Toward Boost.Opaque 0.1.1

Vicente J. Botet Escriba

Table of Contents

> “Strong type checking is gold; normal type checking is silver; and casting is brass”
>
> --

Warning

Opaque is not a part of the Boost libraries.

Overview

How to Use This Documentation

This documentation makes use of the following naming and formatting conventions.

- Code is in `fixed width font` and is syntax-highlighted.
- Replaceable text that you will need to supply is in *`italics`*.
- If a name refers to a free function, it is specified like this: `free_function()`; that is, it is in code font and its name is followed by `()` to indicate that it is a free function.
- If a name refers to a class template, it is specified like this: `class_template<>`; that is, it is in code font and its name is followed by <> to indicate that it is a class template.
- If a name refers to a function-like macro, it is specified like this: `MACRO()`; that is, it is uppercase in code font and its name is followed by `()` to indicate that it is a function-like macro. Object-like macros appear without the trailing `()`.
- Names that refer to *concepts* in the generic programming sense are specified in CamelCase.

Note

In addition, notes such as this one specify non-essential information that provides additional background or rationale.

Finally, you can mentally add the following to any code fragments in this document:

```
// Include all of the core Opaque files
#include <boost/opaque.hpp>

using namespace boost;
```

Motivation

The notion of "opaque typedefs" is a recurring theme (see []). Boost.Serialization contains a macro which is used to define what Robert calls "strong typedefs", but the implementation is not complete.

The Walter E. Brown's proposals (**N1891: Progress toward Opaque Typedefs for C++0X** and **N1706: Toward Opaque Typedefs in C++0X**) include a clear motivation for "opaque typedefs".

Alisdair Meredith showed in **N2141: Strong Typedefs in C++09(Revisited)** that the new C++0x feature that implicitly generates forward constructors to a base class, could not be useful to address this issue completly. Using this new feature it becomes possible to create something very like the required opaque typedef. For example:

```
struct MyType : std::string {
    using string::string;
};
```

This type is distrinct from `std::string`, functions can be overloaded on this type as well as `std::string`, yet `std::string` is not convertible to this type.

As this proposals will not be part of the C++0x standard, a library solution that could satisfy most of the requirements seems a loable alternative.

Description

Boost.Opaque intends to provide a library partial solution to this problem.

Boost.Opaque provides:

- a generic mixin class hierarchy which can be specialized by the user to make new opaque classes.
- a generic class hierarchy which can be instantiated by the user to make new opaque typedefs.

- a meta-mixin concept that allows to compose in a easy way several aspects of a class in an orthogonal way.
- a considerable number of meta-mixins that helps defining new types from an underlying type
- Some helper macros that can reduce the declaration of a new opaque type to a single line, emulating the language-based approach.

Users' Guide

Getting Started

Installing Boost.Opaque

Getting Boost.Opaque

You can get the last stable release of **Boost.Opaque** by downloading `opaque.zip` from the Boost Vault Utilities directory

You can also access the latest (unstable?) state from the Boost Sandbox.

Building Boost.Opaque

There is no need to compile **Boost.Opaque**, since it's a header only library. Just include your Boost header directory in your compiler include path.

Requirements

Boost.Opaque depends only on Boost.Operators (and all libraries it depends on).

Exceptions safety

All functions in the library are exception-neutral and provide strong guarantee of exception safety as long as the underlying parameters provide it.

Thread safety

All functions in the library are thread-unsafe except when noted explicitly.

Tested compilers

The implementation will eventually work with most C++03 conforming compilers. Current version has been tested on:

Windows with

- MSVC 10.0

Cygwin 1.5 with

- GCC 3.4.4

Cygwin 1.7 with

- GCC 4.3.4

MinGW with

- GCC 4.4.0
- GCC 4.5.0
- GCC 4.5.0 C++0x

- GCC 4.6.0
- GCC 4.6.0 C++0x

Ubuntu 10.10

- GCC 4.4.5
- GCC 4.4.5 -std=c++0x
- GCC 4.5.1
- GCC 4.5.1 -std=c++0x
- clang 2.8

Note

Please let us know how this works on other platforms/compilers.

Note

Please send any questions, comments and bug reports to boost <at> lists <dot> boost <dot> org.

Hello World!

Tutorial

How to define a real new typedef?

We will start with the definition of a new type identifier which is based on an underlying `int` type and has the relational operators so it can be stored on a sorted container.

```
struct Identifier_tag;
typedef boost::opaque::new_type< int, Identifier_tag,
    boost::mpl::vector<
        opaque::using_totally_ordered1
    >
> Identifier;
```

The declaration of a new typedef Identifier is done as follows. We need to declare a specific tag that will ensure that two new types are different.

```
struct Identifier_tag;
```

New we use the opaque `new_type` class which has as parameters:

- the underlying type (`int`),
- the unique tag of the `new type` (`Identifier_tag`),
- a MPL sequence of additions (`mpl::vector<opaque::using_totally_ordered1>`)

This will declare `Identifier` as new type different from `int` that provides just the operation associated to a `totally_ordered` concept. The concepts are represented by the library in the form a meta-mixin and are named with the `using_` prefix, e.g. as `using_totally_ordered1`. We will see later what a meta-mixin intends for.

We can now use `Identifier` as new `int` type, but limited to the totally ordered operations.

Boost.Opaque provides a macro to simplify the preceding declaration

```
BOOST_OPAQUE_NEW_TYPE(int, Identifier, (opaque::using_totally_ordered1))
```

The macros is responsable to define a unique tag and transform the preprocessor sequence into a MPL sequence.

How to define a real new type?

In the prededing example we have used a tag to ensure type unicity. When the user needs to add yet some specific methods, this tag is no more necessary as we need to define already a new class.

```
class Identifier : boost::opaque::new_class<Identifier, int,
    boost::mpl::vector<
        opaque::using_totally_ordered1
    >
{
    void print();
    // ...
};
```

In this case we use the mixin class `new_class` instead. The `new_class` class template has the following arguments,

- the final class we are defining
- the underlying type (`int`),
- a MPL sequence of additions `mpl::vector<opaque::using_totally_ordered1>`

Boost.Opaque provides a macro `BOOST_OPAQUE_NEW_CLASS` to simplify the preceding declaration

```
BOOST_OPAQUE_NEW_CLASS(Identifier, int, (opaque::using_totally_ordered1))
{
    void print();
    // ...
}
```

The macros is responsable to transform the preprocessor sequence into a MPL sequence.

