Boost.TypeErasure

Steven Watanabe

Table of Contents

Introduction

The Boost.TypeErasure library provides runtime polymorphism in C++ that is more flexible than that provided by the core language.

C++ has two distinct kinds of polymorphism, virtual functions and templates, each of which has its own advantages and disadvantages.

- Virtual functions are not resolved until runtime, while templates are always resolved at compile time. If your types can vary at runtime (for example, if they depend on user input), then static polymorphism with templates doesn't help much.
- Virtual functions can be used with separate compilation. The body of a template has to be available in every translation unit in which it is used, slowing down compiles and increasing rebuilds.
- Virtual functions automatically make the requirements on the arguments explicit. Templates are only checked when they're instantiated, requiring extra work in testing, assertions, and documentation.
- The compiler creates a new copy of each function template every time it is instantiated. This allows better optimization, because the compiler knows everything statically, but it also causes a significant increase of binary sizes.
- Templates support Value semantics. Objects that "behave like an int" and are not shared are easier to reason about. To use virtual functions, on the other hand, you have to use (smart) pointers or references.
- Template libraries can allow third-party types to be adapted non-intrusively for seamless interoperability. With virtual functions, you have to create a wrapper that inherits from the base class.
- Templates can handle constraints involving multiple types. For example, std::for_each takes an iterator range and a function that can be called on the elements of the range. Virtual functions aren't really able to express such constraints.

The Boost.TypeErasure library combines the superior abstraction capabilities of templates, with the runtime flexibility of virtual functions.

Boost includes several special cases of this kind of polymorphism:

- `boost::any` for CopyConstructible types.
- `boost::function` for objects that can be called like functions.
- Boost.Range provides `any_iterator`.

Boost.TypeErasure generalizes this to support arbitrary requirements and provides a predefined set of common concepts

How to read this documentation

To avoid excessive verbosity, all the examples assume that a few using directives are in place.

```
namespace mpl = boost::mpl;
using namespace boost::type_erasure;
```

Basic Usage

(For the source of the examples in this section see basic.cpp)

The main class in the library is `any`. An `any` can store objects that meet whatever requirements we specify. These requirements are passed to `any` as an MPL sequence.

Note

The MPL sequence combines multiple concepts. In the rare case when we only want a single concept, it doesn't need to be wrapped in an MPL sequence.

```
any<mpl::vector<copy_constructible<>, typeid_<>, relaxed> > x(10);
int i = any_cast<int>(x); // i == 10
```

`copy_constructible` is a builtin concept that allows us to copy and destroy the object. `typeid_` provides run-time type information so that we can use `any_cast`. `relaxed` enables various useful defaults. Without `relaxed`, `any` supports *exactly* what you specify and nothing else. In particular, it allows default construction and assignment of `any`.

Now, this example doesn't do very much. `x` is approximately equivalent to a boost::any. We can make it more interesting by adding some operators, such as `operator++` and `operator<<`.

```
any<
    mpl::vector<
        copy_constructible<>,
        typeid_<>,
        incrementable<>,
        ostreamable<>
    >
> x(10);
++x;
std::cout << x << std::endl; // prints 11
```

The library provides concepts for most C++ operators, but this obviously won't cover all use cases; we often need to define our own requirements. Let's take the `push_back` member, defined by several STL containers.

```
BOOST_TYPE_ERASURE_MEMBER((has_push_back), push_back, 1)

void append_many(any<has_push_back<void(int)>, _self&> container) {
    for(int i = 0; i < 10; ++i)
        container.push_back(i);
}
```

We use the macro `BOOST_TYPE_ERASURE_MEMBER` to define a concept called `has_push_back`. The second parameter is the name of the member function and the last macro parameter indicates the number of arguments which is `1` since `push_back` is unary. When we use `has_push_back`, we have to tell it the signature of the function, `void(int)`. This means that the type we store in the any has to have a member that looks like:

```
void push_back(int);
```

Thus, we could call `append_many` with `std::vector<int>`, `std::list<int>`, or `std::vector<long>` (because `int` is convertible to `long`), but not `std::list<std::string>` or `std::set<int>`.

Also, note that `append_many` has to operate directly on its argument. It cannot make a copy. To handle this we use `_self&` as the second argument of `any`. `_self` is a `placeholder`. By using `_self&`, we indicate that the `any` stores a reference to an external object instead of allocating its own object.

There's actually another `placeholder` here. The second parameter of `has_push_back` defaults to `_self`. If we wanted to define a const member function, we would have to change it to `const _self`, as shown below.

```
BOOST_TYPE_ERASURE_MEMBER((has_empty), empty, 0)
bool is_empty(any<has_empty<bool(), const _self>, const _self&> x) {
    return x.empty();
}
```

For free functions, we can use the macro `BOOST_TYPE_ERASURE_FREE`.

```
BOOST_TYPE_ERASURE_FREE((has_getline), getline, 2)
std::vector<std::string> read_lines(any<has_getline<bool(_self&, std::string&)>, _self&> stream)
{
    std::vector<std::string> result;
    std::string tmp;
    while(getline(stream, tmp))
        result.push_back(tmp);
    return result;
}
```

The use of `has_getline` is very similar to `has_push_back` above. The difference is that the placeholder `_self` is passed in the function signature instead of as a separate argument.

The `placeholder` doesn't have to be the first argument. We could just as easily make it the second argument.

```
void read_line(any<has_getline<bool(std::istream&, _self&)>, _self&> str)
{
    getline(std::cin, str);
}
```

Composing Concepts

(For the source of the examples in this section see compose.cpp)

Multiple concepts can be composed using an MPL sequence.

```
template<class T = _self>
struct arithmetic :
    mpl::vector<
        copy_constructible<T>,
        addable<T>,
        subtractable<T>,
        multipliable<T>,
        dividable<T>,
        equality_comparable<T>,
        less_than_comparable<T>
    >
{};
```

Now, `arithmetic` is a concept that can be used just like any of the base concepts.

Functions with Multiple Arguments

(For the source of the examples in this section see multi.cpp)

Operations can have more than one `any` argument. Let's use binary addition as an example.

```
typedef any<
    mpl::vector<
        copy_constructible<>,
        typeid_<>,
        addable<>,
        ostreamable<>
    >
> any_type;
any_type x(10);
any_type y(7);
any_type z(x + y);
std::cout << z << std::endl; // prints 17
```

This is *not* a multimethod. The underlying types of the arguments of + must be the same or the behavior is undefined. This example is correct because the arguments both hold `int`'s.

Note

Adding `relaxed` leads an exception rather than undefined behavior if the argument types are wrong.

`addable`<> requires the types of the arguments to be exactly the same. This doesn't cover all uses of addition though. For example, pointer arithmetic takes a pointer and an integer and returns a pointer. We can capture this kind of relationship among several types by identifying each type involved with a placeholder. We'll let the placeholder _a represent the pointer and the placeholder _b represent the integer.

```
int array[5];

typedef mpl::vector<
    copy_constructible<_a>,
    copy_constructible<_b>,
    typeid_<_a>,
    addable<_a, _b, _a>
> requirements;
```

Our new concept, `addable<_a, _b, _a>` captures the rules of pointer addition: `_a + _b -> _a`.

Also, we can no longer capture the variables independently.

```
any<requirements, _a> ptr(&array[0]); // illegal
```

This doesn't work because the library needs to know the type that _b binds to when it captures the concept bindings. We need to specify the bindings of both placeholders when we construct the `any`.

```
typedef mpl::map<mpl::pair<_a, int*>, mpl::pair<_b, int> > types;
any<requirements, _a> ptr(&array[0], make_binding<types>());
any<requirements, _b> idx(2, make_binding<types>());
any<requirements, _a> x(ptr + idx);
// x now holds array + 2
```

Now that the arguments of + aren't the same type, we require that both arguments agree that _a maps to `int*` and that _b maps to `int`.

We can also use `tuple` to avoid having to write out the map out explicitly. `tuple` is just a convenience class that combines the placeholder bindings it gets from all its arguments.

```
tuple<requirements, _a, _b> t(&array[0], 2);
any<requirements, _a> y(get<0>(t) + get<1>(t));
```

Concepts in Depth

Defining Custom Concepts

(For the source of the examples in this section see custom.cpp)

Earlier, we used `BOOST_TYPE_ERASURE_MEMBER` to define a concept for containers that support `push_back`. Sometimes this interface isn't flexible enough, however. The library also provides a lower level interface that gives full control of the behavior. Let's take a look at what we would need in order to define `has_push_back`. First, we need to define the `has_push_back` template itself. We'll give it two template parameters, one for the container and one for the element type. This template must have a static member function called apply which is used to execute the operation.

```
template<class C, class T>
struct has_push_back
{
    static void apply(C& cont, const T& arg) { cont.push_back(arg); }
};
```

Now, we can use this in an `any` using `call` to dispatch the operation.

```
std::vector<int> vec;
any<has_push_back<_self, int>, _self&> c(vec);
int i = 10;
call(has_push_back<_self, int>(), c, i);
// vec is [10].
```

Our second task is to customize `any` so that we can call `c.push_back(10)`. We do this by specializing `concept_interface`. The first argument is `has_push_back`, since we want to inject a member into every `any` that uses the `has_push_back` concept. The second argument, `Base`, is used by the library to chain multiple uses of `concept_interface` together. We have to inherit from it publicly. `Base` is also used to get access to the full `any` type. The third argument is the placeholder that represents this any. If someone used `push_back<_c, _b>`, we only want to insert a `push_back` member in the container, not the value type. Thus, the third argument is the container placeholder.

When we define `push_back` the argument type uses the metafunction `as_param`. This is just to handle the case where `T` is a placeholder. If `T` is not a placeholder, then the metafunction just returns its argument, `const T&`, unchanged.

```
namespace boost {
namespace type_erasure {
template<class C, class T, class Base>
struct concept_interface<has_push_back<C, T>, Base, C> : Base
{
    void push_back(typename as_param<Base, const T&>::type arg)
    { call(has_push_back<C, T>(), *this, arg); }
};
}
}
```

Our example now becomes

```
std::vector<int> vec;
any<has_push_back<_self, int>, _self&> c(vec);
c.push_back(10);
```

which is what we want.

Overloading

(For the source of the examples in this section see overload.cpp)

`concept_interface` allows us to inject arbitrary declarations into an `any`. This is very flexible, but there are some pitfalls to watch out for. Sometimes we want to use the same concept several times with different parameters. Specializing `concept_interface` in a way that handles overloads correctly is a bit tricky. Given a concept foo, we'd like the following to work:

```
any<
    mpl::vector<
        foo<_self, int>,
        foo<_self, double>,
        copy_constructible<>
    >
> x = ...;
x.foo(1);   // calls foo(int)
x.foo(1.0); // calls foo(double)
```

Because `concept_interface` creates a linear inheritance chain, without some extra work, one overload of foo will hide the other.

Here are the techniques that I found work reliably.

For member functions I couldn't find a way to avoid using two specializations.

```
template<class T, class U>
struct foo
{
    static void apply(T& t, const U& u) { t.foo(u); }
};

namespace boost {
namespace type_erasure {

template<class T, class U, class Base, class Enable>
struct concept_interface< ::foo<T, U>, Base, T, Enable> : Base
{
    typedef void _fun_defined;
    void foo(typename as_param<Base, const U&>::type arg)
    {
        call(::foo<T, U>(), *this, arg);
    }
};

template<class T, class U, class Base>
struct concept_interface< ::foo<T, U>, Base, T, typename Base::_fun_defined> : Base
{
    using Base::foo;
    void foo(typename as_param<Base, const U&>::type arg)
    {
        call(::foo<T, U>(), *this, arg);
    }
};

}
}
```

This uses SFINAE to detect whether a using declaration is needed. Note that the fourth argument of `concept_interface` is a dummy parameter which is always void and is intended to be used for SFINAE. Another solution to the problem that I've used in the past is to inject a dummy declaration of `fun` and always put in a using declaration. This is an inferior solution for several reasons.

It requires an extra interface to add the dummy overload. It also means that `fun` is always overloaded, even if the user only asked for one overload. This makes it harder to take the address of fun.

Note that while using SFINAE requires some code to be duplicated, the amount of code that has to be duplicated is relatively small, since the implementation of `concept_interface` is usually a one liner. It's a bit annoying, but I believe it's an acceptable cost in lieu of a better solution.

