Introducing an optional parameter for `mem_fn`, which allows to bind an object to its member function

Mikhail Semenov, 2013-05-29

1. Introduction

In some situations it is necessary to bind an object to its member function, in which case the result could be treated as an ordinary function. Let us consider a case when the following function is defined, whose parameter is another function:

```
double integrate

     (function<double(double, double, double, double double)> f)
```

A class has been created, which provides a parameter to a functions **fa** that we want to use:

```
class A
{
    double x;

public:
    A(double x1):x(x1) {}

    double fa(double x, double y, double z, double a, double b);
};
```

We create an object:

```
A a(5.0);
```

If we want to call `integrate` with `a.fa` we have two options:

(1)

```
double r = integrate(std::bind(&A::fa, &a, _1, _2, _3, _4, _5));
```

(2)

```
double r = integrate([&a](double x, double y, double z, double a,
double b){ return a.fa(x, y, z, a, b);});
```

If the function **f** had more parameters we would have to write even a longer statement. It would be much more convenient to be able to write something like this:

```
double r = integrate(some_binder(&A::fa, &a));
```

2. Proposal

The proposal is to allow **mem_fn** to accept a second, optional parameter. The full definition is as follows (the first option is the same as in **N3485**):

template<class R, class T> unspecified mem_fn(R T::* pm);

Returns: A simple call wrapper (20.8.1) fn such that the expression **fn(t, a2, ..., aN)** is equivalent to **INVOKE (pm, t, a2, ..., aN)** (20.8.2). **fn** shall have a nested type **result_type** that is a synonym for the return type of **pm** when **pm** is a pointer to member function.

The simple call wrapper shall define two nested types named **argument_type** and **result_type** as synonyms for **cv T*** and **Ret**, respectively, when pm is a pointer to member function with ***cv***-qualifier ***cv*** and taking no arguments, where ***Ret*** is **pm**'s return type.

The simple call wrapper shall define three nested types named **first_argument_type**, **second_argument_type**, and **result_type** as synonyms for ***cv*** **T***, **T1**, and **Ret**, respectively, when **pm** is a pointer to member function with *cv*-qualifier ***cv*** and taking one argument of type **T1**, where ***Ret*** is **pm**'s return type.

Throws: Nothing.

template<class R, class T> unspecified mem_fn(R T::*, cv T* obj);

Returns: A simple call wrapper (20.8.1) **fn** such that the expression **fn(a1, ..., aN)** is equivalent to **INVOKE (pm, obj, a1, ..., aN)** (20.8.2). **fn** shall have a nested type **result_type** that is a synonym for the return type of **pm** when **pm** is a pointer to member function.

The simple call wrapper shall define one nested type named **result_type** as a synonym for **Ret**, when **pm** is a pointer to member function with ***cv***-qualifier ***cv*** and taking no arguments, where ***Ret*** is **pm**'s return type.

The simple call wrapper shall define two nested types named **argument_type** and **result_type** as synonyms for **T1** and **Ret**, respectively, when **pm** is a pointer to member function with ***cv***-qualifier ***cv*** and taking one argument of type **T1**, where ***Ret*** is **pm**'s return type.

The simple call wrapper shall define three nested types named **first_argument_type**, **second_argument_type**, and **result_type** as synonyms for **T1**, **T2**, and **Ret**, respectively, when **pm** is a pointer to member function with *cv*-qualifier ***cv*** and taking two arguments of types **T1** and **T2**, where ***Ret*** is **pm**'s return type.

Throws: Nothing.

The second option means that the object and the member function will be bound. Using this feature we can now write:

```
double r = integrate(memb_fn(&A::fa, &a));
```

3. Examples

Here is a sample program:

```
#include <iostream>
#include <functional>

class A2
{
    int i;
public:
    A2(int k):i(k) {}

    auto get()const ->int { return i;}
    auto set(int v)->void { i = v;}

    auto inc(int g)->int& { i+=g; return i;}
    auto incp(int& g)->int& { g+=i; return g;}

    auto f8 (int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8)
const -> int
    {
        return i+a1+a2+a3+a4+a5+a6+a7+a8;
    }
};


int main()
{
    A2 a(11);
    auto a_get = std::mem_fn(&A2::get,&a);
    std::cout << a_get() << std::endl;
    auto a_get1 = std::mem_fn(&A2::get);

    std::cout << a_get1(&a) << std::endl;

    auto set1 = std::mem_fn(&A2::set, &a);

    auto inc =  std::mem_fn(&A2::inc, &a);
    auto incp = std::mem_fn(&A2::incp, &a);
    auto af8  = std::mem_fn(&A2::f8,&a);


    set1(15);
    std::cout << "a.get():" << a.get() << std::endl;


    int x = 5;
    int k = inc(x);
    int& k2 = incp(x);

    k2 *= 7;
    std::cout << "a.get():" << a.get() << std::endl;
    std::cout << "k: " << k << std::endl;
    std::cout << "x: " << x << std::endl;
    std::cout << "af8(1,2,3,4,5,6,7,8): "
              <<    af8(1,2,3,4,5,6,7,8) << std::endl;
}
```