Return type for relational operators: `opaque::boolean`

The following code doesn't warms in (1) when we use the integer `0` where a `bool` is expected.

```
typedef bool Bool;
typedef int Int;

Bool f(Int a, Int b)
{
    if(cnd()) return 0;       // 1
    else return a > b;
}
```

We can define an bool opaque typedef so only `true_` and `false_` are accepted as constant expressions

```
typedef int Int;
typedef opaque::boolean Bool;

Bool f(Int a, Int b)
{
    if(cnd()) return 0;   // (1) Error
    else  return a > b;   // (2) Error
}
```

Now we get an error on (1) as `0` is not convertible to `opaque::boolean`. This can be easily solved

```
Bool f(Int a, Int b)
{
    if(cnd()) return opaque::false_;    // OK
    else  return a > b;                 // (2) Error
}
```

But we also get an error on (2) as `a > b` is not convertible to `opaque::boolean`. To solve this issue we need that the signature `Int > Int` be convertible to `opaque::boolean`.

Boost.Opaque provides a meta-mixin that allow to define an integer-like class that use `opaque::boolean` as type for the result of the relational operators.

```
struct Int_tag;
typedef boost::opaque::new_type<int, Int_tag, boost::mpl::vector<
    opaque::using_integer_like<opaque::boolean>
> > Int;
```

With this definition, the preceding example compiles and does what we are expecting.

Or using the macro

```
BOOST_OPAQUE_NEW_TYPE(int, Int, (opaque::using_integer_like<opaque::boolean>))
```

Opaque typedefs

While the preceding examples restrict the operations of the underlying type, sometimes it is useful to define a different type that provides the same operations than the underlying type and that allows some conversions. In this case we use the opaque typedefs.

For example, we can need real new typedefs to be able to overload a function. This can be done with the `BOOST_OPAQUE_PUBLIC_TYPEDEF` macro.

```
// Listing 1
// Cartesian 3D coordinate types
BOOST_OPAQUE_PUBLIC_TYPEDEF(double, X);
BOOST_OPAQUE_PUBLIC_TYPEDEF(double, Y);
BOOST_OPAQUE_PUBLIC_TYPEDEF(double, Z);

// polar 3D coordinate types
BOOST_OPAQUE_PUBLIC_TYPEDEF(double, Rho);
BOOST_OPAQUE_PUBLIC_TYPEDEF(double, Theta);
BOOST_OPAQUE_PUBLIC_TYPEDEF(double, Phi);

class PhysicsVector {
public:
  PhysicsVector(X, Y, Z);
  PhysicsVector(Rho, Theta, Phi);
  ...
}; // PhysicsVector
```

`BOOST_OPAQUE_PUBLIC_TYPEDEF(double, X)` creates a new type `X` with the same operations than `double`. The operations to forward are obtained from the traits class `inherited_from_underlying<double>`. The template class can be specialized for a particular type and the library already do that for all the built-in types.

Explicit versus Implicit Conversions to the UT

Opaque typedef are explicitly convertible to the underlying type. But are they implicitly convertible? It depends. So the library provide two kind of opaque typedef :

- public opaque typedef: implicit conversion to the underlying type
- private opaque typedef: explict conversion to the underlying type

Hiding inherited operations

Sometimes we want to inherit most of the operations inherited from the underlying type. We can either use the `new_type` class and be explicit on which operation we want

```
BOOST_OPAQUE_NEW_TYPE(UT, NT,
    (opaque::using_ope1)
    (opaque::using_ope2)
    ...
    (opaque::using_open)
)
```

or we can use opaque types and state explicitly which operation will be hidden.

```
BOOST_OPAQUE_PUBLIC_OPAQUE_TYPE(UT, NT,
    (opaque::hiding_opeK)
)
```

Underlying types hierarchy

Opaque and underlying types form a hierarchy with a implicit or explicit conversion from the OT to the UT. This convertibility is a relation. It seems natural that this relation satisfy the transitivity property.

Using UDT as Underlying types

For example we can access the fisrt field of the UT pair, but the OT can not give this access. Instead the OT could define some function accessors that inderectly will almost behave as the data member.

```
struct UT {
    T member_;
};

UT ut;
ut.member_ = 88;


struct OT : new_class<OT,UT> {
    T& member() {
        return underlying().member_;
    }
    T member() const {
        return underlying().member_;
    }
};

OT ot;
//ot.member_=88 // compile-error
ot.member() = 88;
```

For pointer to members, we have the same problem

```
T UT::* pm = &UT::member_;
UT* utp;
utp->*pm = 88;
```

If UT had overloaded the pointer to member operator, OT should do the same think

```
struct OT : new_class<OT,UT> {
    T UT::* pmt(T UT::* pm) {
        return underlying().member_;
    }
    T member() const {
        return underlying().member_;
    }
    T& operator->*(T& (OT::* m)()) {

    }
};


T UT::* pm = &UT::member_;
T& (OT::* pm) = &OT::member;
UT* utp;
utp->*pm = 88;

OT* otp;
otp->*pm() = 88;
```

Examples

Identifier

One of the use cases that were at the origin of this library was to be able to define different identifiers that can be stored on a ordered container.