For free functions you can use inline friends.

```
template<class T, class U>
struct bar_concept
{
    static void apply(T& t, const U& u) { bar(t, u); }
};

namespace boost {
namespace type_erasure {

template<class T, class U, class Base>
struct concept_interface< ::bar_concept<T, U>, Base, T> : Base
{
    friend void bar(typename derived<Base>::type& t, typename as_param<Base, const U&>::type u)
    {
        call(::bar_concept<T, U>(), t, u);
    }
};

template<class T, class U, class Base>
struct concept_interface< ::bar_concept<T, U>, Base, U, typename boost::disable_if<is_placehold↵
er<T> >::type> : Base
{
    using Base::bar;
    friend void bar(T& t, const typename derived<Base>::type& u)
    {
        call(::bar_concept<T, U>(), t, u);
    }
};

}
}
```

Basically we have to specialize `concept_interface` once for each argument to make sure that an overload is injected into the first argument that's a placeholder. As you might have noticed, the argument types are a bit tricky. In the first specialization, the first argument uses `derived` instead of `as_param`. The reason for this is that if we used `as_param`, then we could end up violating the one definition rule by defining the same function twice. Similarly, we use SFINAE in the second specialization to make sure that bar is only defined once when both arguments are placeholders. It's possible to merge the two specializations with a bit of metaprogramming, but unless you have a lot of arguments, it's probably not worth while.

Concept Maps

(For the source of the examples in this section see concept_map.cpp)

Sometimes it is useful to non-intrusively adapt a type to model a concept. For example, suppose that we want to make `std::type_info` model `less_than_comparable`. To do this, we simply specialize the concept definition.

```
namespace boost {
namespace type_erasure {

template<>
struct less_than_comparable<std::type_info>
{
    static bool apply(const std::type_info& lhs, const std::type_info& rhs)
    { return lhs.before(rhs) != 0; }
};

}
}
```

Note

Most, but not all of the builtin concepts can be specialized. Constructors, destructors, and RTTI need special treatment from the library and cannot be specialized. Only primitive concepts can be specialized, so the iterator concepts are also out.

Associated Types

(For the source of the examples in this section see associated.cpp)

Associated types are defined using the `deduced` template. `deduced` is just like an ordinary placeholder, except that the type that it binds to is determined by calling a metafunction and does not need to be specified explicitly.

For example, we can define a concept for holding any iterator, raw pointer, or smart pointer as follows.

Note the extra trickery to make sure that it is safe to instantiate pointee with a placeholder, because argument dependant lookup can cause spurious instantiations.

```
template<class T>
struct pointee
{
    typedef typename mpl::eval_if<is_placeholder<T>,
        mpl::identity<void>,
        boost::pointee<T>
    >::type type;
};

template<class T = _self>
struct pointer :
    mpl::vector<
        copy_constructible<T>,
        dereferenceable<deduced<pointee<T> >&, T>
    >
{
    // provide a typedef for convenience
    typedef deduced<pointee<T> > element_type;
};
```

Now the Concept of `x` uses two placeholders, `_self` and `pointer<>::element_type`. When we construct `x`, with an `int*`, `pointer<>::element_type` is deduced as `pointee<int*>::type` which is `int`. Thus, dereferencing `x` returns an `any` that contains an `int`.

```
int i = 10;
any<
    mpl::vector<
        pointer<>,
        typeid_<pointer<>::element_type>
    >
> x(&i);
int j = any_cast<int>(*x); // j == i
```

Sometimes we want to require that the associated type be a specific type. This can be solved using the `same_type` concept. Here we create an any that can hold any pointer whose element type is `int`.

```
int i = 10;
any<
    mpl::vector<
        pointer<>,
        same_type<pointer<>::element_type, int>
    >
> x(&i);
std::cout << *x << std::endl; // prints 10
```

Using `same_type` like this effectively causes the library to replace all uses of `pointer<>::element_type` with `int` and validate that it is always bound to `int`. Thus, dereferencing `x` now returns an `int`.

`same_type` can also be used for two placeholders. This allows us to use a simple name instead of writing out an associated type over and over.

```
int i = 10;
any<
    mpl::vector<
        pointer<>,
        same_type<pointer<>::element_type, _a>,
        typeid_<_a>,
        copy_constructible<_a>,
        addable<_a>,
        ostreamable<std::ostream, _a>
    >
> x(&i);
std::cout << (*x + *x) << std::endl; // prints 20
```

Using Any

Construction

(For the source of the examples in this section see construction.cpp)

The library provides the `constructible` concept to allow an `any` to capture constructors. The single template argument should be a function signature. The return type must be a placeholder specifying the type to be constructed. The arguments are the arguments of the constructor.

```
typedef mpl::vector<
    copy_constructible<_a>,
    copy_constructible<_b>,
    copy_constructible<_c>,
    constructible<_a(const _b&, const _c&)>
> construct;

typedef mpl::map<
    mpl::pair<_a, std::vector<double> >,
    mpl::pair<_b, std::size_t>,
    mpl::pair<_c, double>
> types;

any<construct, _b> size(std::size_t(10), make_binding<types>());
any<construct, _c> val(2.5, make_binding<types>());
any<construct, _a> v(size, val);
// v holds std::vector<double>(10, 2.5);
```

Now, suppose that we want a default constructor? We can't have the default constructor of `any` call the default constructor of the contained type, because it would have no way of knowing what the contained type is. So, we'll need to pass the placeholder binding information explicitly.

```
typedef mpl::vector<
    copy_constructible<>,
    constructible<_self()>
> construct;

any<construct> x(std::string("Test"));
any<construct> y(binding_of(x)); // y == ""
```

This method is not restricted to the default constructor. If the constructor takes arguments, they can be passed after the bindings.

```
typedef mpl::vector<
    copy_constructible<>,
    constructible<_self(std::size_t, char)>
> construct;

any<construct> x(std::string("Test"));
any<construct> y(binding_of(x), 5, 'A');
```

Conversions

(For the source of the examples in this section see convert.cpp)

An `any` can be converted to another `any` as long as the conversion is an "upcast."

```
typedef any<
    mpl::vector<
        copy_constructible<>,
        typeid_<>,
        ostreamable<>
    >
> any_printable;
typedef any<
    mpl::vector<
        copy_constructible<>,
        typeid_<>
    >
> common_any;
any_printable x(10);
common_any y(x);
```

This conversion is okay because the requirements of `common_any` are a subset of the requirements of `any_printable`. Conversion in the other direction is illegal.

```
common_any x(10);
any_printable y(x); // error
```

References

(For the source of the examples in this section see references.cpp)

To capture by reference, we simply add a reference to the `placeholder`.

```
int i;
any<typeid_<>, _self&> x(i);
any_cast<int&>(x) = 5; // now i is 5
```

Note

`_self` is the default `placeholder`, so it is easiest to use `_self&`. We could use another `placeholder` instead. `any<typeid_<_a>, _a&>` has exactly the same behavior.

References cannot be rebound. Just like a built-in C++ reference, once you've initialized it you can't change it to point to something else.

```
int i, j;
any<typeid_<>, _self&> x(i), y(j);
x = y; // error
```

Note

As with any other operation, `x = y` for references acts on `i` and `j`. Assignment like this is legal if `assignable<>` is in the Concept, but `x` would still hold a reference to `i`.

A reference can be bound to another `any`.

```
typedef mpl::vector<
    copy_constructible<>,
    incrementable<>
> requirements;

any<requirements> x(10);
any<requirements, _self&> y(x);
++y; // x is now 11
```

If a reference is used after the underlying object goes out of scope or is reset, the behavior is undefined.

```
typedef mpl::vector<
    copy_constructible<>,
    incrementable<>,
    relaxed
> requirements;
any<requirements> x(10);
any<requirements, _self&> y(x);
x = 1.0;
++y; // undefined behavior.
```

This only applies when a reference is constructed from a value. If a reference is constructed from another reference, the new reference does not depend on the old one.

```
any<requirements> x(10);
boost::shared_ptr<any<requirements, _self&> > p(
    new any<requirements, _self&>(x));
any<requirements, _self&> y(*p); // equivalent to y(x);
p.reset();
++y; // okay
```

Both const and non-const references are supported.

```
int i = 0;
any<incrementable<>, _self&> x(i);
any<incrementable<>, const _self&> y(x);
```

A reference to non-const can be converted to a reference to const, but not the other way around. Naturally, we can't apply mutating operations to a const reference.

```
any<incrementable<>, _self&> z(y); // error
++y; // error
```

Syntax Limitations

In most cases using an any has the same syntax as using the underlying object. However, there are a few cases where this is not possible to implement. An `any` reference is proxy and cannot be used in contexts where a real reference is required. In particular, `forward_iterator` does not create a conforming ForwardIterator (unless the value_type is fixed.) Another difference is that all operations which do not take at least one `any` argument have to be passed the type information explicitly. Static member functions and constructors can fall in this category. All this means that generic algorithms might not work when applied to `any` arguments.

Examples

A polymorphic range formatter

(For the source of this example see print_sequence.cpp)

This example defines a class hierarchy that allows a sequence to be formatted in several different ways. We'd like to be able to handle any sequence and any stream type, since the range formatting is independent of the formatting of individual elements. Thus, our interface needs to look something like this:

```
class abstract_printer {
public:
    template<class CharT, class Traits, class Range>
    virtual void print(std::basic_ostream<CharT, Traits>& os, const Range& r) const = 0;
};
```

Unfortunately, this is illegal because a virtual function cannot be a template. However, we can define a class with much the same behavior using Boost.TypeErasure.

```
#include <boost/type_erasure/any.hpp>
#include <boost/type_erasure/iterator.hpp>
#include <boost/type_erasure/operators.hpp>
#include <boost/type_erasure/tuple.hpp>
#include <boost/type_erasure/same_type.hpp>
#include <boost/range/begin.hpp>
#include <boost/range/end.hpp>
#include <boost/range/iterator.hpp>
#include <iostream>
#include <iomanip>
#include <vector>

using namespace boost::type_erasure;

struct _t : placeholder {};
struct _iter : placeholder {};
struct _os : placeholder {};

template<class T, class U = _self>
struct base_and_derived
{
    static T& apply(U& arg) { return arg; }
};

namespace boost {
namespace type_erasure {

template<class T, class U, class Base>
struct concept_interface<base_and_derived<T, U>, Base, U> : Base
{
    operator typename rebind_any<Base, const T&>::type() const
    {
        return call(base_and_derived<T, U>(), const_cast<concept_interface&>(*this));
    }
    operator typename rebind_any<Base, T&>::type()
    {
        return call(base_and_derived<T, U>(), *this);
    }
};
```

```
}
}

// abstract_printer - An abstract base class for formatting sequences.
class abstract_printer {
public:
    // print - write a sequence to a std::ostream in a manner
    //   specific to the derived class.
    //
    // Requires: Range must be a Forward Range whose elements can be
    //   printed to os.
    template<class CharT, class Traits, class Range>
    void print(std::basic_ostream<CharT, Traits>& os, const Range& r) const {
        // Capture the arguments
        typename boost::range_iterator<const Range>::type
            first(boost::begin(r)),
            last(boost::end(r));
        tuple<requirements, _os&, _iter, _iter> args(os, first, last);
        // and forward to the real implementation
        do_print(get<0>(args), get<1>(args), get<2>(args));
    }
    virtual ~abstract_printer() {}
protected:
    // define the concept requirements of the arguments of
    // print and typedef the any types.
    typedef boost::mpl::vector<
        base_and_derived<std::ios_base, _os>,
        ostreamable<_os, _t>,
        ostreamable<_os, const char*>,
        forward_iterator<_iter, const _t&>,
        same_type<_t, forward_iterator<_iter, const _t&>::value_type>
    > requirements;
    typedef boost::type_erasure::any<requirements, _os&> ostream_type;
    typedef boost::type_erasure::any<requirements, _iter> iterator_type;
    // do_print - This method must be implemented by derived classes
    virtual void do_print(
        ostream_type os, iterator_type first, iterator_type last) const = 0;
};

// separator_printer - writes the elements of a sequence
//   separated by a fixed string.  For example, if
//   the separator is ", " separator_printer produces
//   a comma separated list.
class separator_printer : public abstract_printer {
public:
    explicit separator_printer(const std::string& sep) : separator(sep) {}
protected:
    virtual void do_print(
        ostream_type os, iterator_type first, iterator_type last) const {
        if(first != last) {
            os << *first;
            ++first;
            for(; first != last; ++first) {
                os << separator.c_str() << *first;
            }
        }
    }
private:
    std::string separator;
};

// column_separator_printer - like separator_printer, but
//   also inserts a line break after every n elements.
```

```
class column_separator_printer : public abstract_printer {
public:
    column_separator_printer(const std::string& sep, std::size_t num_columns)
      : separator(sep),
        cols(num_columns)
    {}
protected:
    virtual void do_print(
        ostream_type os, iterator_type first, iterator_type last) const {
        std::size_t count = 0;
        for(; first != last; ++first) {
            os << *first;
            boost::type_erasure::any<requirements, _iter> temp = first;
            ++temp;
            if(temp != last) {
                os << separator.c_str();
            }
            if(++count % cols == 0) {
                os << "\n";
            }
        }
    }
private:
    std::string separator;
    std::size_t cols;
};

// aligned_column_printer - formats a sequence in columns
//   reading down.  For example, given the sequence
//   { 1, 2, 3, 4, 5 }, aligned_column_printer might print
//   1   4
//   2   5
//   3
class aligned_column_printer : public abstract_printer {
public:
    aligned_column_printer(std::size_t column_width, std::size_t num_columns)
      : width(column_width),
        cols(num_columns)
    {}
protected:
    virtual void do_print(
        ostream_type os, iterator_type first, iterator_type last) const
    {
        if(first == last) return;
        std::vector<iterator_type> column_iterators;

        // find the tops of the columns
        std::size_t count = 0;
        for(iterator_type iter = first; iter != last; ++iter) {
            ++count;
        }
        std::size_t rows = (count + cols - 1) / cols;
        count = 0;
        for(iterator_type iter = first; iter != last; ++iter) {
            if(count % rows == 0) {
                column_iterators.push_back(iter);
            }
            ++count;
        }

        iterator_type last_col = column_iterators.back();

        // print the full rows
```

```
        while(column_iterators.back() != last) {
            for(std::vector<iterator_type>::iterator
                iter = column_iterators.begin(),
                end = column_iterators.end(); iter != end; ++iter)
            {
                static_cast<std::ios_base&>(os).width(width);
                os << **iter;
                ++*iter;
            }
            os << "\n";
        }

        // print the rows that are missing the last column
        column_iterators.pop_back();
        if(!column_iterators.empty()) {
            while(column_iterators.back() != last_col) {
                for(std::vector<iterator_type>::iterator
                    iter = column_iterators.begin(),
                    end = column_iterators.end(); iter != end; ++iter)
                {
                    static_cast<std::ios_base&>(os).width(width);
                    os << **iter;
                    ++*iter;
                }
                os << "\n";
            }
        }
    }
private:
    std::size_t width;
    std::size_t cols;
};

int main() {
    int test[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    separator_printer p1(",");
    p1.print(std::cout, test);
    std::cout << std::endl;
    column_separator_printer p2(",", 4);
    p2.print(std::cout, test);
    std::cout << std::endl;
    aligned_column_printer p3(16, 4);
    p3.print(std::cout, test);
}
```

