nullptr, nullptr);
		out_(real_name);
		free(real_name);
#	else
		out_(typeid(T).name());
#	endif
	}
};

#define CHECK_TYPE__(OPT) \
	template <typename T> \
	struct check<T OPT> : check<T> \
	{ \
		using base_t = check<T>; \
		using base_t::out_; \
		check(const output& out) : base_t(out) { out_(#OPT); } \
	};
```

为了让外部的使用依旧简洁，实现一个外敷函数模板是很自然的事情：

```
template <typename T>
inline std::string check_type(void)
{
	std::string str;
	check<T> { str };
	return std::move(str);
}

int main(void)
{
	std::cout << check_type<const volatile void * const*&>() << std::endl;
	system("pause");
	return 0;
}
```

如果我们想实现表达式的类型输出，使用 decltype 包裹一下就行了。

不知道看到这里的朋友有没有注意到，check 在 gcc 下的输出可能会出现问题。原因是 abi::__cxa_demangle 并不能保证永远返回一个有效的字符串。
我们来看看这个函数的返回值说明：

“Returns: A pointer to the start of the NUL-terminated demangled name, or NULL if the demangling fails. The caller is responsible for deallocating this memory using free.”

所以说比较好的做法应该是在 abi::__cxa_demangle 返回空的时候，直接使用 typeid(T).name() 的结果。
一种健壮的写法可以像这样：

```
template <typename T>
struct check
{
	output out_;
	check(const output& out) : out_(out)
	{
#	if defined(__GNUC__)
		const char* typeid_name = typeid(T).name();
		auto deleter = [](char* p)
		{
			if (p) free(p);
		};
		std::unique_ptr<char, decltype(deleter)> real_name
		{
			abi::__cxa_demangle(typeid_name, nullptr, nullptr, nullptr), deleter
		};
		out_(real_name ? real_name.get() : typeid_name);
#	else
		out_(typeid(T).name());
#	endif
	}
};
```

上面我们通过使用 std::unique_ptr 配合 lambda 的自定义 deleter，实现了一个简单的 Scope Guard 机制，来保证当 abi::__cxa_demangle 返回的非 NULL 指针一定会被 free 掉。

三、输出有效的类型定义
3.1 一些准备工作
上面的特化解决了 cv 限定符、引用和指针，甚至对于未特化的数组、类成员指针等都有还不错的显示效果，不过却无法保证输出的类型名称一定是一个有效的类型定义。比如说：

```
check_type<int(*)[]>(); // int [] *
```

原因是因为这个类型是一个指针，指向一个 int[]，所以会先匹配到指针的特化，因此*就被写到了最后面。
对于数组、函数等类型来说，若它们处在一个复合类型（compound types）中“子类型”的位置上，它们就需要用括号把它们的“父类型”给括起来。
因此我们还需要预先完成下面这些工作：
1. 如何判断数组、函数等类型的特化处于 check 继承链中“被继承”（也就是某个类的基类）的位置上
2. 圆括号()、方括号[]，以及函数参数列表的输出逻辑
上面的第 1 点，可以利用模板偏特化这种静态的判断来解决。比如说，给 check 添加一个默认的 bool 模板参数：

```
template <typename T, bool IsBase = false>
struct check
{
	// ...
};

#define CHECK_TYPE__(OPT) \
	template <typename T, bool IsBase> \
	struct check<T OPT, IsBase> : check<T, true> \
	{ \
		using base_t = check<T, true>; \
		using base_t::out_; \
		check(const output& out) : base_t(out) { out_(#OPT); } \
	};
```

这个小小的修改就可以让 check 在继承的时候把父-子信息传递下去。

接下来先考虑圆括号的输出逻辑。我们可以构建一个 bracket 类，在编译期帮我们自动处理圆括号：

```
// ()

template <bool>
struct bracket
{
	output& out_;

	bracket(output& out, const char* = nullptr) : out_(out)
	{ out_("(").compact(); }

	~bracket(void)
	{ out_.compact()(")"); }
};
```

```
template <>
struct bracket<false>
{
	bracket(output& out, const char* str = nullptr)
	{ out(str); }
};
```

在 bracket 里，不仅实现了圆括号的输出，其实还实现了一个编译期 if 的小功能。当不输出圆括号时，我们可以给 bracket 指定一个其它的输出内容。
当然，不实现 bracket，直接在 check 的类型特化里处理括号逻辑也可以，但是这样的话逻辑就被某个 check 特化绑死了。我们可以看到 bracket 的逻辑被剥离出来以后，后面所有需要输出圆括号的部分都可以直接复用这个功能。

然后是[]的输出逻辑。考虑到对于[N]类型的数组，还需要把 N 的具体数值输出来，因此输出逻辑可以这样写：