This program will print:

```
11
11
a.get():15
a.get():20
k: 20
x: 175
af8(1,2,3,4,5,6,7,8): 56
```

4. Implementation

```
#define _two_param_delegate2(_SIG, _NAME)\
template<class _OBJECT1, class _T, class _R, class _Arg1, class _Arg2,
class ... _Arg>\
class _delegate2 ## _NAME\
{\
    _OBJECT1 _obj;\
    _R (_T::*f1)(_Arg1, _Arg2, _Arg ... ) _SIG;\
public:\
    typedef _Arg1 first_argument_type;\
    typedef _Arg2 second_argument_type;\
    typedef _R return_type;\
\
    _delegate2 ## _NAME(_R (_T::*_fx)(_Arg1, _Arg2, _Arg ... ) _SIG,
_OBJECT1 _obj1):_obj(_obj1),f1(_fx)\
    {\
    } \
\
    _R operator()(_Arg1 _x1, _Arg2 _x2, _Arg ...  _x) _SIG\
    {\
        return ((*_obj).*f1)(std::forward<_Arg1>(_x1),
std::forward<_Arg2>(_x2), std::forward<_Arg>(_x) ...);\
    }\
};

_two_param_delegate2(,_simple)
_two_param_delegate2(const, _const)
_two_param_delegate2(const volatile, _const_volatile)
_two_param_delegate2(volatile, _volatile)
```

```
#define _two_param_delegate1(_SIG, _NAME)\
template<class _OBJECT1, class _T, class _R, class _Arg1>\
class _delegate1 ## _NAME\
{\
    _OBJECT1 _obj;\
    _R (_T::*f1)(_Arg1) _SIG;\
public:\
    typedef _Arg1 argument_type;\
    typedef _R return_type;\
\
    _delegate1 ## _NAME(_R (_T::*_fx)(_Arg1) _SIG, _OBJECT1
_obj1):_obj(_obj1),f1(_fx)\
    {\
    } \
\
    _R operator()(_Arg1 _x1) _SIG\
    {\
        return ((*_obj).*f1)(std::forward<_Arg1>(_x1));\
    }\
};

_two_param_delegate1(,_simple)
_two_param_delegate1(const, _const)
_two_param_delegate1(const volatile, _const_volatile)
_two_param_delegate1(volatile, _volatile)

#define _two_param_delegate0(_SIG, _NAME)\
template<class _OBJECT1, class _T, class _R>\
class _delegate0 ## _NAME\
{\
    _OBJECT1 _obj;\
    _R (_T::*f1)() _SIG;\
public:\
    typedef _R return_type;\
\
    _delegate0 ## _NAME(_R (_T::*_fx)() _SIG, _OBJECT1
_obj1):_obj(_obj1),f1(_fx)\
    {\
    } \
\
    _R operator()() _SIG\
    {\
        return ((*_obj).*f1)();\
    }\
};

_two_param_delegate0(,_simple)
_two_param_delegate0(const, _const)
_two_param_delegate0(const volatile, _const_volatile)
_two_param_delegate0(volatile, _volatile)
```

```
#define _one_param_delegate1(_SIG, _NAME)\
template<class _T, class _R, class _Arg1, class ... _Arg>\
class _one_p_delegate1 ## _NAME\
{\
    _R (_T::*f1)(_Arg1, _Arg ... ) _SIG;\
public:\
    typedef _SIG _T* first_argument_type;\
    typedef _Arg1 second_argument_type;\
    typedef _R return_type;\
\
    _one_p_delegate1 ## _NAME(_R (_T::*_fx)(_Arg1, _Arg ... )
_SIG):f1(_fx)\
    {\
    } \
\
    template<class _OBJECT>\
    _R operator()(_OBJECT _obj,_Arg1 _x1, _Arg ...  _x) _SIG\
    {\
        return ((*_obj).*f1)(std::forward<_Arg1>(_x1),
std::forward<_Arg>(_x) ...);\
    }\
};