A type-safe printf

(For the source of this example see printf.cpp)

This example uses the library to implement a type safe printf.

Note

This example uses C++11 features. You'll need a recent compiler for it to work.

```
#include <boost/type_erasure/builtin.hpp>
#include <boost/type_erasure/operators.hpp>
#include <boost/type_erasure/any_cast.hpp>
#include <boost/type_erasure/any.hpp>
#include <boost/mpl/vector.hpp>
#include <boost/io/ios_state.hpp>
#include <iostream>
#include <sstream>
#include <iomanip>
#include <vector>
#include <string>

namespace mpl = boost::mpl;
using namespace boost::type_erasure;
using namespace boost::io;

// We capture the arguments by reference and require nothing
// except that each one must provide a stream insertion operator.
typedef any<
    mpl::vector<
        typeid_<>,
        ostreamable<>
    >,
    const _self&
> any_printable;
typedef std::vector<any_printable> print_storage;

// Forward declaration of the implementation function
void print_impl(std::ostream& os, const char * format, const print_storage& args);

// print
//
// Writes values to a stream like the classic C printf function.  The
// arguments are formatted based on specifiers in the format string,
// which match the pattern:
//
// '%' [ argument-number '$' ] flags * [ width ] [ '.' precision ] [ type-code ] format-specifier
//
// Other characters in the format string are written to the stream unchanged.
// In addition the sequence, "%%" can be used to print a literal '%' character.
// Each component is explained in detail below
//
// argument-number:
//   The value must be between 1 and sizeof... T.  It indicates the
//   index of the argument to be formatted.  If no index is specified
//   the arguments will be processed sequentially.  If an index is
//   specified for one argument, then it must be specified for every argument.
//
// flags:
//   Consists of zero or more of the following:
//   '-': Left justify the argument
//   '+': Print a plus sign for positive integers
//   '0': Use leading 0's to pad instead of filling with spaces.
//   ' ': If the value doesn't begin with a sign, prepend a space
//   '#': Print 0x or 0 for hexadecimal and octal numbers.
//
// width:
//   Indicates the minimum width to print.  This can be either
//   an integer or a '*'.  an asterisk means to read the next
//   argument (which must have type int) as the width.
//
// precision:
//   For numeric arguments, indicates the number of digits to print.  For
```

```
//   strings (%s) the precision indicates the maximum number of characters
//   to print.  Longer strings will be truncated.  As with width
//   this can be either an integer or a '*'.  an asterisk means
//   to read the next argument (which must have type int) as
//   the width.  If both the width and the precision are specified
//   as '*', the width is read first.
//
// type-code:
//   This is ignored, but provided for compatibility with C printf.
//
// format-specifier:
//   Must be one of the following characters:
//   d, i, u: The argument is formatted as a decimal integer
//   o:       The argument is formatted as an octal integer
//   x, X:    The argument is formatted as a hexadecimal integer
//   p:       The argument is formatted as a pointer
//   f:       The argument is formatted as a fixed point decimal
//   e, E:    The argument is formatted in exponential notation
//   g, G:    The argument is formatted as either fixed point or using
//            scientific notation depending on its magnitude
//   c:       The argument is formatted as a character
//   s:       The argument is formatted as a string
//
template<class... T>
void print(std::ostream& os, const char * format, const T&... t)
{
    // capture the arguments
    print_storage args = { any_printable(t)... };
    // and forward to the real implementation
    print_impl(os, format, args);
}

// This overload of print with no explicit stream writes to std::cout.
template<class... T>
void print(const char * format, const T&... t)
{
    print(std::cout, format, t...);
}

// The implementation from here on can be separately compiled.

// utility function to parse an integer
int parse_int(const char *& format) {
    int result = 0;
    while(char ch = *format) {
        switch(ch) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            result = result * 10 + (ch - '0');
            break;
        default: return result;
        }
        ++format;
    }
    return result;
}

// printf implementation
void print_impl(std::ostream& os, const char * format, const print_storage& args) {
    int idx = 0;
    ios_flags_saver savef_outer(os, std::ios_base::dec);
    bool has_positional = false;
    bool has_indexed = false;
```

```
    while(char ch = *format++) {
        if (ch == '%') {
            if (*format == '%') { os << '%'; continue; }

            ios_flags_saver savef(os);
            ios_precision_saver savep(os);
            ios_fill_saver savefill(os);

            int precision = 0;
            bool pad_space = false;
            bool pad_zero = false;

            // parse argument index
            if (*format != '0') {
                int i = parse_int(format);
                if (i != 0) {
                    if(*format == '$') {
                        idx = i - 1;
                        has_indexed = true;
                        ++format;
                    } else {
                        os << std::setw(i);
                        has_positional = true;
                        goto parse_precision;
                    }
                } else {
                    has_positional = true;
                }
            } else {
                has_positional = true;
            }

            // Parse format modifiers
            while((ch = *format)) {
                switch(ch) {
                case '-': os << std::left; break;
                case '+': os << std::showpos; break;
                case '0': pad_zero = true; break;
                case ' ': pad_space = true; break;
                case '#': os << std::showpoint << std::showbase; break;
                default: goto parse_width;
                }
                ++format;
            }

        parse_width:
            int width;
            if (*format == '*') {
                ++format;
                width = any_cast<int>(args.at(idx++));
            } else {
                width = parse_int(format);
            }
            os << std::setw(width);

        parse_precision:
            if (*format == '.') {
                ++format;
                if (*format == '*') {
                    ++format;
                    precision = any_cast<int>(args.at(idx++));
                } else {
                    precision = parse_int(format);
```

```
        }
        os << std::setprecision(precision);
    }

    // parse (and ignore) the type modifier
    switch(*format) {
    case 'h': ++format; if(*format == 'h') ++format; break;
    case 'l': ++format; if(*format == 'l') ++format; break;
    case 'j':
    case 'L':
    case 'q':
    case 't':
    case 'z':
        ++format; break;
    }

    std::size_t truncate = 0;

    // parse the format code
    switch(*format++) {
    case 'd': case 'i': case 'u': os << std::dec; break;
    case 'o': os << std::oct; break;
    case 'p': case 'x': os << std::hex; break;
    case 'X': os << std::uppercase << std::hex; break;
    case 'f': os << std::fixed; break;
    case 'e': os << std::scientific; break;
    case 'E': os << std::uppercase << std::scientific; break;
    case 'g': break;
    case 'G': os << std::uppercase; break;
    case 'c': case 'C': break;
    case 's': case 'S': truncate = precision; os << std::setprecision(6); break;
    default: assert(!"Bad format string");
    }

    if (pad_zero && !(os.flags() & std::ios_base::left)) {
        os << std::setfill('0') << std::internal;
        pad_space = false;
    }

    if (truncate != 0 || pad_space) {
        // These can't be handled by std::setw.  Write to a stringstream and
        // pad/truncate manually.
        std::ostringstream oss;
        oss.copyfmt(os);
        oss << args.at(idx++);
        std::string data = oss.str();

        if (pad_space) {
            if (data.empty() || (data[0] != '+' && data[0] != '-' && data[0] != ' ')) {
                os << ' ';
            }
        }
        if (truncate != 0 && data.size() > truncate) {
            data.resize(truncate);
        }
        os << data;
    } else {
        os << args.at(idx++);
    }

    // we can't have both positional and indexed arguments in
    // the format string.
    assert(has_positional ^ has_indexed);
```

```
        } else {
            std::cout << ch;
        }
    }
}

int main() {
    print("int: %d\n", 10);
    print("int: %0#8X\n", 0xA56E);
    print("double: %g\n", 3.14159265358979323846);
    print("double: %f\n", 3.14159265358979323846);
    print("double: %+20.9e\n", 3.14159265358979323846);
    print("double: %0+20.9g\n", 3.14159265358979323846);
    print("double: %*.*g\n", 20, 5, 3.14159265358979323846);
    print("string: %.10s\n", "Hello World!");
    print("double: %2$*.*g int: %1$d\n", 10, 20, 5, 3.14159265358979323846);
}
```

Boost.Function with multiple signatures

(For the source of this example see multifunction.cpp)

This example implements an extension of Boost.Function that supports multiple signatures.

Note

This example uses C++11 features. You'll need a recent compiler for it to work.

```
#include <boost/type_erasure/any.hpp>
#include <boost/type_erasure/builtin.hpp>
#include <boost/type_erasure/callable.hpp>
#include <boost/mpl/vector.hpp>
#include <boost/variant.hpp>
#include <boost/phoenix/core.hpp>
#include <boost/phoenix/operator.hpp>
#include <boost/range/algorithm.hpp>
#include <algorithm>
#include <vector>
#include <string>
#include <iostream>

namespace mpl = boost::mpl;
using namespace boost::type_erasure;
namespace phoenix = boost::phoenix;

// First of all we'll declare the multifunction template.
// multifunction is like Boost.Function but instead of
// taking one signature, it takes any number of them.
template<class... Sig>
using multifunction =
    any<
        mpl::vector<
            copy_constructible<>,
            typeid_<>,
            relaxed,
            callable<Sig>...
        >
    >;

// Let's use multifunction to process a variant.  We'll start
// by defining a simple recursive variant to use.
typedef boost::make_recursive_variant<
    int,
    double,
    std::string,
    std::vector<boost::recursive_variant_> >::type variant_type;
typedef std::vector<variant_type> vector_type;

// Now we'll define a multifunction that can operate
// on the leaf nodes of the variant.
typedef multifunction<void(int), void(double), void(std::string)> function_type;

class variant_handler
{
public:
    void handle(const variant_type& arg)
    {
        boost::apply_visitor(impl, arg);
    }
    void set_handler(function_type f)
    {
        impl.f = f;
    }
private:
    // A class that works with boost::apply_visitor
    struct dispatcher : boost::static_visitor<void>
    {
        // used for the leaves
        template<class T>
        void operator()(const T& t) { f(t); }
        // For a vector, we recursively operate on the elements
```

```
        void operator()(const vector_type& v)
        {
            boost::for_each(v, boost::apply_visitor(*this));
        }
        function_type f;
    };
    dispatcher impl;
};

int main() {
    variant_handler x;
    x.set_handler(std::cout << phoenix::val("Value: ") << phoenix::placeholders::_1 << std::endl);

    x.handle(1);
    x.handle(2.718);
    x.handle("The quick brown fox jumps over the lazy dog.");
    x.handle(vector_type{ 1.618, "Gallia est omnis divisa in partes tres", 42 });
}
```

Concept Definitions

A Concept defines a set of constraints on the types that are stored in an `any`.