The following template IdentifierGenarator can be used for this propose

```
template <typename UT, typename Tag>
struct IdentifierGenarator
{
    typedef boost::opaque::new_type<UT, Tag,
        boost::mpl::vector<boost::opaque::using::less_than_comparable> type;
};
```

The class can be used as follows

```
struct ConnectionIdentifierTag {};
typedef IdentifierGenarator<int, ConnectionIdentifierTag>::type ConnectionIdentifier;
```

External Resources

N2141: Strong Typedefs in C++09(Revisited)	Alisdair Meredith
N1891: Progress toward Opaque Typedefs for C++0X	Walter E. Brown
N1706: Toward Opaque Typedefs in C++0X	Walter E. Brown
PC-lint/FlexeLint Strong Type Checking	Gimpel Software
True typedefs	Matthew Wilson

Reference

Header `<boost/opaque.hpp>`

Include all the opaque public header files.

```
#include <boost/opaque/opaque.hpp>
```

Header <boost/opaque/new_class.hpp>

```
namespace boost {
namespace opaque {
    class base_new_type;

    template <
        typename Final,
        typename T,
        typename MetaMixinSeq=boost::mpl::vector0<>,
        typename Base=base_new_type
    >
    class new_class;

    template <
        typename T,
        typename Final,
        typename UT,
        typename MetaMixinSeq,
        typename Base
    >
    T opaque_static_cast(new_class<Final, UT, MetaMixinSeq, Base> const& v);
}
}
```

Class Template base_new_type<>

```
class base_new_type {};
```

Class Template new_class<>

`new_class<>` wraps an underlying type providing the regular constructors and copy construction from the underlying type and all the convertibles to the the UT.

Resuming:

- Can instances of UT be explicitly converted to instances of OT? Yes
- Can instances of convertible to UT be explicitly converted to instances of OT? Yes
- Can instances of UT be implicitly converted to instances of OT? No
- Can instances of OT be explicitly converted to instances of UT? No
- Can instances of OT be implicitly converted to instances of UT? No

```
template <
    typename Final,
    typename UT,
    typename MetaMixinSeq,
    typename Base
>
class new_class : public linear_hierarchy<MetaMixinSeq, Final, Base>::type
{
public:
    typedef UT underlying_type;

    template <typename W>
    explicit new_class(W v);
    new_class();
    new_class(const new_class & rhs);
    explicit new_class(underlying_type v);
protected:
    T val_;
    new_class & operator=(const new_class & rhs);

public:
    underlying_type const& underlying() const;
    underlying_type& underlying();

    template<typename F>
    static underlying_type& underlying(F* f);

    template<typename F>
    static underlying_type const& underlying(F const* f);

    template<typename F>
    static Final const& final(F const* f);
    template<typename F>
    static Final& final(F* f);

};
```

Requirements

- `UT` must be a model of a `CopyConstructible` and `Assignable`.
- `Final` must be a model of FinalUnderlying.
- `MetaMixinSeq` must be a model of `MetaMixinSeq`: defaulted to the empty sequence.
- `Base` is defaulted to the base class of all the new types `base_new_type`.

Non-Member Function Template `opaque_static_cast<>`

```
template <
    typename T,
    typename Final,
    typename UT,
    typename MetaMixinSeq,
    typename Base
>
T opaque_static_cast(new_class<Final, UT, MetaMixinSeq, Base> const& v);
```

Requirements

- `T` and `UT` must be a model of a `CopyConstructible` and `Assignable`.

- `Final` must be a model of FinalUnderlying.
- `MetaMixinSeq` must be a model of `MetaMixinSeq`.

Header `<boost/opaque/new_type.hpp>`

```
namespace boost {
namespace opaque {
    template <
        typename UT,
        typename Tag,
        typename MetaMixinSeq=boost::mpl::vector0<>,
        typename Base=base_new_type>
    class new_type;
}
}
```

Class Template `new_type<>`

`new_type<>` provides the equivalent of `new_class<>` but can be used as a typedef.

```
template <
    typename UT,
    typename Tag,
    typename MetaMixinSeq,
    typename Base
class new_type
    : public new_class<new_type<UT, Tag, MetaMixinSeq, Base>, UT, MetaMixinSeq, Base>
{
public:
    template <typename W>
    explicit new_type(W v);
    new_type();
    new_type(const new_type & rhs);
    explicit new_type(UT v);
};
```

Requirements

- `UT` must be a model of a `CopyConstructible` and `Assignable`.
- `Tag` must be a unique class.
- `MetaMixinSeq` must be a model of `MetaMixinSeq`: defaulted to the empty sequence.
- `Base` is defaulted to the base class of all the new types `base_new_type`.

Non-Member Function Template Specialization `opaque_static_cast<>`

```
template <
    typename T,
    typename UT,
    typename Tag,
    typename MetaMixinSeq,
    typename Base
>
T opaque_static_cast(new_type<UT,Tag,MetaMixinSeq,Base> const& v);
```

Header <boost/opaque/boolean.hpp>

```
namespace boost {
    namespace opaque {
        class boolean;
    }
}
```

Class boolean

```
class boolean {
public:
    explicit boolean(const bool b);

    operator unspecified_bool_type() const;
    boolean operator!() const;
    boolean operator&&(boolean rhs) const;
    boolean operator||(boolean rhs) const;
};
const boolean true_;
const boolean false_;
```

Header <boost/opaque/private_opaque_class.hpp>

```
namespace boost {
namespace opaque {
    class base_private_opaque_type;
    template <
        typename Final,
        typename UT,
        typename MetaMixinSeq=boost::mpl::vector0<>,
        typename Base=base_private_opaque_type
    >
    class private_opaque_class;
}
}
```

Class base_private_opaque_type

This is the base class of all the private opaque types.

```
class base_private_opaque_type {};
```

Class Template private_opaque_class<>

Resuming:

- Can instances of UT be explicitly converted to instances of OT? Yes
- Can instances of convertibles to UT be explicitly converted to instances of OT? Yes
- Can instances of UT be implicitly converted to instances of OT? No
- Can instances of OT be explicitly converted to instances of UT? Yes.
- Can instances of OT be explicitly converted to instances of convertible to UT? Yes

- Can instances of OT be implicitly converted to instances of UT? No
- Can instances of OT be implicitly converted to instances of convertible to UT? No

Note

On compilers don't supporting explicit conversion operators, the explicit conversion must be done through the `underlying` function

`private_opaque_class<>` is a `new_class` with the `transitive_explicit_substituable<base_private_opaque_type>` and `inherited_from_underlying<UT>` added to the sequence of meta-mixins MetaMixinSeq, so a `private_opaque_class` inherits from all the operations of tye underlying type and adds transitive explicit conversions to all the substituables. The nested typedef `substituables` is the MPL sequence of all the UT in the opaque type hierarchy.

```
template <
    typename Final,
    typename UT,
    typename MetaMixinSeq,
    typename Base
>
class private_opaque_class : public
        new_class< Final, UT,
            mpl::push<
                transitive_explicit_substituable<base_private_opaque_type>,
                mpl::push<
                    inherited_from_underlying<UT>,
                    MetaMixinSeq
                >
            >,
            Base
        >
{
public:

    private_opaque_class();
    private_opaque_class(const private_opaque_class & rhs);
    private_opaque_class(const Final & rhs);
    explicit private_opaque_class(UT v);
    template <typename W>
    explicit private_opaque_class(W v);
    explicit operator UT() const;
};
```

