Instantiations Must Go!

Matt Calabrese
Zach Laine

May 10, 2010

A funny thing happened on the way to BoostCon

Why did you tell me this ridiculous story?

- Like Kennybrook, we did something stupid with serendipitous results

Why did you tell me this ridiculous story?

- This isn't a “serious business” BoostCon lecture

Why did you tell me this ridiculous story?

- I just like telling it

The Prblem

The Problem

- Template Metaprogramming is fun
  - Allows you to do lots of things at compile time that make your life easier/better
  - Makes your brain seem bigger to others

The Problem

- Template instantiations are expensive at compile-time

The Problem

- Each instantiation must do a lot of work inherently
- Some compilers have really slow instantiation code (I'm looking at you, GCC < 4.5)

The Problem

- Notable Boost examples:
  - Proto uses macro workarounds extensively
  - Spirit code can be unmaintainable in large enough quantities

The “Solution”

The Solution

- Easy – don't instantiate templates...
- ... which is stupid, remember?

The Solution

- Here's what we want to avoid:

```
template <typename T>
struct metafunction
{
    typedef /* something with T */ type;
};

metafunction<int>::type x = /* … */;
```

- Just don't reach into `metafunction` to grab `type`, and we don't instantiate any templates, right?
- Not really (more on this later)

The Solution: The General Idea

- C++0x's `decltype` and trailing return type function declaration syntax make it possible to do a lot of TMP without struct templates – let's try some approaches that use expressions passed to `decltype`.

The Solution: The General Idea

- `decltype`, in loose terms, yields the type of the expression given to it, turning this:

```
template <typename T, typename U>
struct result_of_plus
{ typedef /* usually something complicated */ type; };

template <typename T, typename U>
typename result_of_plus<T, U>::type
operator+(const T& t, const U& u)
{ return t + u; }
```

- … into this:

```
template <typename T, typename U>
decltype((*(T*)0) + (*(U*)0))
operator+(const T& t, const U& u)
{ return t + u; }
```

The Solution: The General Idea

- The trailing return type syntax allows us to use parameter names in `decltype` expressions. This:

```
template <typename T, typename U>
decltype((*(T*)0) + (*(U*)0))
operator+(const T& t, const U& u)
{ return t + u; }
```

- … becomes this:

```
template <typename T, typename U>
auto operator+(const T& t, const U& u) -> decltype(t + u)
{ return t + u; }
```

The Solution: Approach 1

```
// vector.h:

template <
    typename ... Types
>
struct vector_;

template <>
struct vector_<>
{};

template <
    typename Head,
    typename ... Tail
>
struct vector_<Head, Tail ...> :
    vector_<Tail ...>
{};
```

The Solution: Approach 1

```
#include <boost/mpl/print.hpp>
#include "vector.h"

template <typename ... Args, typename T>
vector_<Args ..., T>
(* push_back_(vector_<Args ...> (*) (), T (*) ())) ();

int main()
{
    typedef vector_<> empty_vec;

    typedef decltype(
        push_back_(
            (empty_vec (*)())0, (int (*)())0
        )()
    ) int_vec;

    typedef boost::mpl::print<int_vec>::type foo;

    return 0;
}
```

Compiler Output:

```
.../boost/mpl/print.hpp: In instantiation of
boost::mpl::print<vector_<int> >: ...
```

The Solution: Approach 1

- Pros:
  - It works!
- Cons:
  - It's disgusting!