There are three kinds of concepts.

1. The library defines a number of predefined concepts. Most of these are equivalent to user-defined concepts, but a few require special handling.

2. Users can define their own primitive concepts as described below. The macros `BOOST_TYPE_ERASURE_MEMBER` and `BOOST_TYPE_ERASURE_FREE` define concepts of this form.

3. Any MPL Forward Sequence whose elements are concepts is also a concept. This allows concepts to be composed easily.

Each primitive concept defines a single function. A primitive concept must be a specialization of a class template, with a static member function called `apply`, which will be executed when the function is dispatched by `call`. The template can only take template type parameters. non-type template parameters and template template parameters are not allowed.

The template parameters of the concept may involve placeholders. The following are considered.

- Each template argument may be a cv and/or reference qualified placeholder type.

- If a template argument is a function type, its arguments and return type may be cv/reference qualified placeholders.

Any other placeholders are ignored.

A concept is instantiated by constructing an `any` from a raw value or by constructing a `binding`. When a concept is instantiated with a specific set of type bindings, each placeholder is bound to a cv-unqualified non-reference type. After replacing each placeholder in the template argument list with the type that it binds to, the following must hold.

- The number of arguments of apply in the bound concept must be the same as the number of arguments in the unbound concept.

- The arguments and return type of apply in the bound concept can be derived from the corresponding arguments and the return type in the unbound concept as follows: If the argument in the unbound concept is a placeholder with optional cv and reference qualifiers, then the argument in the bound concept can be found by replacing the placeholder. Otherwise, the argument in the unbound concept must be the same as the argument in the bound concept.

```
// Correct.
template<class T = _self>
struct foo1 {
  static void apply(const T& t) { t.foo(); }
};

// Wrong.  The signature of apply is different from the
// primary template
template<>
struct foo1<int> {
  static void apply(int i);
};

// Wrong.  A concept must be a template
struct foo2 {
  static void apply(const _self&);
};

// Wrong. apply must be static
template<class T = _self>
struct foo3 {
  void apply(const T&);
};

// Wrong.  apply cannot be overloaded
template<class T = _self>
struct foo3 {
  static void apply(T&);
  static void apply(const T&);
};

// Wrong.  Only top level placeholders are detected
template<class T>
struct foo4;
template<class T>
struct foo4<boost::mpl::vector<T> > {
  static void apply(const T&);
};

// Wrong.  Template template parameters are not allowed.
template<template<class> class T>
struct foo5
{
    static void apply(T<int>&);
};
```

Predefined Concepts

In the following tables, `T` and `U` are the types that the operation applies to, `R` is the result type. `T` always defaults to `_self` to match the default behavior of any. These concepts assume normal semantics. Thus, comparison operators always return bool, and references will be added to the arguments and results as appropriate.

Except as otherwise noted, primitive concepts defined by the library can be specialized to provide concept maps. `copy_constructible`, and the iterator concepts cannot be specialized because they are composites. `constructible`, `destructible`, `typeid_`, and `same_type` cannot be specialized because they require special handling in the library.

Table 1. Special Members

concept	notes
`constructible<Sig>`	-
`copy_constructible<T>`	-
`destructible<T>`	-
`assignable<T, U = T>`	-
`typeid_<T>`	-

Table 2. Unary Operators

operator	concept	notes
`operator++`	`incrementable<T>`	There is no separate post-increment
`operator--`	`decrementable<T>`	There is no separate post-decrement
`operator*`	`dereferenceable<R, T>`	`R` should usually be a reference
`operator~`	`complementable<T, R = T>`	-
`operator-`	`negatable<T, R = T>`	-

Table 3. Binary Operators

operator	concept	notes
operator+	addable<T, U = T, R = T>	-
operator-	subtractable<T, U = T, R = T>	-
operator*	multipliable<T, U = T, R = T>	-
operator/	dividable<T, U = T, R = T>	-
operator%	modable<T, U = T, R = T>	-
operator&	bitandable<T, U = T, R = T>	-
operator|	bitorable<T, U = T, R = T>	-
operator^	bitxorable<T, U = T, R = T>	-
operator<<	left_shiftable<T, U = T, R = T>	-
operator>>	right_shiftable<T, U = T, R = T>	-
operator== and !=	equality_comparable<T, U = T>	!= is implemented in terms of ==
operator<, >, <=, and >=	less_than_comparable<T, U = T>	All are implemented in terms of <
operator+=	add_assignable<T, U = T>	-
operator-=	subtract_assignable<T, U = T>	-
operator*=	multiply_assignable<T, U = T>	-
operator/=	divide_assignable<T, U = T>	-
operator%=	mod_assignable<T, U = T>	-
operator&=	bitand_assignable<T, U = T>	-
operator|=	bitor_assignable<T, U = T>	-
operator^=	bitxor_assignable<T, U = T>	-
operator<<=	left_shift_assignable<T, U = T>	-
operator>>=	right_shift_assignable<T, U = T>	-
operator<<	ostreamable<Os = std::ostream, T = _self>	-
operator>>	istreamable<Is = std::istream, T = _self>	-

Table 4. Miscellaneous Operators

operator	concept	notes
`operator()`	`callable<Sig, T>`	`Sig` should be a function type. `T` may be const qualified.
`operator[]`	`subscriptable<R, T, N = std::ptrdiff_t>`	`R` should usually be a reference. `T` can be optionally const qualified.

Table 5. Iterator Concepts

concept	notes
`iterator<Traversal, T, Reference, Difference>`	Use `same_type` to control the iterator's value type.
`forward_iterator<T, Reference, Difference>`	-
`bidirectional_iterator<T, Reference, Difference>`	-
`random_access_iterator<T, Reference, Difference>`	-

Table 6. Special Concepts

concept	notes
`same_type<T>`	Indicates that two types are the same.

Reference

Header <boost/type_erasure/any.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Sig> struct constructible;
    template<typename T> struct destructible;
    template<typename T, typename U> struct assignable;

    template<typename Concept, typename T = _self> class any;

    template<typename Concept, typename T> class any<Concept, T &>;
    template<typename Concept, typename T> class any<Concept, const T &>;
    template<typename Concept, typename T> class any<Concept, T &&>;
  }
}
```

Struct template constructible

boost::type_erasure::constructible

Synopsis

```
// In header: <boost/type_erasure/any.hpp>

template<typename Sig>
struct constructible {
};
```

Description

The constructible concept enables calling the constructor of a type contained by an any. `Sig` should be a function signature. The return type is the placeholder specifying the type to be constructed. The arguments are the argument types of the constructor. The arguments of `Sig` may be placeholders.

Note

constructible may not be specialized and may not be passed to call as it depends on the implementation details of any.

Struct template destructible

boost::type_erasure::destructible

Synopsis

```
// In header: <boost/type_erasure/any.hpp>

template<typename T>
struct destructible {
};
```

Description

The destructible concept enables forwarding to the destructor of the contained type. This is required whenever an any is created by value.

Note

The destructible concept rarely needs to be specified explicitly, because it is included in the copy_constructible concept.

destructible may not be specialized and may not be passed to call as it depends on the implementation details of any.

Struct template assignable

boost::type_erasure::assignable

Synopsis

```
// In header: <boost/type_erasure/any.hpp>

template<typename T, typename U>
struct assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

Enables assignment of any types.

`assignable` public static functions

1. ```
   static void apply(T & dst, const U & src);
   ```

Class template any

boost::type_erasure::any
   ```

Synopsis

```
// In header: <boost/type_erasure/any.hpp>

template<typename Concept, typename T = _self>
class any {
public:
  // construct/copy/destruct
  any();
  template<typename U> any(U &&);
  template<typename U, typename Map> any(U &&, const static_binding< Map > &);
  any(const any &);
  template<typename Concept2, typename Tag2>
    any(const any< Concept2, Tag2 > &);
  template<typename Concept2, typename Tag2, typename Map>
    any(const any< Concept2, Tag2 > &, const static_binding< Map > &);
  template<typename Concept2, typename Tag2>
    any(const any< Concept2, Tag2 > &, const binding< Concept > &);
  template<class... U> explicit any(U &&...);
  template<class... U> explicit any(const binding< Concept > &, U &&...);
  any& operator=(const any &);
  template<typename U> any& operator=(const U &);
  ~any();
};
```

Description

The class template any can store any object that models a specific Concept. It dispatches all the functions defined by the Concept to the contained type at runtime.

See Also:

concept_of, placeholder_of, any_cast, is_empty, binding_of, typeid_of

Template Parameters

1. `typename Concept`

   The Concept that the stored type should model.

2. `typename T = _self`

   A `placeholder` specifying which type this is.

any public construct/copy/destruct

1. `any();`

   Constructs an empty `any`.

   Except as otherwise noted, all operations on an empty `any` result in a `bad_function_call` exception. The copy-constructor of an empty `any` creates another null `any`. The destructor of an empty `any` is a no-op. Comparison operators treat all empty `anys` as equal. `typeid_of` applied to an empty `any` returns `typeid(void)`.

   An `any` which does not include `relaxed` in its Concept can never be null.

   See Also:

`is_empty`
Requires: relaxed must be in `Concept`.
Throws: Nothing.

2.
```
template<typename U> any(U && data);
```

Constructs an `any` to hold a copy of `data`. The `Concept` will be instantiated with the placeholder `T` bound to U.

Note

This constructor never matches if the argument is an `any`, `binding`, or `static_binding`.

Parameters: `data` The object to store in the `any`.
Requires: `U` is a model of `Concept`.

`U` must be CopyConstructible.

`Concept` must not refer to any non-deduced placeholder besides `T`.
Throws: std::bad_alloc or whatever that the copy constructor of `U` throws.

3.
```
template<typename U, typename Map>
  any(U && data, const static_binding< Map > & binding);
```

Constructs an `any` to hold a copy of `data` with explicitly specified placeholder bindings.

Note

This constructor never matches if the argument is an `any`.

Parameters: `binding` Specifies the types that all the placeholders should bind to.
`data` The object to store in the `any`.
Requires: `U` is a model of `Concept`.

`U` must be CopyConstructible.

`Map` is an MPL map with an entry for every non-deduced placeholder referred to by `Concept`.

`T` must map to `U` in `Map`.
Throws: std::bad_alloc or whatever that the copy constructor of `U` throws.

4.
```
any(const any & other);
```

Copies an `any`.

Parameters: `other` The object to make a copy of.
Requires: `Concept` must contain constructible<T(const T&)>. (This is included in copy_constructible<T>)
Throws: std::bad_alloc or whatever that the copy constructor of the contained type throws.

5.
```
template<typename Concept2, typename Tag2>
  any(const any< Concept2, Tag2 > & other);
```

Upcasts from an `any` with stricter requirements to an `any` with weaker requirements.

Parameters: `other` The object to make a copy of.

Requires: `Concept` must contain constructible<T(const T&)>.

`Concept` must not refer to any non-deduced placeholder besides `T`.

After substituting `T` for `Tag2`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.

Throws: std::bad_alloc or whatever that the copy constructor of the contained type throws.

6.
```
template<typename Concept2, typename Tag2, typename Map>
  any(const any< Concept2, Tag2 > & other,
      const static_binding< Map > & binding);
```

Constructs an any from another any.

Parameters: `binding` Specifies the mapping between the placeholders used by the two concepts.
`other` The object to make a copy of.

Requires: `Concept` must contain constructible<T(const T&)>.

`Map` must be an MPL map with keys for all the non-deduced placeholders used by `Concept` and values for the corresponding placeholders in `Concept2`.

After substituting placeholders according to `Map`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.

Throws: std::bad_alloc or whatever that the copy constructor of the contained type throws.

7.
```
template<typename Concept2, typename Tag2>
  any(const any< Concept2, Tag2 > & other, const binding< Concept > & binding);
```

Constructs an any from another any.

Warning

This constructor is potentially dangerous, as it cannot check at compile time whether the arguments match.

Parameters: `binding` Specifies the bindings of placeholders to actual types.
`other` The object to make a copy of.