Header <boost/opaque/private_opaque_type.hpp>

```
namespace boost {
namespace opaque {
    template <
        typename Final,
        typename UT,
        typename MetaMixinSeq=boost::mpl::vector0<>,
        typename Base=base_private_opaque_type
    >
    class private_opaque_type;
}
}
```

Class Template `private_opaque_type<>`

`private_opaque_type<>` provides the equivalent of `private_opaque_class<>` but can be used as a typedef.

```
template <
    typename Final,
    typename UT,
    typename MetaMixinSeq,
    typename Base
>
class private_opaque_type : public
    private_opaque_class<
        private_opaque_type<UT,Tag,MetaMixinSeq,Base>, UT, MetaMixinSeq, Base>
{
public:

    private_opaque_type();
    private_opaque_type(const private_opaque_class & rhs);
    private_opaque_type(const Final & rhs);
    explicit private_opaque_type(UT v);
    template <typename W>
    explicit private_opaque_type(W v);
};
```

Requirements

- `UT` must be a model of a `CopyConstructible` and `Assignable`.
- `Tag` must be a unique class.
- `MetaMixinSeq` must be a model of `MetaMixinSeq`: defaulted to the empty sequence.
- `Base` is defaulted to the base class of all the new types `base_new_type`.

Header `<boost/opaque/public_opaque_class.hpp>`

```
namespace boost {
namespace opaque {
    class base_public_opaque_type;

    template <typename T>
    struct get_substituables;

    template <
        typename Final,
        typename UT,
        typename MetaMixinSeq=boost::mpl::vector0<>,
        typename Base=base_public_opaque_type
    >
    class public_opaque_class;
}
}
```

Class `base_public_opaque_type`

This is the base class of all the public opaque types.

```
class base_public_opaque_type {};
```

Class Template `get_substituables<>`

```
template <typename T>
struct get_substituables {
    typedef <see below> type;
}
```

The nested typedef `type` is an MPL sequence of all the UT in the opaque type hierarchy.

Class Template `public_opaque_class<>`

Resuming:

- Can instances of UT be explicitly converted to instances of OT? Yes
- Can instances of UT be implicitly converted to instances of OT? No
- Can instances of OT be explicitly converted to instances of UT? Yes
- Can instances of OT be explicitly converted to instances of convertible to UT? Yes
- Can instances of OT be implicitly converted to instances of UT? Yes
- Can instances of OT be implicitly converted to instances of convertible to UT? Yes

`public_opaque_class<>` is a `new_class` with the `transitive_substituable<base_public_opaque_type>` and `inherited_from_underlying<UT>` added to the sequence of meta-mixins MetaMixinSeq, so a public_opaque_type inherits from all the operations of tye underlying type and adds transitive implicit conversions to all the substituables. The nested typedef `substituables` is the MPL sequence of all the UT in the opaque type hierarchy.

```
template <
    typename Final,
    typename UT,
    typename MetaMixinSeq=boost::mpl::vector0<>,
    typename Base=base_public_opaque_type
>
class public_opaque_class
    : public new_class< Final, UT,
        mpl::push<
            transitive_substituable<base_public_opaque_type>,
            mpl::push<
                inherited_from_underlying<UT>,
                MetaMixinSeq
            >
        >,
        Base
    >
{
public:
    typedef <see below> substituables;

    public_opaque_class();
    public_opaque_class(const public_opaque_class & rhs);
    public_opaque_class(const Final & rhs);
    explicit public_opaque_class(UT v);
    template <typename W>
    explicit public_opaque_class(W v);

};
```

Header <boost/opaque/public_opaque_type.hpp>

```
namespace boost {
namespace opaque {
    template <
        typename Final,
        typename UT,
        typename MetaMixinSeq=boost::mpl::vector0<>,
        typename Base=base_public_opaque_type
    >
    class public_opaque_type;
}
}
```

Class Template public_opaque_type<>

`public_opaque_type<>` provides the equivalent of `public_opaque_class<>` but can be used as a typedef.

```
template <
    typename Final,
    typename UT,
    typename MetaMixinSeq,
    typename Base
>
class public_opaque_type : public
    public_opaque_class<
        public_opaque_type<UT,Tag,MetaMixinSeq,Base>, UT, MetaMixinSeq, Base>
{
public:

    public_opaque_type();
    public_opaque_type(const public_opaque_type & rhs);
    public_opaque_type(const Final & rhs);
    explicit public_opaque_type(UT v);
    template <typename W>
    explicit public_opaque_type(W v);
};
```

Requirements

- `UT` must be a model of a `CopyConstructible` and `Assignable`.
- `Tag` must be a unique class.
- `MetaMixinSeq` must be a model of `MetaMixinSeq`: defaulted to the empty sequence.
- `Base` is defaulted to the base class of all the new types `base_new_type`.

Header <boost/opaque/macros.hpp>

Meta-Mixins

Concepts

Final

Classes that models the `Final` concept satisfy the following expressions:

Let `B` a base class of `Final`, `b` an instance of `B`, `bc` an instance of `B const`.

- `Final::final(&b)` return the `Final&` reference associated to `b`.
- `Final::final(&b)` returns the `const Final&` reference associated to `bc`.

FinalUnderlying

Classes that models the `FinalUnderlying` concept satisfy the following expressions:

Let `B` a base class of `Final`, `b` an instance of `B`, `bc` an instance of `B const`, `f` an instace of `Final` and `fc` and instance of `Final const '.

- `Final::underlying_type` the underlying type.
- `Final::underlying(&b)` return a reference to `Final::underlying_type&` associated to `b`.
- `Final::underlying(&bc)` return a constant reference to `Final::underlying_type const&` associated to `bc`.
- `f.underlying()` return a reference to `Final::underlying_type&` associated to `f`.
- `fc.underlying()` return a constant reference to `Final::underlying_type const&` associated to `fc`.

