C++的模板技巧:编译器探测类成员

C++0x 提供了丰富的 type trait 用于 generic 编程。但是，其中并没有探测类成员的 type trait.不借助编译器的帮助，要实现这个 type trait 是很困难的。这里我们对需求进行适当的修改：探测类中是否存在指定名称和类型的成员。

在 C++中，函数重载是最常见的实现 type trait 的方法。但是，函数重载是基于类型的。默认参数和访问权限都在函数重载之后进行。这里我们希望探测指定的成员是否存在，所以需要找到一种将成员转换为类型的方法。幸运的是，模板支持非类型的参数。下面展示了基于这一想法的实现：

```
namespace van {
    namespace type_traits {
        namespace detail {
            typedef char Small;
            struct Big {char dummy[2];};

            template<typename Type,Type Ptr>
            struct MemberHelperClass;

            template<typename T,typename Type>
            Small MemberHelper_f(MemberHelperClass<Type,&T::f> *);
            template<typename T,typename Type>
            Big MemberHelper_f(...);
        }

        template<typename T,typename Type>
        struct has_member_f
        {
            enum
{value=sizeof(detail::MemberHelper_f<T,Type>(0))==sizeof(detail::Small)};
        };
    }
}

struct A
{
    static void f();
```

```
};
struct B
{
};

#include <iostream>
using namespace std;

int main()
{
    cout<<boolalpha;
    cout<<van::type_traits::has_member_f<A,void (*)()>::value<<endl;
    cout<<van::type_traits::has_member_f<B,void (*)()>::value<<endl;
}
```

如果成员“f”不存在，那么地址“&T:：f”到类型“MemberHelperClass”的转换是无效的，所以接受不定长参数的重载版本会被选中。否则，因为接受不定长参数的版本在重载决议中优先级最低，接受“MemberHelperClass”的版本会被选中。然后 has_member_f 就能够通过检查被重载决议选中的 MemberHelper_f 函数的返回值来判断成员“f”是否存在。上面的代码既支持静态成员，也支持非静态成员。它也同时支持成员函数和成员变量。不过，上面的方法有一个缺陷。如果探测的成员不是 public 的，会导致编译错误。这是因为访问权限检查是在重载决议之后进行的。

因为成员名称本身不能作为模板参数，我们必须将它显式的加入我们的辅助类的类名中以便区分。为了避免重复工作，我们可以利用宏编写出如下的通用版本：

```
 #define DEFINEHASMEMBER(Name)\
namespace van {\
    namespace type_traits {\
namespace detail {\
            template<typename T,typename Type>\
            Small MemberHelper_##Name(MemberHelperClass<Type,&T::Name> *);\
            template<typename T,typename Type>\
            Big MemberHelper_##Name(...);\
        }\
\
        template<typename T,typename Type>\
        struct has_member_##Name\
        {\
            enum
{value=sizeof(detail::MemberHelper_##Name<T,Type>(0))==sizeof(detail::Small)};\
```

```
        };\
    }\
```

这个 type trait 的一个用处是简化 dispatcher.比如，为了提供程序的性能，我们需要针对不同的平台提供不同的实现。利用成员探测的 type trait，我们可以让 dispatch 完全自动化。

首先，我们将不同的实现放在同一个辅助类里

```
 struct MemoryCopyHelper
{
    typedef void (*FunctionType)(const void *lpDest, void *lpSrc, size_t
n);
    static void Default(const void *lpDest, void *lpSrc, size_t n){}
    static void MMX(const void *lpDest, void *lpSrc, size_t n){}
};
```

其次，我们创建一个函数数组用于存放每个实现的地址。如果某个平台没有对应的优化实现，我们将采用默认的版本（假定默认版本总是存在）

```
 DEFINEHASMEMBER(Default)
DEFINEHASMEMBER(MMX)
DEFINEHASMEMBER(SSE2)

#define DEFINESELECTSTATICMEMBER(MemberName)\
    template<typename T,typename FunType,bool =
van::type_traits::has_member_##MemberName<T,FunType>::value>\
    struct select_member_##MemberName;\
    template<typename T,typename FunType>\
    struct select_member_##MemberName<T,FunType,true> {static const FunType value;};\

    template<typename T,typename FunType>\
    struct select_member_##MemberName<T,FunType,false> {static const FunType
value;};\
    template<typename T,typename FunType>\
    const FunType
select_member_##MemberName<T,FunType,true>::value=&T::MemberName;\
    template<typename T,typename FunType>\
    const FunType select_member_##MemberName<T,FunType,false>::value=&T::Default;
DEFINESELECTSTATICMEMBER(Default)
DEFINESELECTSTATICMEMBER(MMX)
DEFINESELECTSTATICMEMBER(SSE2)

MemoryCopyHelper::FunctionType gDispatchArray_MemoryCopy[]={
```

```
    select_member_Default<MemoryCopyHelper,
MemoryCopyHelper::FunctionType>::value,
    select_member_MMX<MemoryCopyHelper, MemoryCopyHelper::FunctionType>::value,
    select_member_SSE2<MemoryCopyHelper, MemoryCopyHelper::FunctionType>::value,
};
```

然后你就可以将精力集中在实现上。你可以在以后向辅助类中添加其它平台的优化实现版本，函数数组会自动更新（注：上面的数组是在程序进入 main 之前动态初始化的）

上面的代码在 VC8、VC9和 gcc 3.4.5上测试通过。一些老的编译器可能无法编译通过。