Requires: `Concept` must contain constructible<T(const T&)>.

The type stored in `other` must match the type expected by `binding`.

Postconditions: binding_of(*this) == `binding`

Throws: std::bad_alloc or whatever that the copy constructor of the contained type throws.

8.
```
template<class... U> explicit any(U &&... arg);
```

Calls a constructor of the contained type. The bindings will be deduced from the arguments.

Note

This constructor is never chosen if any other constructor can be called instead.

Parameters: `arg` The arguments to be passed to the underlying constructor.

Requires: `Concept` must contain an instance of constructible which can be called with these arguments.

At least one of the arguments must by an any with the same `Concept` as this.

The bindings of all the arguments that are any's, must be the same.

Throws: std::bad_alloc or whatever that the constructor of the contained type throws.

9.
```
template<class... U>
  explicit any(const binding< Concept > & binding, U &&... arg);
```

Calls a constructor of the contained type.

Parameters: `arg` The arguments to be passed to the underlying constructor.
`binding` Specifies the bindings of placeholders to actual types.
Requires: `Concept` must contain a matching instance of constructible.

The contained type of every argument that is an any, must be the same as that specified by `binding`.
Postconditions: binding_of(*this) == `binding`
Throws: std::bad_alloc or whatever that the constructor of the contained type throws.

10.
```
any& operator=(const any & other);
```

Assigns to an `any`.

If an appropriate overload of `assignable` is not available and `relaxed` is in `Concept`, falls back on constructing from `other`.

Throws: Whatever the assignment operator of the contained type throws. When falling back on construction, throws `std::bad_alloc` or whatever the copy constructor of the contained type throws. In this case assignment provides the strong exception guarantee. When calling the assignment operator of the contained type, the exception guarantee is whatever the contained type provides.

11.
```
template<typename U> any& operator=(const U & other);
```

Assigns to an `any`.

If an appropriate overload of `assignable` is not available and `relaxed` is in `Concept`, falls back on constructing from `other`.

Throws: Whatever the assignment operator of the contained type throws. When falling back on construction, throws `std::bad_alloc` or whatever the copy constructor of the contained type throws. In this case assignment provides the strong exception guarantee. When calling an assignment operator of the contained type, the exception guarantee is whatever the contained type provides.

12.
```
~any();
```

Requires: `Concept` includes destructible<T>.

Specializations

- Class template any<Concept, T &>
- Class template any<Concept, const T &>
- Class template any<Concept, T &&>

Class template any<Concept, T &>

boost::type_erasure::any<Concept, T &>

Synopsis

```
// In header: <boost/type_erasure/any.hpp>

template<typename Concept, typename T>
class any<Concept, T &> {
public:
  // construct/copy/destruct
  template<typename U> any(U &);
  template<typename U, typename Map> any(U &, const static_binding< Map > &);
  any(const any &);
  any(any< Concept, T > &);
  template<typename Concept2, typename Tag2>
    any(const any< Concept2, Tag2 & > &);
  template<typename Concept2, typename Tag2> any(any< Concept2, Tag2 > &);
  template<typename Concept2, typename Tag2, typename Map>
    any(const any< Concept2, Tag2 & > &, const static_binding< Map > &);
  template<typename Concept2, typename Tag2, typename Map>
    any(any< Concept2, Tag2 > &, const static_binding< Map > &);
  template<typename Concept2, typename Tag2>
    any(const any< Concept2, Tag2 & > &, const binding< Concept > &);
  template<typename Concept2, typename Tag2>
    any(any< Concept2, Tag2 > &, const binding< Concept > &);
  any& operator=(const any &);
  template<typename U> any& operator=(U &);
  template<typename U> any& operator=(const U &);
};
```

Description

any public construct/copy/destruct

1. ```
   template<typename U> any(U & arg);
   ```

   Constructs an any from a reference.

   Parameters: `arg` The object to bind the reference to.
   Requires: `U` is a model of `Concept`.

   `Concept` must not refer to any non-deduced placeholder besides `T`.
   Throws: Nothing.

2. ```
   template<typename U, typename Map>
     any(U & arg, const static_binding< Map > & binding);
   ```

   Constructs an any from a reference.

   Parameters: `arg` The object to bind the reference to.
   `binding` Specifies the actual types that all the placeholders should bind to.
   Requires: `U` is a model of `Concept`.

   `Map` is an MPL map with an entry for every non-deduced placeholder referred to by `Concept`.
   Throws: Nothing.

3. ```
   any(const any & other);
   ```

   Constructs an any from another reference.
   ```

Parameters: `other` The reference to copy.
Throws: Nothing.

4.
```
any(any< Concept, T > & other);
```

Constructs an `any` from another `any`.

Parameters: `other` The object to bind the reference to.
Throws: Nothing.

5.
```
template<typename Concept2, typename Tag2>
  any(const any< Concept2, Tag2 & > & other);
```

Constructs an `any` from another reference.

Parameters: `other` The reference to copy.
Requires: `Concept` must not refer to any non-deduced placeholder besides `T`.

After substituting `T` for `Tag2`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.
Throws: std::bad_alloc

6.
```
template<typename Concept2, typename Tag2> any(any< Concept2, Tag2 > & other);
```

Constructs an `any` from another `any`.

Parameters: `other` The object to bind the reference to.
Requires: `Concept` must not refer to any non-deduced placeholder besides `T`.

After substituting `T` for `Tag2`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.
Throws: std::bad_alloc

7.
```
template<typename Concept2, typename Tag2, typename Map>
  any(const any< Concept2, Tag2 & > & other,
      const static_binding< Map > & binding);
```

Constructs an `any` from another reference.

Parameters: `binding` Specifies the mapping between the two concepts.
`other` The reference to copy.
Requires: `Map` must be an MPL map with keys for all the non-deduced placeholders used by `Concept` and values for the corresponding placeholders in `Concept2`.

After substituting placeholders according to `Map`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.
Throws: std::bad_alloc

8.
```
template<typename Concept2, typename Tag2, typename Map>
  any(any< Concept2, Tag2 > & other, const static_binding< Map > & binding);
```

Constructs an `any` from another `any`.

Parameters: `binding` Specifies the mapping between the two concepts.
`other` The object to bind the reference to.
Requires: `Map` must be an MPL map with keys for all the non-deduced placeholders used by `Concept` and values for the corresponding placeholders in `Concept2`.

After substituting placeholders according to `Map`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.

Throws: std::bad_alloc

9.
```
template<typename Concept2, typename Tag2>
  any(const any< Concept2, Tag2 & > & other,
      const binding< Concept > & binding);
```

Constructs an `any` from another reference.

Parameters:	`binding`	Specifies the bindings of placeholders to actual types.
	`other`	The reference to copy.
Requires:	The type stored in `other` must match the type expected by `binding`.	
Postconditions:	binding_of(*this) == `binding`	
Throws:	Nothing.	

10.
```
template<typename Concept2, typename Tag2>
  any(any< Concept2, Tag2 > & other, const binding< Concept > & binding);
```

Constructs an `any` from another `any`.

Parameters:	`binding`	Specifies the bindings of placeholders to actual types.
	`other`	The object to bind the reference to.
Requires:	The type stored in `other` must match the type expected by `binding`.	
Postconditions:	binding_of(*this) == `binding`	
Throws:	Nothing.	

11.
```
any& operator=(const any & other);
```

Assigns to an `any`.

If an appropriate overload of `assignable` is not available and `relaxed` is in `Concept`, falls back on constructing from `other`.

Throws: Whatever the assignment operator of the contained type throws. When falling back on construction, throws `std::bad_alloc`. In this case assignment provides the strong exception guarantee. When calling the assignment operator of the contained type, the exception guarantee is whatever the contained type provides.

12.
```
template<typename U> any& operator=(U & other);
```

Assigns to an `any`.

If an appropriate overload of `assignable` is not available and `relaxed` is in `Concept`, falls back on constructing from `other`.

Throws: Whatever the assignment operator of the contained type throws. When falling back on construction, throws `std::bad_alloc`. In this case assignment provides the strong exception guarantee. When calling the assignment operator of the contained type, the exception guarantee is whatever the contained type provides.

13.
```
template<typename U> any& operator=(const U & other);
```

Assigns to an `any`.

If an appropriate overload of `assignable` is not available and `relaxed` is in `Concept`, falls back on constructing from `other`.

Throws: Whatever the assignment operator of the contained type throws. When falling back on construction, throws `std::bad_alloc`. In this case assignment provides the strong exception guarantee. When calling the assignment operator of the contained type, the exception guarantee is whatever the contained type provides.

Class template any<Concept, const T &>

boost::type_erasure::any<Concept, const T &>

Synopsis

```
// In header: <boost/type_erasure/any.hpp>

template<typename Concept, typename T>
class any<Concept, const T &> {
public:
  // construct/copy/destruct
  template<typename U> any(const U &);
  template<typename U, typename Map>
    any(const U &, const static_binding< Map > &);
  any(const any &);
  any(const any< Concept, T & > &);
  any(const any< Concept, T > &);
  any(const any< Concept, T && > &);
  template<typename Concept2, typename Tag2>
    any(const any< Concept2, Tag2 > &);
  template<typename Concept2, typename Tag2, typename Map>
    any(const any< Concept2, Tag2 > &, const static_binding< Map > &);
  template<typename Concept2, typename Tag2>
    any(const any< Concept2, Tag2 > &, const binding< Concept > &);
  any& operator=(const any &);
  template<typename U> any& operator=(const U &);
};
```

Description

any public construct/copy/destruct

1. ```
   template<typename U> any(const U & arg);
   ```

   Constructs an any from a reference.

   Parameters: `arg` The object to bind the reference to.
   Requires: `U` is a model of `Concept`.

   `Concept` must not refer to any non-deduced placeholder besides `T`.
   Throws: Nothing.

2. ```
   template<typename U, typename Map>
     any(const U & arg, const static_binding< Map > & binding);
   ```

   Constructs an any from a reference.

   Parameters: `arg` The object to bind the reference to.
   `binding` Specifies the actual types that all the placeholders should bind to.
   Requires: `U` is a model of `Concept`.

   `Map` is an MPL map with an entry for every non-deduced placeholder referred to by `Concept`.
   Throws: Nothing.

3. ```
   any(const any & other);
   ```
   ```

Constructs an any from another any.

Parameters: `other` The reference to copy.
Throws: Nothing.

4.
```
any(const any< Concept, T & > & other);
```

Constructs an any from another any.

Parameters: `other` The reference to copy.
Throws: Nothing.

5.
```
any(const any< Concept, T > & other);
```

Constructs an any from another any.

Parameters: `other` The object to bind the reference to.
Throws: Nothing.

6.
```
any(const any< Concept, T && > & other);
```

Constructs an any from another any.

Parameters: `other` The object to bind the reference to.
Throws: Nothing.

7.
```
template<typename Concept2, typename Tag2>
  any(const any< Concept2, Tag2 > & other);
```

Constructs an any from another any.

Parameters: `other` The object to bind the reference to.
Requires: `Concept` must not refer to any non-deduced placeholder besides `T`.

After substituting `T` for `Tag2`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.
Throws: std::bad_alloc

8.
```
template<typename Concept2, typename Tag2, typename Map>
  any(const any< Concept2, Tag2 > & other,
      const static_binding< Map > & binding);
```

Constructs an any from another any.

Parameters: `binding` Specifies the mapping between the two concepts.
`other` The object to bind the reference to.
Requires: `Map` must be an MPL map with keys for all the non-deduced placeholders used by `Concept` and values for the corresponding placeholders in `Concept2`.

After substituting placeholders according to `Map`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.
Throws: std::bad_alloc

9.
```
template<typename Concept2, typename Tag2>
  any(const any< Concept2, Tag2 > & other, const binding< Concept > & binding);
```

Constructs an any from another any.

Parameters:	`binding`	Specifies the bindings of placeholders to actual types.
	`other`	The object to bind the reference to.
Requires:	The type stored in `other` must match the type expected by `binding`.	
Postconditions:	binding_of(*this) == `binding`	
Throws:	Nothing.	

10.
```
any& operator=(const any & other);
```

Assigns to an any.

Requires:	relaxed is in `Concept`.
Throws:	Nothing.

11.
```
template<typename U> any& operator=(const U & other);
```

Assigns to an any.

Requires:	relaxed is in `Concept`.
Throws:	std::bad_alloc. Provides the strong exception guarantee.