Mixin

A Mixin is a template class having two template parameters, the Final type and the Base type.

The archetype of a Mixin is

```
template <typename Final, typename Base>
struct MixinArchetype : Base
{
    ...
};
```

The `Final` class must satisfy the `FinalUnderlying` requirements.

MetaMixin

A MetaMixin is a meta-function having as nested type a Mixin. The archetype of a MetaMixin is

```
struct MetaMixinArchetype {
    template <typename Final, typename Base>
    struct type : Base
    {
        ...
    };
};
```

The `Final` class must satisfy the `FinalUnderlying` requirements.

MetaMixinSequence

A `MetaMixinSequence` is MPL Sequence of `MetaMixin`.

Header <boost/opaque/meta_mixin/linear_hierarchy.hpp>

```
namespace boost {
    template<typename MetaMixinSeq, typename Final, typename Base>
    struct linear_hierarchy;
}
```

Class Template linear_hierarchy<>

The `linear_hierarchy` metafunction gererates a linear hierarchy by folding the Mixins obtained by application of the `MetaMixins` in `MetaMixinSeq`.

```
template<typename MetaMixinSeq, typename Final, typename Base>
struct linear_hierarchy {
    typedef <see below> type;
};
```

The nested `type` is equivalent to `typename boost::mpl::fold<MetaMixinSeq, Base, implementation_defined<Final> >::type`.

Header <boost/opaque/meta_mixin/inherited_from_underlying.hpp>

```
namespace boost {
    template <typename T, typename Bool=bool>
    struct inherited_from_underlying;
}
```

Class Template inherited_from_underlying<>

```
template <typename T, typename Bool>
struct inherited_from_underlying;
```

`inherited_from_underlying` is a `MetaMixin` which adds wrapping member to the underlyibg type `UT`.

This class must be specialized for specific types in order to make easier the construction of opaque types having `UT` as underlying type. For example the library provide specializations for each one of the built-in types.

```
template <typename Bool>
struct inherited_from_underlying<int> {
    template <typename Final, typename Base>
    struct type; //implementation defined
};
```

Header <boost/opaque/meta_mixin/using_operators.hpp>

This file includes meta-mixins that are used to add an operator overload forwarding from the final type to the underlying type. There is a meta-mixin for each one of the C++ overloadable operators.

These meta-mixins have names that follows the naming used in Boost.ConceptsTraits, but prefixed by `using_`.

Arithmetic Operators

The arithmetic meta-mixins ease the task of creating a custom numeric type based on the underlying type. Given an underlying type, the templates add forward operators from the numeric class to the underlying type. These operations are like the ones the standard arithmetic types have, and may include comparisons, adding, incrementing, logical and bitwise manipulations, etc. Further, since

most numeric types need more than one of these operators, some templates are provided to combine several of the basic operator templates in one declaration.

The requirements for the types used to instantiate the simple operator templates are specified in terms of expressions which must be valid and the expression's return type.

These meta-mixins are "simple" since they provide un operator based on a single operation the underlying type has to provide. They have an additional optional template parameter Base, which is not shown, for the base class chaining technique.

In the following table the meta-mixin follows the schema

```
struct meta-mixin {
    template <typename NT, typename Base>
    struct type: Base {
        // Supplied Operation
    };
};
```

- `NT`/`NT2` is expected to satisfy the `FinalUnderlying` requirements.
- `UT` stands for `NT::underlying_type`.
- `UT2` stands for `NT2::underlying_type`.
- `this_ut` is the instance `UT` reference obtained `NT::underlying(this)`.
- `lhs` is a `NT`/`NT2 const` reference.
- `rhs` is a `NT`/`NT2 const` reference.
- `lhs_ut` is the instance `UT` reference obtained `lhs.underlying()`.
- `rhs_ut` is the instance `UT` reference obtained `rhs.underlying()`.

Table 1. Relational operators

Meta-Mixin	Supplied Operation	Requirements	Ref
`using_equal<Bool>`	`Bool operator==(const NT& rhs) const`	`Bool(this_ut == rhs_ut)`	13.5.2
`using_not_equal<Bool>`	`Bool operator!=(const NT& rhs) const`	`Bool(this_ut != rhs_ut)`	13.5.2
`using_less_than<Bool>`	`Bool operator<(const NT& rhs) const`	`Bool(this_ut < rhs_ut)`	13.5.2
`using_less_than_equal<Bool>`	`Bool operator<=(const NT& rhs) const`	`Bool(this_ut <= rhs_ut)`	13.5.2
`using_greater_than_equal<Bool>`	`Bool operator>=(const NT& rhs) const`	`Bool(this_ut >= rhs_ut)`	13.5.2
`using_greater_than<Bool>`	`Bool operator>(const NT& rhs) const`	`Bool(this_ut > rhs_ut)`	13.5.2
`using_equal2<NT2, Bool>`	`Bool operator==(const NT2& rhs) const`	`Bool(this_ut == rhs_ut)`	13.5.2
`using_not_equal2<NT2, Bool>`	`Bool operator!=(const NT2& rhs) const`	`Bool(this_ut != rhs_ut)`	13.5.2
`using_less_than2<NT2, Bool>`	`Bool operator<(const NT& rhs) const`	`Bool(this_ut < rhs_ut)`	13.5.2
`using_less_than_equal2<NT2, Bool>`	`Bool operator<=(const NT2& rhs) const`	`Bool(this_ut <= rhs_ut)`	13.5.2
`using_greater_than_equal2<NT2, Bool>`	`Bool operator>=(const NT2& rhs) const`	`Bool(this_ut >= rhs_ut)`	13.5.2
`using_greater_than2<NT2, Bool>`	`Bool operator>(const NT2& rhs) const`	`Bool(this_ut > rhs_ut)`	13.5.2