```
// [N]

template <size_t N = 0>
struct bound
{
	output& out_;

	bound(output& out) : out_(out) {}
	~bound(void)
	{
		if (N == 0) out_("[]");
		else		out_("[").compact()
					( N ).compact()
					("]");
	}
};
```

输出逻辑需要写在 bound 类的析构，而不是构造里。原因是对于一个数组类型，[N]总是写在最后面的。
这里在输出的时候直接使用了运行时的 if-else，而没有再用特化来处理。是因为当 N 是一个编译期数值时，对于现代的编译器来说“if (N == 0) ; else ;”语句会被优化掉，只生成确定逻辑的汇编码。

最后，是函数参数的输出逻辑。函数参数列表需要使用变参模板适配，用编译期递归的元编程手法输出参数，最后在两头加上括号。
我们可以先写出递归的结束条件：

```
template <bool, typename... P>
struct parameter;
```

```
template <bool IsStart>
struct parameter<IsStart>
{
	output& out_;

	parameter(output& out) : out_(out) {}
	~parameter(void)
	{ bracket<IsStart> { out_ }; }
};
```

输出逻辑写在析构里的理由，和 bound 一致。结束条件是显然的：当参数包为空时，parameter 将只输出一对括号。
注意到模板的 bool 类型参数，让我们在使用的时候需要这样写：

```
parameter<true, P...> parameter_;
```

这是因为 bool 模板参数混在变参里，指定默认值也是没办法省略 true 的。
稍微有点复杂的是参数列表的输出。一个简单的写法是这样：

```
template <bool IsStart, typename P1, typename... P>
struct parameter<IsStart, P1, P...>
{
	output& out_;

	parameter(output& out) : out_(out) {}
	~parameter(void)
	{
		bracket<IsStart> bk { out_, "," }; (void)bk;
		check<P1> { out_ };
		parameter<false, P...> { out_.compact() };
	}
};
```

parameter 在析构的时候，析构函数的 scope 就是 bracket 的影响范围，后面的其它显示内容，都应该被包括在 bracket 之内，因此 bracket 需要显式定义临时变量 bk；
check 的调用理由很简单，因为我们需要显示出每个参数的具体类型；
最下面是 parameter 的递归调用。在把 out_丢进去之前，我们需要思考下具体的显示效果。是希望打印出(P1, P2, P3)呢，还是(P1 , P2 , P3)？
在这里我们选择了逗号之前没有空格的第一个版本，因此给 parameter 传递的是 out_.compact()。

对 parameter 的代码来说，看起来不明显的就是 bracket 的作用域了，check 和 parameter 的调用其实是被 bracket 包围住的。为了强调 bracket 的作用范围，同时规避掉莫名其妙的“(void)bk;”手法，我们可以使用 lambda 表达式来凸显逻辑：

```
template <bool IsStart, typename P1, typename... P>
struct parameter<IsStart, P1, P...>
{
	output& out_;

	parameter(output& out) : out_(out) {}
	~parameter(void)
	{
		[this](bracket<IsStart>&&)
		{
			check<P1> { out_ };
			parameter<false, P...> { out_.compact() };
		} (bracket<IsStart> { out_, "," });
	}
};
```

这样 bracket 的作用域一目了然，并且和 check、parameter 的定义方式保持一致，同时也更容易看出来 out_.compact()的意图。

3.2 数组（Arrays）的处理
好了，有了上面的这些准备工作，写一个 check 的 T[]特化是很简单的：

```
template <typename T, bool IsBase>
struct check<T[], IsBase> : check<T, true>
{
	using base_t = check<T, true>;
	using base_t::out_;

	bound<>		bound_;
	bracket<IsBase> bracket_;

	check(const output& out) : base_t(out)
		, bound_	(out_)
		, bracket_(out_)
	{}
};
```

这时对于不指定数组长度的[]类型，输出结果如下：