Class template any<Concept, T &&>

boost::type_erasure::any<Concept, T &&>

Synopsis

```
// In header: <boost/type_erasure/any.hpp>

template<typename Concept, typename T>
class any<Concept, T &&> {
public:
  // construct/copy/destruct
  template<typename U> any(U &&);
  template<typename U, typename Map> any(U &&, const static_binding< Map > &);
  any(any< Concept, T > &&);
  template<typename Concept2, typename Tag2> any(any< Concept2, Tag2 && > &&);
  template<typename Concept2, typename Tag2> any(any< Concept2, Tag2 > &&);
  template<typename Concept2, typename Tag2, typename Map>
    any(const any< Concept2, Tag2 && > &, const static_binding< Map > &);
  template<typename Concept2, typename Tag2, typename Map>
    any(any< Concept2, Tag2 > &&, const static_binding< Map > &);
  template<typename Concept2, typename Tag2>
    any(const any< Concept2, Tag2 && > &, const binding< Concept > &);
  template<typename Concept2, typename Tag2>
    any(any< Concept2, Tag2 > &&, const binding< Concept > &);
  any& operator=(const any &);
  template<typename U> any& operator=(U &);
  template<typename U> any& operator=(const U &);
};
```

Description

any public construct/copy/destruct

1.
```
template<typename U> any(U && arg);
```

Constructs an any from a reference.

Parameters: `arg` The object to bind the reference to.
Requires: `U` is a model of `Concept`.

`Concept` must not refer to any non-deduced placeholder besides `T`.
Throws: Nothing.

2.
```
template<typename U, typename Map>
  any(U && arg, const static_binding< Map > & binding);
```

Constructs an `any` from a reference.

Parameters: `arg` The object to bind the reference to.
`binding` Specifies the actual types that all the placeholders should bind to.
Requires: `U` is a model of `Concept`.

`Map` is an MPL map with an entry for every non-deduced placeholder referred to by `Concept`.
Throws: Nothing.

3.
```
any(any< Concept, T > && other);
```

Constructs an `any` from another rvalue reference.

Parameters: `other` The reference to copy.

The object to bind the reference to.
Throws: Nothing. Constructs an `any` from another `any`.

Nothing.

4.
```
template<typename Concept2, typename Tag2>
  any(any< Concept2, Tag2 && > && other);
```

Constructs an `any` from another rvalue reference.

Parameters: `other` The reference to copy.
Requires: `Concept` must not refer to any non-deduced placeholder besides `T`.

After substituting `T` for `Tag2`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.
Throws: std::bad_alloc

5.
```
template<typename Concept2, typename Tag2> any(any< Concept2, Tag2 > && other);
```

Constructs an `any` from another `any`.

Parameters: `other` The object to bind the reference to.
Requires: `Concept` must not refer to any non-deduced placeholder besides `T`.

After substituting `T` for `Tag2`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.
Throws: std::bad_alloc

6.
```
template<typename Concept2, typename Tag2, typename Map>
  any(const any< Concept2, Tag2 && > & other,
      const static_binding< Map > & binding);
```

Constructs an `any` from another reference.

Parameters: `binding` Specifies the mapping between the two concepts.

`other` The reference to copy.

Requires: `Map` must be an MPL map with keys for all the non-deduced placeholders used by `Concept` and values for the corresponding placeholders in `Concept2`.

After substituting placeholders according to `Map`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.

Throws: std::bad_alloc

7.
```
template<typename Concept2, typename Tag2, typename Map>
  any(any< Concept2, Tag2 > && other, const static_binding< Map > & binding);
```

Constructs an `any` from another `any`.

Parameters: `binding` Specifies the mapping between the two concepts.
`other` The object to bind the reference to.

Requires: `Map` must be an MPL map with keys for all the non-deduced placeholders used by `Concept` and values for the corresponding placeholders in `Concept2`.

After substituting placeholders according to `Map`, the requirements of `Concept2` must be a superset of the requirements of `Concept`.

Throws: std::bad_alloc

8.
```
template<typename Concept2, typename Tag2>
  any(const any< Concept2, Tag2 && > & other,
      const binding< Concept > & binding);
```

Constructs an `any` from another rvalue reference.

Parameters: `binding` Specifies the bindings of placeholders to actual types.
`other` The reference to copy.

Requires: The type stored in `other` must match the type expected by `binding`.

Postconditions: binding_of(*this) == `binding`

Throws: Nothing.

9.
```
template<typename Concept2, typename Tag2>
  any(any< Concept2, Tag2 > && other, const binding< Concept > & binding);
```

Constructs an `any` from another `any`.

Parameters: `binding` Specifies the bindings of placeholders to actual types.
`other` The object to bind the reference to.

Requires: The type stored in `other` must match the type expected by `binding`.

Postconditions: binding_of(*this) == `binding`

Throws: Nothing.

10.
```
any& operator=(const any & other);
```

Assigns to an `any`.

If an appropriate overload of `assignable` is not available and `relaxed` is in `Concept`, falls back on constructing from `other`.

Throws: Whatever the assignment operator of the contained type throws. When falling back on construction, throws `std::bad_alloc`. In this case assignment provides the strong exception guarantee. When calling the assignment operator of the contained type, the exception guarantee is whatever the contained type provides.

11.
```
template<typename U> any& operator=(U & other);
```

Assigns to an `any`.

If an appropriate overload of `assignable` is not available and `relaxed` is in `Concept`, falls back on constructing from `other`.

Throws: Whatever the assignment operator of the contained type throws. When falling back on construction, throws `std::bad_alloc`. In this case assignment provides the strong exception guarantee. When calling the assignment operator of the contained type, the exception guarantee is whatever the contained type provides.

12.
```
template<typename U> any& operator=(const U & other);
```

Assigns to an `any`.

If an appropriate overload of `assignable` is not available and `relaxed` is in `Concept`, falls back on constructing from `other`.

Throws: Whatever the assignment operator of the contained type throws. When falling back on construction, throws `std::bad_alloc`. In this case assignment provides the strong exception guarantee. When calling the assignment operator of the contained type, the exception guarantee is whatever the contained type provides.

Header <boost/type_erasure/any_cast.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename T, typename Concept, typename Tag>
      T any_cast(any< Concept, Tag > &);
    template<typename T, typename Concept, typename Tag>
      T any_cast(const any< Concept, Tag > &);
    template<typename T, typename Concept, typename Tag>
      T any_cast(any< Concept, Tag > *);
    template<typename T, typename Concept, typename Tag>
      T any_cast(const any< Concept, Tag > *);
  }
}
```

Function any_cast

boost::type_erasure::any_cast

Synopsis

```
// In header: <boost/type_erasure/any_cast.hpp>


template<typename T, typename Concept, typename Tag>
  T any_cast(any< Concept, Tag > & arg);
template<typename T, typename Concept, typename Tag>
  T any_cast(const any< Concept, Tag > & arg);
template<typename T, typename Concept, typename Tag>
  T any_cast(any< Concept, Tag > * arg);
template<typename T, typename Concept, typename Tag>
  T any_cast(const any< Concept, Tag > * arg);
```

Description

Attempts to extract the object that `arg` holds. If casting to a pointer fails, `any_cast` returns a null pointer. Casting to `void*` always succeeds and returns the address of stored object.

Requires: if `arg` is a pointer, `T` must be a pointer type.

`Concept` must contain typeid_<`Tag`>.

Throws: bad_any_cast if `arg` doesn't contain an object of type `T` and we're casting to a value or reference.

Header <boost/type_erasure/binding.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Concept> class binding;
  }
}
```

Class template binding

boost::type_erasure::binding

Synopsis

```
// In header: <boost/type_erasure/binding.hpp>

template<typename Concept>
class binding {
public:
  // construct/copy/destruct
  binding();
  template<typename Map> explicit binding(const Map &);
  template<typename Map> binding(const static_binding< Map > &);
  template<typename Concept2, typename Map>
    binding(const binding< Concept2 > &, const Map &);
  template<typename Concept2, typename Map>
    binding(const binding< Concept2 > &, const static_binding< Map > &);

  // friend functions
  friend bool operator==(const binding &, const binding &);
  friend bool operator!=(const binding &, const binding &);
};
```

Description

Stores the binding of a `Concept` to a set of actual types. `Concept` is interpreted in the same way as with any.

`binding` public construct/copy/destruct

1.
```
binding();
```

Requires: relaxed must be in `Concept`.
Throws: Nothing.

2.
```
template<typename Map> explicit binding(const Map &);
```

Requires: `Map` must be an MPL map with an entry for each placeholder referred to by `Concept`.
Throws: Nothing.

3.
```
template<typename Map> binding(const static_binding< Map > &);
```

Requires: `Map` must be an MPL map with an entry for each placeholder referred to by `Concept`.

Throws: Nothing.

4.
```
template<typename Concept2, typename Map>
  binding(const binding< Concept2 > & other, const Map &);
```

Converts from another set of bindings.

Requires: Map must be an MPL map with an entry for each placeholder referred to by `Concept`. The mapped type should be the corresponding placeholder in Concept2.
Throws: std::bad_alloc

5.
```
template<typename Concept2, typename Map>
  binding(const binding< Concept2 > & other, const static_binding< Map > &);
```

Converts from another set of bindings.

Requires: Map must be an MPL map with an entry for each placeholder referred to by `Concept`. The mapped type should be the corresponding placeholder in Concept2.
Throws: std::bad_alloc

`binding` friend functions

1.
```
friend bool operator==(const binding & lhs, const binding & rhs);
```

Returns: true iff the sets of types that the placeholders bind to are the same for both arguments.
Throws: Nothing.

2.
```
friend bool operator!=(const binding & lhs, const binding & rhs);
```

Returns: true iff the arguments do not map to identical sets of types.
Throws: Nothing.

Header <boost/type_erasure/binding_of.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Concept, typename T>
      const binding< Concept > & binding_of(const any< Concept, T > &);
  }
}
```

Function template binding_of

boost::type_erasure::binding_of

Synopsis

```
// In header: <boost/type_erasure/binding_of.hpp>


template<typename Concept, typename T>
  const binding< Concept > & binding_of(const any< Concept, T > & arg);
```

Description

Returns: The type bindings of an any.
Throws: Nothing.

Header <boost/type_erasure/builtin.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename T = _self> struct copy_constructible;
    template<typename T = _self> struct typeid_;
  }
}
```

Struct template copy_constructible

boost::type_erasure::copy_constructible

Synopsis

```
// In header: <boost/type_erasure/builtin.hpp>

template<typename T = _self>
struct copy_constructible : public boost::mpl::vector< constructible< T(const T &)>, destruct↵
ible< T > >
{
};
```

Description

The copy_constructible concept allows objects to be copied and destroyed.

Note

This concept is defined to match C++ 2003, [lib.copyconstructible]. It is not equivalent to the concept of the same name in C++11.

Struct template typeid_

boost::type_erasure::typeid_

Synopsis

```
// In header: <boost/type_erasure/builtin.hpp>

template<typename T = _self>
struct typeid_ {
};
```

Description

Enables runtime type information. This is required if you want to use any_cast or typeid_of.

Note

typeid_ cannot be specialized because several library components including any_cast would not work correctly if its behavior changed. There is no need to specialize it anyway, since it works for all types. typeid_ also cannot be passed to call. To access it, use typeid_of.

Header <boost/type_erasure/call.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Concept, typename Op, class... U>
      unspecified call(const binding< Concept > &, const Op &, U &&...);
    template<typename Op, class... U> unspecified call(const Op &, U &&...);
  }
}
```

Function call

boost::type_erasure::call

Synopsis

```
// In header: <boost/type_erasure/call.hpp>


template<typename Concept, typename Op, class... U>
  unspecified call(const binding< Concept > & binding, const Op &,
                   U &&... args);
template<typename Op, class... U> unspecified call(const Op &, U &&... args);
```

Description

Dispatches a type erased function.

`Op` must be a primitive concept which is present in `Concept`. Its signature determines how the arguments of `call` are handled. If the argument is a `placeholder`, `call` expects an `any` using that `placeholder`. This `any` is unwrapped by `call`. The type that it stores must be the same type specified by `binding`. Any arguments that are not placeholders in the signature of `Op` are passed through unchanged.

If `binding` is not specified, it will be deduced from the arguments. Naturally this requires at least one argument to be an `any`. In this case, all `any` arguments must have the same `binding`.

Example:

```
typedef mpl::vector<
  copy_constructible<_b>,
  addable<_a, int, _b> > concept;
any<concept, _a> a = ...;
any<concept, _b> b(call(addable<_a, int, _b>(), a, 10));
```

The signature of `addable` is `_b(const _a&, const int&)`

Returns: The result of the operation. If the result type of the signature of `Op` is a placeholder, the result will be converted to the appropriate any type.

Throws: bad_function_call if relaxed is in `Concept` and there is a type mismatch.

Header <boost/type_erasure/callable.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Sig, typename F = _self> struct callable;
  }
}
```

Struct template callable

boost::type_erasure::callable

Synopsis

```
// In header: <boost/type_erasure/callable.hpp>

template<typename Sig, typename F = _self>
struct callable {

  // public static functions
  static R apply(F &, T...);
};
```

Description

The callable concept allows an any to hold function objects. `Sig` is interpreted in the same way as for Boost.Function, except that the arguments and return type are allowed to be placeholders. `F` must be a placeholder.