Table 2. Numeric operators

Meta-Mixin	Supplied Operation	Requirements	Ref
`using_plus`	`NT operator+(const NT& rhs) const`	`NT(this_ut + rhs_ut)`	13.5.2
`using_plus_assign`	`NT& operator+=(const NT& rhs)`	`this_ut += rhs_ut`	13.5.2
`using_minus`	`NT operator-(const NT& rhs) const`	`NT(this_ut - rhs_ut)`	13.5.2
`using_minus_assign`	`NT& operator-=(const NT& rhs)`	`this_ut -= rhs_ut`	13.5.2
`using_multiply`	`NT operator*(const NT& rhs) const`	`NT(this_ut * rhs_ut)`	13.5.2
`using_multiply_assign`	`NT& operator*=(const NT& rhs)`	`this_ut *= rhs_ut`	13.5.2
`using_divide`	`NT operator/(const NT& rhs) const`	`NT(this_ut / rhs_ut)`	13.5.2
`using_divide_assign`	`NT& operator/=(const NT& rhs)`	`this_ut /= rhs_ut`	13.5.2
`using_modulus`	`NT operator%(const NT& rhs) const`	`NT(this_ut % rhs_ut)`	13.5.2
`using_modulus_assign`	`NT& operator%=(const NT& rhs)`	`this_ut %= rhs_ut`	13.5.2
`using_unary_plus`	`NT operator+() const`	`NT(+this_ut)`	13.5.1
`using_unary_minus`	`NT operator-() const`	`NT(-this_ut)`	13.5.1
`using_pre_increment`	`NT& operator++()`	`++this_ut`	13.5.1
`using_pre_decrement`	`NT& operator--()`	`--this_ut`	13.5.1
`using_post_increment`	`NT operator++(int) const`	`this_ut++`	13.5.1
`using_post_decrement`	`NT operator--(int) const`	`this_ut++`	13.5.1
`using_plus2<NT2>`	`NT operator+(const NT2& rhs) const`	`NT(this_ut + rhs_ut)`	13.5.2
`using_plus_assign2<NT2>`	`NT& operator+=(const NT2& rhs)`	`this_ut += rhs_ut`	13.5.2
`using_minus2<NT2>`	`NT operator-(const NT2& rhs) const`	`NT(this_ut - rhs_ut)`	13.5.2
`using_minus_assign2<NT2>`	`NT& operator-=(const NT2& rhs)`	`this_ut -= rhs_ut`	13.5.2
`using_multiply2<NT2>`	`NT operator*(const NT2& rhs) const`	`NT(this_ut * rhs_ut)`	13.5.2
`using_multiply_assign2<NT2>`	`NT& operator*=(const NT2& rhs)`	`this_ut *= rhs_ut`	13.5.2
`using_divide2<NT2>`	`NT operator/(const NT2& rhs) const`	`NT(this_ut / rhs_ut)`	13.5.2
`using_divide_assign2<NT2>`	`NT& operator/=(const NT2& rhs)`	`this_ut /= rhs_ut`	13.5.2
`using_modulus2<NT2>`	`NT operator%(const NT2& rhs) const`	`NT(this_ut % rhs_ut)`	13.5.2
`using_modulus_assign2<NT2>`	`NT& operator%=(const NT2& rhs)`	`this_ut %= rhs_ut`	13.5.2

Table 3. Bitwise operators

Meta-Mixin	Supplied Operation	Requirements	Ref
using_bitwise_not	NT operator~() const	NT(~this_ut)	13.5.1
using_bitwise_xor_assign	NT& operator^=(const NT& rhs)	this_ut ^= rhs_ut	13.5.2
using_bitwise_and_assign	NT& operator&=(const NT& rhs)	this_ut &= rhs_ut	13.5.2
using_bitwise_or_assign	NT& operator|=(const NT& rhs)	this_ut |= rhs_ut	13.5.2
using_left_right_assign	NT& operator<<=(const NT& rhs)	this_ut <<= rhs_ut	13.5.2
using_right_shift_assign	NT& operator>>=(const NT& rhs)	this_ut >>= rhs_ut	13.5.2
using_bitwise_xor	NT operator^(const NT& rhs) const	NT(this_ut ^ rhs_ut)	13.5.2
using_bitwise_or	NT operator|(const NT& rhs) const	NT(this_ut | rhs_ut)	13.5.2
using_bitwise_and	NT operator&(const NT& rhs) const	NT(this_ut & rhs_ut)	13.5.2
using_left_shift1	NT operator<<(const NT& rhs) const	NT(this_ut << rhs_ut)	13.5.2
using_right_shift1	NT operator>>(const NT& rhs) const	NT(this_ut >> rhs_ut)	13.5.2
using_bitwise_xor2<NT2>	NT operator^(const NT2& rhs) const	NT(this_ut ^ rhs_ut)	13.5.2
using_bitwise_xor_assign2<NT2>	NT& operator^=(const NT2& rhs)	this_ut ^= rhs_ut	13.5.2
using_bitwise_or2<NT2>	NT operator|(const NT2& rhs) const	NT(this_ut | rhs_ut)	13.5.2
using_bitwise_and2<NT2>	NT operator&(const NT2& rhs) const	NT(this_ut & rhs_ut)	13.5.2
using_bitwise_and_assign2<NT2>	NT& operator&=(const NT2& rhs)	this_ut &= rhs_ut	13.5.2
using_bitwise_or_assign2<NT2>	NT& operator|=(const NT2& rhs)	this_ut |= rhs_ut	13.5.2
using_left_shift2<NT2>	NT operator<<(const NT2& rhs) const	NT(this_ut << rhs_ut)	13.5.2
using_left_right_assign2<NT2>	NT& operator<<=(const NT2& rhs)	this_ut <<= rhs_ut	13.5.2
using_right_shift2<NT2>	NT operator>>(const NT2& rhs) const	NT(this_ut >> rhs_ut)	13.5.2
using_right_shift_assign2<NT2>	NT& operator>>=(const NT2& rhs)	this_ut >>= rhs_ut	13.5.2

Logical Operators

Table 4. Logical operators

Meta-Mixin	Supplied Operation	Requirements	Ref
`using_logical_not<Bool>`	`Bool operator!() const`	`Bool(!this_ut)`	13.5.1
`using_logical_and<Bool>`	`Bool operator&&(const NT& rhs) const`	`Bool(this_ut && rhs_ut)`	13.5.2
`using_logical_or<Bool>`	`Bool operator||(const NT& rhs) const`	`Bool(this_ut || rhs_ut)`	13.5.2

Conversion Operators

Table 5. Conversion Operators

Meta-Mixin	Supplied Operation	Requirements	Ref
`using_underlying_conversion<UT>`	`operator UT() const`	None	12.3.2
`using_conversion_to<T>`	`operator T() const`	`T(ut)`	12.3.2
`using_safe_bool_conversion`	`operator unspecified_bool_type() const`	`ut?true:false`	12.3.2

Dereference Operators

Table 6. Dereference Operators

Meta-Mixin	Supplied Operation	Requirements	Ref
`using_address_of`	`NT& operator&()`	??	
`using_address_of2<A>`	`A operator&()`	??	
`using_derreference`	`NT& operator*()`	??	
`using_derreference<R>`	`R operator*()`	??	
`using_member_access<P>`	`P operator->()`	??	
`using_subscript<D,T>`	`T operator[](D)`	??	