```
check_type<int(*)[]>(); // int (*) []
```

当我们开始兴致勃勃的接着追加[N]的模板特化之前，需要先检查下 cv 的检查机制是否运作良好：

check_type<const int[]>();

尝试编译时，gcc 会给我们吐出一堆类似这样的 compile error：

```
error: ambiguous class template instantiation for 'struct check<const int [], false>'
        check<T> { str };
        ^
```

检查了出错信息后，我们会惊讶的发现对于 const int[]类型，竟然可以同时匹配 T const 和 T[]。
这是因为按照 C++标准 ISO/IEC-14882:2011，3.9.3 CV-qualifiers，第 5 款：

“Cv-qualifiers applied to an array type attach to the underlying element type, so the notation “cv T,” where T is an array type, refers to an array whose elements are so-qualified. Such array types can be said to be more (or less) cv-qualified than other types based on the cv-qualification of the underlying element types.”

可能描述有点晦涩，不过没关系，在 8.3.4 Arrays 的第 1 款最下面还有一行批注如下：

“[Note: An “array of N cv-qualifier-seq T” has cv-qualified type; see 3.9.3. —end note]”

意思就是对于 const int[]来说，const 不仅属于数组里面的 int 元素所有，同时还会作用到数组本身上。
所以说，我们不得不多做点工作，把 cv 限定符也特化进来：

```
#define CHECK_TYPE_ARRAY__(CV_OPT) \
    template <typename T, bool IsBase> \
    struct check<T CV_OPT [], IsBase> : check<T CV_OPT, true> \
    { \
        using base_t = check<T CV_OPT, true>; \
        using base_t::out_; \
    \
        bound<>             bound_; \
        bracket<IsBase> bracket_; \
    \
        check(const output& out) : base_t(out) \
            , bound_   (out_) \
            , bracket_(out_) \
        {} \
    };

#define CHECK_TYPE_PLACEHOLDER__
CHECK_TYPE_ARRAY__(CHECK_TYPE_PLACEHOLDER__)
CHECK_TYPE_ARRAY__(const)
```

```
CHECK_TYPE_ARRAY__(volatile)
CHECK_TYPE_ARRAY__(const volatile)
```

这样对于加了 cv 属性的数组而言，编译和显示才是正常的。
接下来，考虑[N]，我们需要稍微修改一下上面的 CHECK_TYPE_ARRAY__宏，让它可以同时处理[]和[N]:

```
#define CHECK_TYPE_ARRAY__(CV_OPT, BOUND_OPT, ...) \
    template <typename T, bool IsBase __VA_ARGS__> \
    struct check<T CV_OPT [BOUND_OPT], IsBase> : check<T CV_OPT, true> \
    { \
        using base_t = check<T CV_OPT, true>; \
        using base_t::out_; \
    \
        bound<BOUND_OPT> bound_; \
        bracket<IsBase>    bracket_; \
    \
        check(const output& out) : base_t(out) \
            , bound_    (out_) \
            , bracket_(out_) \
        {} \
    };

#define CHECK_TYPE_ARRAY_CV__(BOUND_OPT, ...) \
    CHECK_TYPE_ARRAY__(, BOUND_OPT, ,##__VA_ARGS__) \
    CHECK_TYPE_ARRAY__(const, BOUND_OPT, ,##__VA_ARGS__) \
    CHECK_TYPE_ARRAY__(volatile, BOUND_OPT, ,##__VA_ARGS__) \
    CHECK_TYPE_ARRAY__(const volatile, BOUND_OPT, ,##__VA_ARGS__)
```

这段代码里稍微用了点“preprocessor”式的技巧。gcc 的__VA_ARGS__处理其实不那么人性化。虽然我们可以通过“,##__VA_ARGS__”，在变参为空时消除掉前面的逗号，但这个机制却只对第一层宏有效。当我们把__VA_ARGS__继续向下传递时，变参为空逗号也不会消失。
因此，我们只有用上面这种略显抽搐的写法来干掉第二层宏里的逗号。这个处理技巧也同样适用于 vc。

然后，实现各种特化模板的时候到了:

```
#define CHECK_TYPE_PLACEHOLDER__
CHECK_TYPE_ARRAY_CV__(CHECK_TYPE_PLACEHOLDER__)
#if defined(__GNUC__)
CHECK_TYPE_ARRAY_CV__(0)
#endif
CHECK_TYPE_ARRAY_CV__(N, size_t N)
```

这里有个有意思的地方是:gcc 里可以定义 0 长数组[0]，也叫“柔性数组”。这玩意在 gcc 里不

会适配到 T[N]或 T[]上，所以要单独考虑。

现在，我们适配上了所有的引用、数组，以及普通指针：