_one_param_delegate1(,_simple)
_one_param_delegate1(const, _const)
_one_param_delegate1(const volatile, _const_volatile)
_one_param_delegate1(volatile, _volatile)

#define _one_param_delegate0(_SIG, _NAME)\
template<class _T, class _R>\
class _one_p_delegate0 ## _NAME\
{\
    _R (_T::*f1)() _SIG;\
public:\
    typedef _SIG _T* argument_type;\
    typedef _R return_type;\
\
    _one_p_delegate0 ## _NAME(_R (_T::*_fx)() _SIG):f1(_fx)\
    {\
    } \
\
    template<class _OBJECT>\
    _R operator()(_OBJECT _obj) _SIG\
    {\
        return ((*_obj).*f1)();\
    }\
};

_one_param_delegate0(,_simple)
_one_param_delegate0(const, _const)
_one_param_delegate0(const volatile, _const_volatile)
_one_param_delegate0(volatile, _volatile)

#define _mem_fn_macro2(_SIG,_NAME)\
template<class _OBJECT, class _T, class _R, class _Arg1, class _Arg2,
class ... _Arg>\
_delegate2 ## _NAME<_OBJECT,_T,_R,_Arg1, _Arg2, _Arg ... > mem_fn(_R
(_T::*_f)(_Arg1, _Arg2, _Arg ... ) _SIG, _OBJECT _obj)\
{\
    return _delegate2 ## _NAME <_OBJECT,_T,_R, _Arg1, _Arg2, _Arg ... >(_f,
_obj);\
}
```

```
_mem_fn_macro2(,_simple)
_mem_fn_macro2(const,_const)
_mem_fn_macro2(const volatile, _const_volatile)
_mem_fn_macro2(volatile, _volatile)


#define _mem_fn_macro1(_SIG,_NAME)\
template<class _OBJECT, class _T, class _R, class _Arg1>\
_delegate1 ## _NAME<_OBJECT,_T,_R,_Arg1> mem_fn(_R (_T::*_f)(_Arg1) _SIG,
_OBJECT _obj)\
{\
    return _delegate1 ## _NAME <_OBJECT,_T,_R, _Arg1>(_f, _obj);\
}

_mem_fn_macro1(,_simple)
_mem_fn_macro1(const,_const)
_mem_fn_macro1(const volatile, _const_volatile)
_mem_fn_macro1(volatile, _volatile)

#define _mem_fn_macro0(_SIG,_NAME)\
template<class _OBJECT, class _T, class _R>\
_delegate0 ## _NAME<_OBJECT,_T,_R> mem_fn(_R (_T::*_f)() _SIG, _OBJECT
_obj)\
{\
    return _delegate0 ## _NAME <_OBJECT,_T,_R>(_f, _obj);\
}

_mem_fn_macro0(,_simple)
_mem_fn_macro0(const,_const)
_mem_fn_macro0(const volatile, _const_volatile)
_mem_fn_macro0(volatile, _volatile)

#define _one_p_mem_fn_macro1(_SIG,_NAME)\
template<class _T, class _R, class _Arg1, class ... _Arg>\
_one_p_delegate1 ## _NAME<_T,_R,_Arg1, _Arg ... > mem_fn(_R (_T::*_f)(_Arg1,
_Arg ... ) _SIG)\
{\
    return _one_p_delegate1 ## _NAME <_T,_R,_Arg1, _Arg ... >(_f);\
}

_one_p_mem_fn_macro1(,_simple)
_one_p_mem_fn_macro1(const,_const)
_one_p_mem_fn_macro1(const volatile, _const_volatile)
_one_p_mem_fn_macro1(volatile, _volatile)

#define _one_p_mem_fn_macro0(_SIG,_NAME)\
template<class _T, class _R>\
_one_p_delegate0 ## _NAME<_T,_R> mem_fn(_R (_T::*_f)() _SIG)\
{\
    return _one_p_delegate0 ## _NAME <_T,_R>(_f);\
}

_one_p_mem_fn_macro0(,_simple)
_one_p_mem_fn_macro0(const,_const)
_one_p_mem_fn_macro0(const volatile, _const_volatile)
_one_p_mem_fn_macro0(volatile, _volatile)
```