Boost.Functional/Forward 1.0

Tobias Schwinger

Copyright © 2007, 2008 Tobias Schwinger

Table of Contents

Brief Description

`boost::forward_adapter` provides a reusable adapter template for function objects. It forwards RValues as references to const, while leaving LValues as-is.

```cpp
struct g // function object that only accept LValues
{
    template< typename T0, typename T1, typename T2 >
    void operator()(T0 & t0, T1 & t1, T2 & t2) const;

    typedef void result_type;
};

// Adapted version also accepts RValues and forwards
// them as references to const, LValues as-is
typedef boost::forward_adapter<g> f;
```

Another adapter, `boost::lighweight_forward_adapter` allows forwarding with some help from the user accepting and unwrapping reference wrappers (see Boost.Ref) for reference arguments, const qualifying all other arguments.

The target functions must be compatible with Boost.ResultOf, and so are the adapters.

Background

Let's suppose we have some function `f` that we can call like this:

```
f(123,a_variable);
```

Now we want to write another, generic function `g` that can be called the same way and returns some object that calls `f` with the same arguments.

```
f(123,a_variable) == g(f,123,a_variable).call_f()
```

Why would we want to do it, anyway?

Maybe we want to run `f` several times. Or maybe we want to run it within another thread. Maybe we just want to encapsulate the call expression for now, and then use it with other code that allows to compose more complex expressions in order to decompose it with C++ templates and have the compiler generate some machinery that eventually calls `f` at runtime (in other words; apply a technique that is commonly referred to as Expression Templates).

Now, how do we do it?

The bad news is: It's impossible.

That is so because there is a slight difference between a variable and an expression that evaluates to its value: Given

```
int y;
int const z = 0;
```

and

```
template< typename T > void func1(T & x);
```

we can call

```
func1(y); // x is a reference to a non-const object
func1(z); // x is a reference to a const object
```

where

```
func1(1); // fails to compile.
```

This way we can safely have `func1` store its reference argument and the compiler keeps us from storing a reference to an object with temporary lifetime.

It is important to realize that non-constness and whether an object binds to a non-const reference parameter are two different properties. The latter is the distinction between LValues and RValues. The names stem from the left hand side and the right hand side of assignment expressions, thus LValues are typically the ones you can assign to, and RValues the temporary results from the right hand side expression.

```
y = 1+2;        // a is LValue, 1+2 is the expression producing the RValue,
// 1+2 = a;     // usually makes no sense.

func1(y);       // works, because y is an LValue
// func1(1+2);  // fails to compile, because we only got an RValue.
```

3

If we add const qualification on the parameter, our function also accepts RValues:

```
template< typename T > void func2(T const & x);

// [...] function scope:
func2(1); // x is a reference to a const temporary, object,
func2(y); // x is a reference to a const object, while y is not const, and
func2(z); // x is a reference to a const object, just like z.
```

In all cases, the argument x in func2 is a const-qualified LValue. We can use function overloading to identify non-const LValues:

```
template< typename T > void func3(T const & x); // #1
template< typename T > void func3(T & x);       // #2

// [...] function scope:
func3(1); // x is a reference to a const, temporary object in #1,
func3(y); // x is a reference to a non-const object in #2, and
func3(z); // x is a reference to a const object in #1.
```

Note that all arguments x in the overloaded function func3 are LValues. In fact, there is no way to transport RValues into a function as-is in C++98. Also note that we can't distinguish between what used to be a const qualified LValue and an RValue.

That's as close as we can get to a generic forwarding function g as described above by the means of C++ 98. See The Forwarding Problem for a very detailed discussion including solutions that require language changes.

Now, for actually implementing it, we need 2^N overloads for N parameters (each with and without const qualifier) for each number of arguments (that is 2^(Nmax+1) - 2^Nmin). Right, that means the compile-time complexity is O(2^N), however the factor is low so it works quite well for a reasonable number (< 10) of arguments.

Reference

forward_adapter

Description

Function object adapter template whose instances are callable with LValue and RValue arguments. RValue arguments are forwarded as reference-to-const typed LValues.