Other Operators

Meta-Mixin	Supplied Operation	Requirements	Ref
`using_comma`	`NT operator,()`	??	
`using_function_call`	`operator()()`	??	
`using_pointer_to_member`	`operator->*()`	??	

Header <boost/opaque/meta_mixin/hiding_operators.hpp>

This file includes meta-mixins that are used to hide operator overloads that have been defined by a base class for each one of the C++ overloadable operators.

These meta-mixins have names that follows the naming used in Boost.ConceptsTraits, but prefixed by `hiding_`.

Meta-Mixin	Hidden Operation	Requirements	Ref
`hiding_assignment`	`NT& operator=(NT const&)`	None	13.5.3
`hiding_copy_constructor`	`NT(NT const&)`	None	13.5.3

Header <boost/opaque/meta_mixin/using_combined_operators.hpp>

This file includes meta-mixins combining several operators.

These meta-mixins have names that follows the naming used in Boost.Operators, but prefixed by `using_`.

The composite operator templates only list what other templates they use. The supplied operations and requirements of the composite operator templates can be inferred from the operations and requirements of the listed components.

Grouped Arithmetic Operators

The following meta-mixins provide common groups of related meta-mixins. For example, since a type which is addable is usually also subractable, the additive template provides the combined meta-mixins of both.

Meta-Mixin	Meta-Mixin Sequence	Ref
`using_equality_comparable1<Bool>`	`using_equal<Bool>,` `using_not_equal<Bool>`	20.1.1

`using_less_than_comparable1<Bool>`	`using_less_than<Bool>`
	`using_less_than_equal<Bool>`
	`using_greater_than_equal<Bool>`
	`using_greater_than<Bool>`
		20.1.2

`using_partially_ordered1<Bool>`	`using_less_than<Bool>`
	`using_less_than_equal<Bool>`
	`using_greater_than_equal<Bool>`
	`using_greater_than<Bool>`
		20.1.2

`using_addable1`	`using_plus_assign` `using_plus`	
`using_subtractable1`	`using_minus_assign` `using_minus`	
`using_multipliable1`	`using_multiply_assign` `using_multiply`	
`using_dividable1`	`using_divide_assign` `using_divide`	
`using_modable1`	`using_modulus_assign` `using_modulus`	
`using_bitwise_xorable1`	`using_bitwise_xor_assign` `using_bitwise_xor`	
`using_bitwise_orable1`	`using_bitwise_or_assign` `using_bitwise_or`	
`using_bitwise_andable1`	`using_bitwise_and_assign` `using_bitwise_and`	
`using_left_shiftable1`	`using_left_right_assign` `using_left_shift1`	
`using_right_shiftable1`	`using_right_shift_assign` `using_right_shift1`	
`using_incrementable`	`using_pre_increment>` `using_post_increment`	
`using_decrementable`	`using_pre_decrement>` `using_post_decrement`	

Meta-Mixin	Meta-Mixin Sequence	Ref
`using_totally_ordered1<Bool>`	`using_equality_comparable1<Bool>` `using_less_than_comparable1<Bool>`	
`using_additive1`	`using_addable1` `using_subtractable1`	
`using_multiplicative1`	`using_multipliable1` `using_dividable1`	
`integer_multiplicative1`	`using_multiplicative1` `using_modable1`	
`using_arithmetic1`	`using_additive1` `using_multiplicative1`	
`using_integer_arithmetic1`	`using_additive1` `integer_multiplicative1`	

`using_bitwise1`	`using_bitwise_xorable1`
	`using_bitwise_orable1`
	`using_bitwise_andable1`	

`using_unit_steppable`	`using_incrementable` `using_decrementable`	
`using_shiftable1`	`using_left_shiftable1` `using_right_shiftable1`	
`using_ring_operators1`	`using_additive1` `using_multipliable1`	
`using_ordered_ring_operators1<Bool>`	`using_ring_operators1` `using_totally_ordered1<Bool>`	
`using_field_operators1<Bool>`	`using_ring_operators1` `using_dividable1<Bool>`	
`using_ordered_field_operators1<Bool>`	`using_field_operators1` `using_totally_ordered1<Bool>`	

Header <boost/opaque/meta_mixin/hiding_combined_operators.hpp>

This file includes meta-mixins combining several hiding meta-mixins that are used to hide operator overloads that have been defined by a base clas.

These meta-mixins have names that follows the naming used in Boost.Operators, but prefixed by `hiding_`.

Header <boost/opaque/meta_mixin/transitive_substituable.hpp>

```
namespace boost {
    template <typename BaseClass, typename UT>
    struct transitive_substituable;
}
```

Class Template `transitive_substituable<>`

This meta-mixin provides the Final class implicit conversions for all the underlying types hierarchy.

```
template <typename BaseClass, typename UT>
struct transitive_substituable;
```

Header `<boost/opaque/meta_mixin/transitive_explicit_substituable.hpp>`

```
namespace boost {
    template <typename BaseClass, typename UT>
    struct transitive_explicit_substituable;
}
```

Class Template `transitive_explicit_substituable<>`

```
template <typename BaseClass, typename UT>
struct transitive_explicit_substituable;
```

This meta-mixin provides the Final class explicit conversions for all the underlying types hierarchy (when the compiler supports explicit conversion operators). For portability purposed the library provide as workarround a convert_to non member function.