```
check_type<const volatile void *(&)[10]>(); // void const volatile * (&) [10]
check_type<int [1][2][3]>();                // int (([1]) [2]) [3]
```

这里看起来有点不一样的是多维数组的输出结果，每个维度都被括号限定了结合范围。这种用括号明确标明数组每个维度的结合优先级的写法，虽然看起来不那么干脆，不过在 C++中也是合法的。
当然，如果觉得这样不好看，想搞定这个也很简单，稍微改一下 CHECK_TYPE_ARRAY__就可以了：

```
#define CHECK_TYPE_ARRAY__(CV_OPT, BOUND_OPT, ...) \
    template <typename T, bool IsBase __VA_ARGS__> \
    struct check<T CV_OPT [BOUND_OPT], IsBase> : check<T CV_OPT, !std::is_array<T>::value> \
    { \
        using base_t = check<T CV_OPT, !std::is_array<T>::value>; \
        using base_t::out_; \
    \
        bound<BOUND_OPT> bound_; \
        bracket<IsBase>  bracket_; \
    \
        check(const output& out) : base_t(out) \
            , bound_  (out_) \
            , bracket_(out_) \
        {} \
    };
```

这里使用了 std::is_array 来判断下一层类型是否仍旧是数组，如果是的话，则不输出括号。

3.3 函数（Functions）的处理
有了前面准备好的 parameter，实现一个函数的特化处理非常轻松：

```
template <typename T, bool IsBase, typename... P>
struct check<T(P...), IsBase> : check<T, true>
{
    using base_t = check<T, true>;
    using base_t::out_;

    parameter<true, P...> parameter_;
    bracket<IsBase>       bracket_;

    check(const output& out) : base_t(out)
```

```
        , parameter_(out_)
        , bracket_  (out_)
    {}
};
```

这里有一个小注意点：函数和数组一样，处于被继承的位置时需要加括号；parameter 的构造时机应该在 bracket 的前面，这样可以保证它在 bracket 之后被析构，否则参数列表将被添加到错误位置上。
我们可以打印一个变态一点的类型来验证下正确性：

```
std::cout << check_type<char(* (* const)(const int(&)[10]) )[10]>() << std::endl;
// 输出：char (* (* const) (int const (&) [10])) [10]
// 这是一个常函数指针，参数是一个常 int 数组的引用，返回值是一个 char 数组的指针
```

我们可以看到，函数指针已经被正确的处理掉了。这是因为一个函数指针会适配到指针上，之后去掉指针的类型将是一个正常的函数类型。
这里我们没有考虑 stdcall、fastcall 等调用约定的处理，如有需要的话，读者可自行添加。

3.4 类成员指针（Pointers to members）的处理
类成员指针的处理非常简单：

```
template <typename T, bool IsBase, typename C>
struct check<T C::*, IsBase> : check<T, true>
{
    using base_t = check<T, true>;
    using base_t::out_;

    check(const output& out) : base_t(out)
    {
        check<C> { out_ };
        out_.compact()("::*");
    }
};
```

显示效果：

```
class Foo {};
std::cout << check_type<int (Foo::* const)[3]>() << std::endl;
// 输出：int (Foo::* const) [3]
// 这是一个常类成员指针，指向 Foo 里的一个 int[3]成员
```

3.5 类成员函数指针（Pointers to member functions）的处理
其实我们不用做什么特别的处理，通过 T C::*已经可以适配无 cv 限定符的普通类成员函数指针了。只是在 vc 下，提取出来的 T 却无法适配上 T(P...)的特化。

这是因为 vc 中通过 T C::*提取出来的函数类型带上了一个隐藏的 thiscall 调用约定。在 vc 里，我们无法声明或定义一个 thiscall 的普通函数类型，于是 T C::*的特化适配无法完美的达到我们想要的效果。
所以，我们还是需要处理无 cv 限定的类成员函数指针。通过一个和上面 T C::*的特化很像的特化模板，就可以处理掉一般的类成员函数指针：