An arity can be given as second, numeric non-type template argument to restrict forwarding to a specific arity. If a third, numeric non-type template argument is present, the second and third template argument are treated as minimum and maximum arity, respectively. Specifying an arity can be helpful to improve the readability of diagnostic messages and compile time performance.

Boost.ResultOf can be used to determine the result types of specific call expressions.

Header

```
#include <boost/functional/forward_adapter.hpp>
```

Synopsis

```
namespace boost
{
    template< class Function,
        int Arity_Or_MinArity = unspecified, int MaxArity = unspecified >
    class forward_adapter;
}
```

Notation

F   a possibly const qualified function object type or reference type thereof

f   an object convertible to F

FA   the type forward_adapter<F>

fa   an instance object of FA, initialized with f

a0...aN   arguments to fa

The result type of a target function invocation must be

```
boost::result_of<F*(TA0 [const]&...TAN [const]&)>::type
```

where TA0...TAN denote the argument types of a0...aN.

Expression Semantics

Expression	Semantics
FA(f)	creates an adapter, initializes the target function with f.
FA()	creates an adapter, attempts to use F's default constructor.
fa(a0...aN)	calls f with with arguments a0...aN.

Limits

The macro BOOST_FUNCTIONAL_FORWARD_ADAPTER_MAX_ARITY can be defined to set the maximum call arity. It defaults to 6.

Complexity

Preprocessing time: $O(2^N)$, where N is the arity limit. Compile time: $O(2^N)$, where N depends on the arity range. Run time: $O(0)$ if the compiler inlines, $O(1)$ otherwise.

lightweight_forward_adapter

Description

Function object adapter template whose instances are callable with LValue and RValue arguments. All arguments are forwarded as reference-to-const typed LValues, except for reference wrappers which are unwrapped and may yield non-const LValues.

An arity can be given as second, numeric non-type template argument to restrict forwarding to a specific arity. If a third, numeric non-type template argument is present, the second and third template argument are treated as minimum and maximum arity, respectively. Specifying an arity can be helpful to improve the readability of diagnostic messages and compile time performance.

Boost.ResultOf can be used to determine the result types of specific call expressions.

Header

```
#include <boost/functional/lightweight_forward_adapter.hpp>
```

Synopsis

```
namespace boost
{
    template< class Function,
        int Arity_Or_MinArity = unspecified, int MaxArity = unspecified >
    struct lightweight_forward_adapter;
}
```

Notation

F	a possibly const qualified function object type or reference type thereof
f	an object convertible to F
FA	the type lightweight_forward_adapter<F>
fa	an instance of FA, initialized with f
a0...aN	arguments to fa

The result type of a target function invocation must be

```
boost::result_of<F*(TA0 [const]&...TAN [const]&)>::type
```

where TA0...TAN denote the argument types of a0...aN.

Expression Semantics

Expression	Semantics
`FA(f)`	creates an adapter, initializes the target function with `f`.
`FA()`	creates an adapter, attempts to use `F`'s default constructor.
`fa(a0...aN)`	calls `f` with with const arguments `a0...aN`. If `aI` is a reference wrapper it is unwrapped.

Limits

The macro BOOST_FUNCTIONAL_LIGHTWEIGHT_FORWARD_ADAPTER_MAX_ARITY can be defined to set the maximum call arity. It defaults to 10.

Complexity

Preprocessing time: O(N), where N is the arity limit. Compile time: O(N), where N is the effective arity of a call. Run time: O(0) if the compiler inlines, O(1) otherwise.

Acknowledgements

As these utilities are factored out of the Boost.Fusion functional module, I want to thank Dan Marsden and Joel de Guzman for letting me participate in the development of that great library in the first place.

Further, I want to credit the authors of the references below, for their in-depth investigation of the problem and the solution implemented here.

Last but not least I want to thank Vesa Karnoven and Paul Mensonides for the Boost Preprocessor library. Without it, I would have ended up with an external code generator for this one.

References

1. The Forwarding Problem, Peter Dimov, Howard E. Hinnant, David Abrahams, 2002

2. Boost.ResultOf, Douglas Gregor, 2004

3. Boost.Ref, Jaakko Jarvi, Peter Dimov, Douglas Gregor, David Abrahams, 1999-2002