Appendices

Appendix A: History

Version 0.1.1, Febraury 18, 2011

Tests:

- Test pass on Ubuntu 10.10 for
- GCC 4.4.5
- GCC 4.4.5 -std=c++0x
- GCC 4.5.1
- GCC 4.5.1 -std=c++0x
- clang 2.8

Version 0.1.0, October 18, 2010

Initial version.

Features:

- a

Appendix B: Design Rationale

lala

Appendix C: Implementation Notes

lala

Appendix D: Acknowledgements

Thanks to .

Appendix E: Tests

new_class

Table 7. Contructors and Assignement

Name	kind	Description	Result
regular_pass	run	check constructors and assignements	Pass
assign_siblings_fail	compile-fail	check siblings assignement fail	Pass
assign_up_fail	compile-fail	check NT can not be assigned to UT	Pass
assign_down_fail	compile-fail	check UT can not be assigned to NT	Pass
copy_construct_from_non_convertible_fail	compile-fail	check constructor from non convertibe to UT fails	Pass

Table 8. Relational Operators

Name	kind	Description	Result
using_equal_pass	run	check `oparator==` is available when adding `using_equal` meta-mixin	Pass
using_not_equal_pass	run	check `oparator!=` is available when adding `using_not_equal` meta-mixin	Pass
using_less_than_pass	run	check `oparator<` is available when adding `using_less_than` meta-mixin	Pass
using_less_than_equal_pass	run	check `oparator<=` is available when adding `using_less_than_equal` meta-mixin	Pass
using_greater_than_pass	run	check `oparator>` is available when adding `using_greater_than` meta-mixin	Pass
using_greater_than_equal_pass	run	check `oparator>=` is available when adding `using_greater_than_equal` meta-mixin	Pass
equal_fail	compile-fail	check check `oparator==` fail on default new_class	Pass
not_equal_fail	compile-fail	check check `oparator!=` fail on default new_class	Pass
less_than_equal_fail	compile-fail	check check `oparator<=` fail on default new_class	Pass
less_than_fail	compile-fail	check check `oparator<` fail on default new_class	Pass
greater_than_equal_fail	compile-fail	check check `oparator>=` fail on default new_class	Pass
greater_than_fail	compile-fail	check check `oparator>` fail on default new_class	Pass

Table 9. Arithmetic Operators

Name	kind	Description	Result
using_plus_pass	run	check `oparator+` is available when adding `using_plus` meta-mixin	Pass
using_plus_assign_pass	run	check `oparator+=` is available when adding `using_plus_assign` meta-mixin	Pass
using_minus_pass	run	check `oparator-` is available when adding `using_minus` meta-mixin	Pass
using_minus_assign_pass	run	check `oparator-=` is available when adding `using_minus_assign` meta-mixin	Pass
using_multiply_pass	run	check `oparator*` is available when adding `using_multiply` meta-mixin	Pass
using_multiply_assign_pass	run	check `oparator*=` is available when adding `using_multiply_assign` meta-mixin	Pass
using_divide_pass	run	check `oparator/` is available when adding `using_divide` meta-mixin	Pass
using_divide_assign_pass	run	check `oparator/=` is available when adding `using_divide_assign` meta-mixin	Pass
using_modulus_pass	run	check `oparator%` is available when adding `using_modulus` meta-mixin	Pass
using_modulus_assign_pass	run	check `oparator%=` is available when adding `using_modulus_assign` meta-mixin	Pass
using_unary_plus_pass	run	check `oparator+()` is available when adding `using_unary_plus` meta-mixin	Pass
using_unary_minus_pass	run	check `oparator-()` is available when adding `using_unary_minus` meta-mixin	Pass
using_pre_increment_pass	run	check `oparator++()` is available when adding `using_pre_increment` meta-mixin	Pass
using_post_increment_pass	run	check `oparator++(int)` is available when adding `using_post_increment` meta-mixin	Pass
using_pre_decrement_pass	run	check `oparator--()` is available when adding `using_pre_decrement` meta-mixin	Pass
using_post_decrement_pass	run	check `oparator--(int)` is available when adding `using_post_decrement` meta-mixin	Pass

Table 10. Bitwise Operators

Name	kind	Description	Result
using_bitwise_not_pass	run	check `oparator~` is available when adding `using_bitwise_not` meta-mixin	Pass
using_bitwise_xor_pass	run	check `oparator^` is available when adding `using_bitwise_xor` meta-mixin	Pass
using_bitwise_xor_assign_pass	run	check `oparator^=` is available when adding `using_bitwise_xor_assign` meta-mixin	Pass
using_bitwise_or_pass	run	check `oparator|` is available when adding `using_bitwise_or` meta-mixin	Pass
using_bitwise_or_assign_pass	run	check `oparator|=` is available when adding `using_bitwise_or_assign` meta-mixin	Pass
using_bitwise_and_pass	run	check `oparator&` is available when adding `using_bitwise_and` meta-mixin	Pass
using_bitwise_and_assign_pass	run	check `oparator&=` is available when adding `using_bitwise_and_assign` meta-mixin	Pass
using_left_shift_pass	run	check `oparator<<` is available when adding `using_left_shift` meta-mixin	Pass
using_left_shift_assign_pass	run	check `oparator<<=` is available when adding `using_left_shift_assign` meta-mixin	Pass
using_right_shift_pass	run	check `oparator>>` is available when adding `using_right_shift` meta-mixin	Pass
using_right_shift_assign_pass	run	check `oparator>>=` is available when adding `using_right_shift_assign` meta-mixin	Pass

Table 11. Logical Operators

Name	kind	Description	Result
using_logical_not_pass	run	check `oparator!` is available when adding `using_logical_not` meta-mixin	Pass
using_logical_and_pass	run	check `oparator&&` is available when adding `using_logical_and` meta-mixin	Pass
using_logical_or_pass	run	check `oparator||` is available when adding `using_logical_and` meta-mixin	Pass

new_type

Name	kind	Description	Result
regular.pass	run	check constructors and assignements	Pass

public_opaque_class

Name	kind	Description	Result
regular.pass	run	check constructors and assignements	Pass

Appendix F: Tickets

Appendix F: Future plans

Tasks to do before review

- Complete the tests
- Add hiding meta-mixins