```
template <typename T, bool IsBase, typename C, typename... P>
struct check<T(C::*)(P...), IsBase> : check<T(P...), true>
{
	using base_t = check<T(P...), true>;
	using base_t::out_;

	check(const output& out) : base_t(out)
	{
		check<C> { out_ };
		out_.compact()("::*");
	}
};
```

下面考虑带 cv 限定符的类成员函数指针。在开始书写后面的代码之前，我们需要先思考一下，cv 限定符在类成员函数指针上的显示位置是哪里？答案当然是在函数的参数表后面。
所以我们必须把 cv 限定符的输出时机放在 T(P...)显示完毕之后。
因此想要正确的输出 cv 限定符，我们必须调整 T(P...)特化的调用时机：

```
// Do output at destruct

struct at_destruct
{
	output&		out_;
	const char* str_;

	at_destruct(output& out, const char* str = nullptr)
		: out_(out)
		, str_(str)
	{}
	~at_destruct(void)
	{ out_(str_); }

	void set_str(const char* str = nullptr)
	{ str_ = str; }
};

#define CHECK_TYPE_MEM_FUNC__(...) \
	template <typename T, bool IsBase, typename C, typename... P> \
```

```
struct check<T(C::*)(P...) __VA_ARGS__, IsBase> \
{ \
    at_destruct cv_; \
    check<T(P...), true> base_; \
    output& out_ = base_.out_; \
\
    check(const output& out) \
        : cv_(base_.out_) \
        , base_(out) \
    { \
        cv_.set_str(#__VA_ARGS__); \
        check<C> { out_ }; \
        out_.compact()("::*"); \
    } \
};
```

CHECK_TYPE_MEM_FUNC__()
CHECK_TYPE_MEM_FUNC__(const)
CHECK_TYPE_MEM_FUNC__(volatile)
CHECK_TYPE_MEM_FUNC__(const volatile)

上面这段代码先定义了一个 at_destruct，用来在析构时执行“输出 cv 限定符”的动作；同时把原本处在基类位置上的 T(P...)特化放在了第二成员的位置上，这样就保证了它将会在 cv_之后才被析构。

这里要注意的是，at_destruct 的构造在 base_和 out_之前，所以如果直接给 cv_传递 out_时不行的，这个时候 out_还没有初始化呢。但是在这个时候，虽然 base_同样尚未初始化，但 base_.out_的引用却是有效的，因此我们可以给 cv_传递一个 base_.out_。

另外，at_destruct 虽然定义了带 str 参数的构造函数，CHECK_TYPE_MEM_FUNC__宏中却没有使用它。原因是若在宏中使用#__VA_ARGS__作为参数，那么当变参为空时，#__VA_ARGS__前面的逗号在 vc 中不会被自动忽略掉（gcc 会忽略）。

最后，来一起看看输出效果吧：

```
class Foo {};
std::cout << check_type<int (Foo::* const)(int, Foo&&, int) volatile>() << std::endl;
// 输出：int (Foo::* const) (int, Foo &&, int) volatile
// 这是一个常类成员函数指针，指向 Foo 里的一个 volatile 成员函数
```

尾声

折腾 C++的类型系统是一个很有意思的事情。当钻进去之后就会发现，一些原先比较晦涩的基本概念，在研究的过程中都清晰了不少。

check_type 的实用价值在于，可以利用它清晰的看见 C++中一些隐藏的类型变化。比如完美转发时的引用折叠：

```
class Foo {};
```

```
template <typename T>
auto func(T&&) -> T;

std::cout << check_type<decltype(func<Foo>)>() << std::endl;
std::cout << check_type<decltype(func<Foo&>)>() << std::endl;
std::cout << check_type<decltype(func<Foo&&>)>() << std::endl;
```

在上面实现 check_type 的过程中，用到了不少泛型，甚至元编程的小技巧，充分运用了 C++在预处理期、编译期和运行期（RAII）的处理能力。虽然这些代码仅是学习研究时的兴趣之作，实际项目中往往 typeid 的返回结果就足够了，但上面的不少技巧对一些现实中的项目开发也有一定的参考和学习价值。

顺便说一下：上面的代码里使用了大量 C++11 的特征。若想在老 C++中实现 check_type，大部分的新特征也都可以找到替代的手法。只是适配函数类型时使用的变参模板，在 C++98/03 下实现起来实在抽搐。论代码的表现力和舒适度，C++11 强过 C++98/03 太多了。

完整代码及测试下载请点击：check_type

Wrote by mutouyun. (http://darkc.at/cxx-get-the-name-of-the-given-type/)