The Solution: Approach 2: Wrap/Unwrap

- Let's automate this wrapping business

```
// wrap_unwrap.h

template <typename T>
struct wrap;

template <typename T>
struct unwrap;

template <typename T>
struct unwrap<wrap<T> *>
{ typedef T type; };
```

The Solution: Approach 2, Part II

```
#include <boost/mpl/print.hpp>
#include "vector.h"
#include "wrap_unwrap.h"

template <typename ... Args, typename T>
wrap<vector_<Args ..., T> > *
push_back_(wrap<vector_<Args ...> > *, wrap<T> *);

int main()
{
    typedef vector_<> empty_vec;

    typedef unwrap<
        decltype(
            push_back_(
                (wrap<empty_vec> *)0,
                (wrap<int> *)0
            )
        )
    >::type int_vec;

    typedef boost::mpl::print<int_vec>::type foo;

    return 0;
}
```

Compiler Output:

```
.../boost/mpl/print.hpp: In instantiation of boost::mpl::print<vector_<int>
>: ...
```

The Solution: Approach 2

- Pros:
  - It still works!
- Cons:
  - It's slightly less disgusting!

The Solution: The final version's wrappers

```
// wrap_unwrap_2.h

template< typename T >
struct type_ {};

template< typename T >
type_< T >& type();

template< typename T >
struct unwrap;

template< typename T >
struct unwrap< type_<T>& >
{ typedef T type; };
```

- We've established a new parameter/return type convention, that:
  - Leaves types incomplete where possible
  - Requires type() and unwrap() calls, but only at the library boundaries

The Solution: The final version

```
#include <boost/mpl/print.hpp>
#include “vector.h”
#include “wrap_unwrap_2.h”

template< typename ... T >
type_< vector_< T... > >& vector();

template< typename ... T, typename Elem >
type_< vector_< T..., type_< Elem >& > >&
push_back( type_< vector_< T... > >&, type_< Elem >& );


int main()
{
    typedef unwrap<
        decltype(
            push_back(vector(), type<int>())
        )
    >::type int_vec;

    typedef boost::mpl::print<int_vec>::type foo;

    return 0;
}
```

Compiler Output:

```
.../boost/mpl/print.hpp: In instantiation of
boost::mpl::print<vector_<type_<int>& > >: ...
```

The Solution: The final version

- Pros:
  - As designed, it does TMP without instantiating `struct` definitions
  - It's more readable than standard TMP
    - Allows mere mortals to read TMP code written by others
    - Is much more teachable
  - It's damn sexy

The Solution: The final version

- Cons:
  - It still instantiates `struct` declarations
  - Its compile-time-reducing performance is questionable

The Solution: The numbers

- Compile-time performance measures:
  - Test was a deeply nested series of vector push_back's
  - A robust test would require reimplementing Boost.MPL and Spirit, Proto, or some other MPL user

The Solution: The numbers

- GCC >= 4.4 Numbers

```
gcc (64-bit) 4.4.0:  mpl: 0.5575s ftmpl: 0.4467s -- 1.25x ftmpl speedup
gcc (64-bit) 4.4.1:  mpl: 0.5726s ftmpl: 0.4419s -- 1.30x ftmpl speedup
gcc (64-bit) 4.4.2:  mpl: 0.5628s ftmpl: 0.4409s -- 1.28x ftmpl speedup
gcc (32-bit) 4.4.3:  mpl: 0.5777s ftmpl: 0.4559s -- 1.27x ftmpl speedup
gcc (64-bit) 4.4.4:  mpl: 0.5679s ftmpl: 0.4592s -- 1.24x ftmpl speedup
gcc (64-bit) 4.5.0:  mpl: 0.5116s ftmpl: 0.4773s -- 1.07x ftmpl speedup
```

- The FTMPL version:

```
#include <boost/mpl/print.hpp>

#include "vector.hpp"
#include "type.hpp"
#include "unwrap.hpp"

int main()
{
    using namespace boost::ftmpl;

    typedef BOOST_FTMPL_UNWRAP(
        push_back(vector(), type<int>())
    ) int_vec;

    typedef boost::mpl::print<int_vec>::type foo;

    return 0;
}
```