Multiple instances of callable can be used simultaneously. Overload resolution works normally. Note that unlike Boost.Function, callable does not provide result_type. It does, however, support `boost::result_of`.

`callable` public static functions

1. ```
   static R apply(F & f, T... arg);
   ```

   `R` is the result type of `Sig` and `T` is the argument types of `Sig`.

Header <boost/type_erasure/check_match.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Concept, typename Op, class... U>
      bool check_match(const binding< Concept > &, const Op &, U &&...);
    template<typename Op, class... U> bool check_match(const Op &, U &&...);
  }
}
```

Function check_match

boost::type_erasure::check_match
```

Synopsis

```
// In header: <boost/type_erasure/check_match.hpp>


template<typename Concept, typename Op, class... U>
  bool check_match(const binding< Concept > & binding, const Op & f,
                   U &&... args);
template<typename Op, class... U> bool check_match(const Op & f, U &&... args);
```

Description

If `relaxed` is in `Concept`, checks whether the arguments to `f` match the types specified by `binding`. If `relaxed` is not in `Concept`, returns true. If `binding` is not specified, it will be deduced from the arguments.

Header <boost/type_erasure/concept_interface.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Concept, typename Base, typename ID,
             typename Enable = void>
      struct concept_interface;
  }
}
```

Struct template concept_interface

boost::type_erasure::concept_interface

Synopsis

```
// In header: <boost/type_erasure/concept_interface.hpp>

template<typename Concept, typename Base, typename ID, typename Enable = void>
struct concept_interface : public Base {
};
```

Description

The concept_interface class can be specialized to add behavior to an any. An any inherits from all the relevant specializations of concept_interface.

concept_interface can be specialized for either primitive or composite concepts. If a concept `C1` contains another concept `C2`, then the library guarantees that the specialization of concept_interface for `C2` is a base class of the specialization for `C1`. This means that `C1` can safely override members of `C2`.

concept_interface may only be specialized for user-defined concepts. The library owns the specializations of its own built in concepts.

The metafunctions derived, rebind_any, and as_param (which can be applied to `Base`) are useful for determining the argument and return types of functions defined in concept_interface.

For dispatching the function use call.

Template Parameters

1. ```
   typename Concept
   ```

   The concept that we're specializing `concept_interface` for. One of its placeholders should be `ID`.

2. ```
   typename Base
   ```

   The base of this class. Specializations of `concept_interface` must inherit publicly from this type.

3. ```
   typename ID
   ```

   The placeholder representing this type.

4. ```
   typename Enable = void
   ```

   A dummy parameter that can be used for SFINAE.

Header <boost/type_erasure/concept_of.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Concept, typename T> class param;

    template<typename T> struct concept_of;
  }
}
```

Class template param

boost::type_erasure::param — A wrapper to help with overload resolution for functions operating on an any.

Synopsis

```
// In header: <boost/type_erasure/concept_of.hpp>

template<typename Concept, typename T>
class param {
public:
  // construct/copy/destruct
  template<typename U> param(any< Concept, U > &);
  template<typename U> param(const any< Concept, U > &);
  template<typename U> param(any< Concept, U > &&);

  // public member functions
  any< Concept, T > get() const;
};
```

Description

The template arguments are interpreted in the same way as any.

A parameter of type param can be initialized with an any that has the same `Concept` and base placeholder when there exists a corresponding standard conversion for the placeholder. A conversion sequence from any<C, P> to param<C, P1> is a better conversion

sequence than any<C, P> to param<C, P2> iff the corresponding placeholder standard conversion sequence from P to P1 is a better conversion sequence than P to P2.

Note

Overloading based on cv-qualifiers and rvalue-ness is only supported in C++11. In C++03, all conversion sequences from any to param have the same rank.

Example:

```
void f(param<C, _a&>);
void f(param<C, const _a&>);
void g(param<C, const _a&>);
void g(param<C, _a&&>);

any<C, _a> a;
f(any<C, _a>()); // calls void f(param<C, const _a&>);
f(a);            // calls void f(param<C, _a&>); (ambiguous in C++03)
g(any<C, _a>()); // calls void g(param<C, _a&&>); (ambiguous in C++03)
g(a);            // calls void g(param<C, const _a&>);
```

`param` public construct/copy/destruct

1. ```
   template<typename U> param(any< Concept, U > & a);
   ```

2. ```
   template<typename U> param(const any< Concept, U > & a);
   ```

3. ```
   template<typename U> param(any< Concept, U > && a);
   ```

`param` public member functions

1. ```
   any< Concept, T > get() const;
   ```

   Returns the stored any.

Struct template concept_of

boost::type_erasure::concept_of

Synopsis

```
// In header: <boost/type_erasure/concept_of.hpp>

template<typename T>
struct concept_of {
  // types
  typedef unspecified type;
};
```

Description

A metafunction returning the concept corresponding to an any. It will also work for all bases of any, so it can be applied to the `Base` parameter of concept_interface.

Header <boost/type_erasure/config.hpp>

```
BOOST_TYPE_ERASURE_MAX_FUNCTIONS
BOOST_TYPE_ERASURE_MAX_ARITY
BOOST_TYPE_ERASURE_MAX_TUPLE_SIZE
```

Macro BOOST_TYPE_ERASURE_MAX_FUNCTIONS

BOOST_TYPE_ERASURE_MAX_FUNCTIONS

Synopsis

```
// In header: <boost/type_erasure/config.hpp>

BOOST_TYPE_ERASURE_MAX_FUNCTIONS
```

Description

The maximum number of functions that an any can have.

Macro BOOST_TYPE_ERASURE_MAX_ARITY

BOOST_TYPE_ERASURE_MAX_ARITY

Synopsis

```
// In header: <boost/type_erasure/config.hpp>

BOOST_TYPE_ERASURE_MAX_ARITY
```

Description

The maximum number of arguments that functions in the library support.

Macro BOOST_TYPE_ERASURE_MAX_TUPLE_SIZE

BOOST_TYPE_ERASURE_MAX_TUPLE_SIZE

Synopsis

```
// In header: <boost/type_erasure/config.hpp>

BOOST_TYPE_ERASURE_MAX_TUPLE_SIZE
```

Description

The maximum number of elements in a tuple.

Header <boost/type_erasure/constructible.hpp>

Header <boost/type_erasure/deduced.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Metafunction> struct deduced;
  }
}
```

Struct template deduced

boost::type_erasure::deduced

Synopsis

```
// In header: <boost/type_erasure/deduced.hpp>

template<typename Metafunction>
struct deduced : public boost::type_erasure::placeholder {
  // types
  typedef unspecified type;
};
```

Description

A placeholder for an associated type. The type corresponding to this placeholder is deduced by substituting placeholders in the arguments of the metafunction and then evaluating it.

When using deduced in a template context, if it is possible for Metafunction to contain no placeholders at all, use the nested type, to automatically evaluate it early as needed.

Header <boost/type_erasure/derived.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename T> struct derived;
  }
}
```

Struct template derived

boost::type_erasure::derived

Synopsis

```
// In header: <boost/type_erasure/derived.hpp>

template<typename T>
struct derived {
  // types
  typedef unspecified type;
};
```

Description

A metafunction which returns the full any type, when given any of its base classes. This is primarily intended to be used when implementing concept_interface.

See Also:

rebind_any, as_param

Header <boost/type_erasure/exception.hpp>

```
namespace boost {
  namespace type_erasure {
    class bad_function_call;
    class bad_any_cast;
  }
}
```

Class bad_function_call

boost::type_erasure::bad_function_call

Synopsis

```
// In header: <boost/type_erasure/exception.hpp>


class bad_function_call : public invalid_argument {
public:
  // construct/copy/destruct
  bad_function_call();
};
```

Description

Exception thrown when the arguments to a primitive concept are incorrect.

See Also:

call, require_match

`bad_function_call` public construct/copy/destruct

1. ```
   bad_function_call();
   ```
   ```

Class bad_any_cast

boost::type_erasure::bad_any_cast

Synopsis

```
// In header: <boost/type_erasure/exception.hpp>


class bad_any_cast : public bad_cast {
};
```

Description

Exception thrown when an any_cast to a reference or value fails.

Header <boost/type_erasure/free.hpp>

```


BOOST_TYPE_ERASURE_FREE(qualified_name, function_name, N)
```

Macro BOOST_TYPE_ERASURE_FREE

BOOST_TYPE_ERASURE_FREE — Defines a primitive concept for a free function.

Synopsis

```
// In header: <boost/type_erasure/free.hpp>

BOOST_TYPE_ERASURE_FREE(qualified_name, function_name, N)
```

Description

The declaration of the concept is

```
template<class Sig>
struct ::namespace1::namespace2::...::concept_name;
```

where Sig is a function type giving the signature of the function.

This macro can only be used in the global namespace.

Example:

```
BOOST_TYPE_ERASURE_FREE((boost)(has_to_string), to_string, 1)
```

Parameters:		
	`N`	is the number of arguments of the function.
	`function_name`	is the name of the function.
	`qualified_name`	should be a preprocessor sequence of the form (namespace1)(namespace2)...(concept_name).

Header <boost/type_erasure/is_empty.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename T> bool is_empty(const T &);
  }
}
```

Function template is_empty

boost::type_erasure::is_empty

Synopsis

```
// In header: <boost/type_erasure/is_empty.hpp>


template<typename T> bool is_empty(const T & arg);
```

Description

Returns true for an empty any.

Header <boost/type_erasure/is_placeholder.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename T> struct is_placeholder;
  }
}
```

Struct template is_placeholder

boost::type_erasure::is_placeholder

Synopsis

```
// In header: <boost/type_erasure/is_placeholder.hpp>

template<typename T>
struct is_placeholder : public boost::is_base_and_derived< placeholder, T > {
};
```

Description

A metafunction that indicates whether a type is a placeholder.

Header <boost/type_erasure/is_subconcept.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Sub, typename Super, typename PlaceholderMap = void>
      struct is_subconcept;
  }
}
```

Struct template is_subconcept

boost::type_erasure::is_subconcept

Synopsis

```
// In header: <boost/type_erasure/is_subconcept.hpp>

template<typename Sub, typename Super, typename PlaceholderMap = void>
struct is_subconcept {
};
```

Description

is_subconcept is a boolean metafunction that determines whether one concept is a sub-concept of another.

```
is_subconcept<incrementable<>, incrementable<> >             -> true
is_subconcept<incrementable<>, addable<> >                   -> false
is_subconcept<incrementable<_a>, forward_iterator<_iter>,
  mpl::map<mpl::pair<_a, _iter> > >                          -> true
```

Template Parameters

1. ```
   typename Sub
   ```

   The sub concept

2. ```
   typename Super
   ```

   The super concept

3. ```
   typename PlaceholderMap = void
   ```

   (optional) An MPL map with keys for every non-deduced placeholder in Sub. The associated value of each key is the corresponding placeholder in Super. If `PlaceholderMap` is omitted, `Super` and `Sub` are presumed to use the same set of placeholders.
   ```

Header <boost/type_erasure/iterator.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Traversal, typename T = _self,
             typename Reference = ::boost::use_default,
             typename DifferenceType = ::std::ptrdiff_t,
             typename ValueType = typename deduced<iterator_value_type<T> >::type>
      struct iterator;
    template<typename T = _self, typename Reference = boost::use_default,
             typename DifferenceType = std::ptrdiff_t>
      struct forward_iterator;
    template<typename T = _self, typename Reference = boost::use_default,
             typename DifferenceType = std::ptrdiff_t>
      struct bidirectional_iterator;
    template<typename T = _self, typename Reference = boost::use_default,
             typename DifferenceType = std::ptrdiff_t>
      struct random_access_iterator;
  }
}
```

Struct template iterator

boost::type_erasure::iterator

Synopsis

```
// In header: <boost/type_erasure/iterator.hpp>

template<typename Traversal, typename T = _self,
         typename Reference = ::boost::use_default,
         typename DifferenceType = ::std::ptrdiff_t,
         typename ValueType = typename deduced<iterator_value_type<T> >::type>
struct iterator {
  // types
  typedef unspecified    value_type;
  typedef Reference      reference;
  typedef DifferenceType difference_type;
};
```

Description

The iterator concept can be used for any iterator category.

The value_type of the iterator is deduced. To force it to be a specific type, use the same_type concept.

Example:

```
mpl::vector<
  iterator<boost::forward_traversal_tag>,
  same_type<iterator<boost::forward_traversal_tag>::value_type, int> > int_it;
```

Template Parameters

1. ```
   typename Traversal
   ```
   ```

must be one of `boost::incrementable_traversal_tag`, `boost::single_pass_traversal_tag`, `boost::forward_traversal_tag`, `boost::bidirectional_traversal_tag`, and `boost::random_access_traversal_tag`.

2. ```
   typename T = _self
   ```

   The placeholder representing the iterator.