FTMPL: Readability

- `decltype` allows us to do some very interesting TMP with expressions:

```
typedef BOOST_FTMPL_UNWRAP(
    int_<3>() + int_<6>()
) int_9;
typedef boost::mpl::print<int_9>::type foo;

int result = BOOST_FTMPL_UNWRAP_VALUE(
    int_<3>() & int_<6>()
);
std::cout << "result = " << result << '\n';

Compiler Output:
.../boost/mpl/print.hpp: In instantiation of
boost::mpl::print<boost::ftmpl::value_t<int, 9> >: ...

Program Output:
result = 2
```

FTMPL: Readability

- Metafunction objects actually look like … function objects!

```
struct is_same_t
{
    template< typename L, typename R >
    type_< false_t >& operator()( L&, R& );

    template< typename L >
    type_< true_t >& operator()( L&, L& );
};

typedef BOOST_FTMPL_UNWRAP(
    is_same_t()(type< char >(), type< char >())
) is_same_result;
typedef boost::mpl::print<is_same_result>::type foo;

Compiler Output:
.../boost/mpl/print.hpp: In instantiation of
boost::mpl::print<boost::ftmpl::value_t<bool, true> >: ...
```

FTMPL: Readability

- Contrast this with type_traits/is_same.hpp:

```
template <typename T>
::boost::type_traits::yes_type
BOOST_TT_DECL is_same_tester(T*, T*);

::boost::type_traits::no_type
BOOST_TT_DECL is_same_tester(...);

template <typename T, typename U>
struct is_same_impl
{
  static T t;
  static U u;

  BOOST_STATIC_CONSTANT(bool, value =
    (::boost::type_traits::ice_and<
      (sizeof(type_traits::yes_type) == sizeof(detail::is_same_tester(&t,&u))),
      (::boost::is_reference<T>::value == ::boost::is_reference<U>::value),
      (sizeof(T) == sizeof(U))
     >::value));
};
```

FTMPL: Readability

- Higher-order metafunctions are actually simple enough to grok at first glance:

```
template< typename Metafun, typename ... Args >
auto apply( type_< Metafun >&, Args& ... args )
    -> decltype( Metafun()( args... ) );

extern type_< is_same_t >& is_same;

typedef BOOST_FTMPL_UNWRAP(
    apply( is_same, type< char >(), type< int >() )
) applied_is_same_result;
typedef boost::mpl::print<applied_is_same_result>::type foo;

Compiler Output:
.../boost/mpl/print.hpp: In instantiation of
boost::mpl::print<boost::ftmpl::value_t<bool, false> >: ...
```

FTMPL: Readability

- Higher-order metafunctions are actually simple enough to grok at first glance:

```
template<
    typename Fn,
    typename State
>
State& fold( Fn&, State&, type_< vector_t<> >& );

template<
    typename Fn,
    typename State,
    typename Head,
    typename ... Tail
>
auto fold(
    type_< Fn >&,
    type_< State >&,
    type_< vector_t< type_< Head >, Tail... > >&
)
-> decltype(
    fold(
        type< Fn >(),
        Fn()( type< State >(), type< Head >() ),
        vector< Tail... >()
    )
);
```

FTMPL: Readability

- Contrast with mpl/fold.hpp (main implementation is elsewhere):

```
template<
      typename BOOST_MPL_AUX_NA_PARAM(Sequence)
    , typename BOOST_MPL_AUX_NA_PARAM(State)
    , typename BOOST_MPL_AUX_NA_PARAM(ForwardOp)
    >
struct fold
{
    typedef typename aux::fold_impl<
          ::boost::mpl::O1_size<Sequence>::value
        , typename begin<Sequence>::type
        , typename end<Sequence>::type
        , State
        , ForwardOp
        >::state type;

    BOOST_MPL_AUX_LAMBDA_SUPPORT(3,fold,(Sequence,State,ForwardOp))
};
```

Functional MPL Work-In-Progress

- FTMPL (quite broken)
  - https://svn.boost.org/svn/boost/sandbox/ftmpl
- MPL 0x (maybe broken?)
  - http://github.com/dabrahams/mpl0x
  - http://github.com/swatanabe/mpl0x

Questions?