3. ```
   typename Reference = ::boost::use_default
   ```

   The reference type. If it is boost::use_default, then reference will be value_type&.

4. ```
   typename DifferenceType = ::std::ptrdiff_t
   ```

   The iterator's difference type.

5. ```
   typename ValueType = typename deduced<iterator_value_type<T> >::type
   ```

Struct template forward_iterator

boost::type_erasure::forward_iterator

Synopsis

```
// In header: <boost/type_erasure/iterator.hpp>

template<typename T = _self, typename Reference = boost::use_default,
         typename DifferenceType = std::ptrdiff_t>
struct forward_iterator : public boost::type_erasure::iterator< boost::forward_traversal_tag, ↵
T, Reference, DifferenceType >
{
};
```

Struct template bidirectional_iterator

boost::type_erasure::bidirectional_iterator

Synopsis

```
// In header: <boost/type_erasure/iterator.hpp>

template<typename T = _self, typename Reference = boost::use_default,
         typename DifferenceType = std::ptrdiff_t>
struct bidirectional_iterator : public boost::type_erasure::iterator< boost::bidirectional_tra↵
versal_tag, T, Reference, DifferenceType >
{
};
```

Struct template random_access_iterator

boost::type_erasure::random_access_iterator

Synopsis

```
// In header: <boost/type_erasure/iterator.hpp>

template<typename T = _self, typename Reference = boost::use_default,
         typename DifferenceType = std::ptrdiff_t>
struct random_access_iterator : public boost::type_erasure::iterator< boost::random_access_tra↵
versal_tag, T, Reference, DifferenceType >
{
};
```

Header <boost/type_erasure/member.hpp>

```
BOOST_TYPE_ERASURE_MEMBER(qualified_name, member, N)
```

Macro BOOST_TYPE_ERASURE_MEMBER

BOOST_TYPE_ERASURE_MEMBER — Defines a primitive concept for a member function.

Synopsis

```
// In header: <boost/type_erasure/member.hpp>

BOOST_TYPE_ERASURE_MEMBER(qualified_name, member, N)
```

Description

The declaration of the concept is

```
template<class Sig, class T = _self>
struct ::namespace1::namespace2::...::concept_name;
```

where Sig is a function type giving the signature of the member function, and T is the object type. T may be const-qualified for const member functions.

This macro can only be used in the global namespace.

Example:

```
BOOST_TYPE_ERASURE_MEMBER((boost)(has_push_back), push_back, 1)
typedef boost::has_push_back<void(int), _self> push_back_concept;
```

Note

In C++11 the argument N is ignored and may be omitted. BOOST_TYPE_ERASURE_MEMBER will always define a variadic concept.

Parameters:

`N`	is the number of arguments of the function.
`member`	is the name of the member function.
`qualified_name`	should be a preprocessor sequence of the form (namespace1)(namespace2)...(concept_name).

Header <boost/type_erasure/operators.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename T = _self> struct incrementable;
    template<typename T = _self> struct decrementable;
    template<typename T = _self, typename R = T> struct complementable;
    template<typename T = _self, typename R = T> struct negatable;
    template<typename R, typename T = _self> struct dereferenceable;
    template<typename T = _self, typename U = T, typename R = T> struct addable;
    template<typename T = _self, typename U = T, typename R = T>
      struct subtractable;
    template<typename T = _self, typename U = T, typename R = T>
      struct multipliable;
    template<typename T = _self, typename U = T, typename R = T>
      struct dividable;
    template<typename T = _self, typename U = T, typename R = T> struct modable;
    template<typename T = _self, typename U = T, typename R = T>
      struct left_shiftable;
    template<typename T = _self, typename U = T, typename R = T>
      struct right_shiftable;
    template<typename T = _self, typename U = T, typename R = T>
      struct bitandable;
    template<typename T = _self, typename U = T, typename R = T>
      struct bitorable;
    template<typename T = _self, typename U = T, typename R = T>
      struct bitxorable;
    template<typename T = _self, typename U = T> struct add_assignable;
    template<typename T = _self, typename U = T> struct subtract_assignable;
    template<typename T = _self, typename U = T> struct multiply_assignable;
    template<typename T = _self, typename U = T> struct divide_assignable;
    template<typename T = _self, typename U = T> struct mod_assignable;
    template<typename T = _self, typename U = T> struct left_shift_assignable;
    template<typename T = _self, typename U = T> struct right_shift_assignable;
    template<typename T = _self, typename U = T> struct bitand_assignable;
    template<typename T = _self, typename U = T> struct bitor_assignable;
    template<typename T = _self, typename U = T> struct bitxor_assignable;
    template<typename T = _self, typename U = T> struct equality_comparable;
    template<typename T = _self, typename U = T> struct less_than_comparable;
    template<typename R, typename T = _self, typename N = std::ptrdiff_t>
      struct subscriptable;
    template<typename Os = std::ostream, typename T = _self> struct ostreamable;
    template<typename Is = std::istream, typename T = _self> struct istreamable;
  }
}
```

Struct template incrementable

boost::type_erasure::incrementable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self>
struct incrementable {

  // public static functions
  static void apply(T &);
};
```

Description

The incrementable concept allow pre and post increment on an any. The contained type must provide a pre-increment operator.

`incrementable` public static functions

1. `static void apply(T &);`

Struct template decrementable

boost::type_erasure::decrementable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self>
struct decrementable {

  // public static functions
  static void apply(T &);
};
```

Description

The decrementable concept allow pre and post decrement on an any. The contained type must provide a pre-decrement operator.

`decrementable` public static functions

1. `static void apply(T &);`

Struct template complementable

boost::type_erasure::complementable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename R = T>
struct complementable {

  // public static functions
  static R apply(const T &);
};
```

Description

The complementable concept allow use of the bitwise complement operator on an any.

`complementable` public static functions

1. `static R apply(const T &);`

Struct template negatable

boost::type_erasure::negatable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename R = T>
struct negatable {

  // public static functions
  static R apply(const T &);
};
```

Description

The negatable concept allow use of the unary minus operator on an any.

`negatable` public static functions

1. `static R apply(const T &);`

Struct template dereferenceable

boost::type_erasure::dereferenceable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename R, typename T = _self>
struct dereferenceable {

  // public static functions
  static R apply(const T &);
};
```

Description

`dereferenceable` public static functions

1. `static R apply(const T & arg);`

Struct template addable

boost::type_erasure::addable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T, typename R = T>
struct addable {

  // public static functions
  static R apply(const T &, const U &);
};
```

Description

`addable` public static functions

1. `static R apply(const T &, const U &);`

Struct template subtractable

boost::type_erasure::subtractable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T, typename R = T>
struct subtractable {

  // public static functions
  static R apply(const T &, const U &);
};
```

Description

`subtractable` public static functions

1.
```
static R apply(const T &, const U &);
```

Struct template multipliable

boost::type_erasure::multipliable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T, typename R = T>
struct multipliable {

  // public static functions
  static R apply(const T &, const U &);
};
```

Description

`multipliable` public static functions

1.
```
static R apply(const T &, const U &);
```

Struct template dividable

boost::type_erasure::dividable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T, typename R = T>
struct dividable {

  // public static functions
  static R apply(const T &, const U &);
};
```

Description

`dividable` public static functions

1.
```
static R apply(const T &, const U &);
```

Struct template modable

boost::type_erasure::modable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T, typename R = T>
struct modable {

  // public static functions
  static R apply(const T &, const U &);
};
```

Description

`modable` public static functions

1. `static R apply(const T &, const U &);`

Struct template left_shiftable

boost::type_erasure::left_shiftable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T, typename R = T>
struct left_shiftable {

  // public static functions
  static R apply(const T &, const U &);
};
```

Description

`left_shiftable` public static functions

1. `static R apply(const T &, const U &);`

Struct template right_shiftable

boost::type_erasure::right_shiftable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T, typename R = T>
struct right_shiftable {

  // public static functions
  static R apply(const T &, const U &);
};
```

Description

`right_shiftable` public static functions

1. `static R apply(const T &, const U &);`

Struct template bitandable

boost::type_erasure::bitandable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T, typename R = T>
struct bitandable {

  // public static functions
  static R apply(const T &, const U &);
};
```

Description

`bitandable` public static functions

1. `static R apply(const T &, const U &);`

Struct template bitorable

boost::type_erasure::bitorable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T, typename R = T>
struct bitorable {

  // public static functions
  static R apply(const T &, const U &);
};
```

Description

`bitorable` public static functions

1. ```
   static R apply(const T &, const U &);
   ```

Struct template bitxorable

boost::type_erasure::bitxorable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T, typename R = T>
struct bitxorable {

  // public static functions
  static R apply(const T &, const U &);
};
```

Description

`bitxorable` public static functions

1. ```
   static R apply(const T &, const U &);
   ```

Struct template add_assignable

boost::type_erasure::add_assignable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct add_assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

`add_assignable` public static functions

1. ```
   static void apply(T &, const U &);
   ```
   ```

Struct template subtract_assignable

boost::type_erasure::subtract_assignable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct subtract_assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

`subtract_assignable` public static functions

1. `static void apply(T &, const U &);`

Struct template multiply_assignable

boost::type_erasure::multiply_assignable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct multiply_assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

`multiply_assignable` public static functions

1. `static void apply(T &, const U &);`

Struct template divide_assignable

boost::type_erasure::divide_assignable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct divide_assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

`divide_assignable` public static functions

1. `static void apply(T &, const U &);`

Struct template mod_assignable

boost::type_erasure::mod_assignable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct mod_assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

`mod_assignable` public static functions

1. `static void apply(T &, const U &);`

Struct template left_shift_assignable

boost::type_erasure::left_shift_assignable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct left_shift_assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

`left_shift_assignable` public static functions

1.
```
static void apply(T &, const U &);
```

Struct template right_shift_assignable

boost::type_erasure::right_shift_assignable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct right_shift_assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

`right_shift_assignable` public static functions

1.
```
static void apply(T &, const U &);
```

Struct template bitand_assignable

boost::type_erasure::bitand_assignable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct bitand_assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

`bitand_assignable` public static functions

1.
```
static void apply(T &, const U &);
```

Struct template bitor_assignable

boost::type_erasure::bitor_assignable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct bitor_assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

`bitor_assignable` public static functions

1. `static void apply(T &, const U &);`

Struct template bitxor_assignable

boost::type_erasure::bitxor_assignable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct bitxor_assignable {

  // public static functions
  static void apply(T &, const U &);
};
```

Description

`bitxor_assignable` public static functions

1. `static void apply(T &, const U &);`

Struct template equality_comparable

boost::type_erasure::equality_comparable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct equality_comparable {

  // public static functions
  static bool apply(const T &, const U &);
};
```

Description

`equality_comparable` public static functions

1. `static bool apply(const T & lhs, const U & rhs);`

Struct template less_than_comparable

boost::type_erasure::less_than_comparable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename T = _self, typename U = T>
struct less_than_comparable {

  // public static functions
  static bool apply(const T &, const U &);
};
```

Description

`less_than_comparable` public static functions

1. `static bool apply(const T & lhs, const U & rhs);`

Struct template subscriptable

boost::type_erasure::subscriptable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename R, typename T = _self, typename N = std::ptrdiff_t>
struct subscriptable {

  // public static functions
  static R apply(T &, const N &);
};
```

Description

`subscriptable` public static functions

1. ```
   static R apply(T & arg, const N & index);
   ```

Struct template ostreamable

boost::type_erasure::ostreamable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename Os = std::ostream, typename T = _self>
struct ostreamable {

  // public static functions
  static void apply(Os &, const T &);
};
```

Description

The ostreamable concept allows an any to be written to a `std::ostream`.

`ostreamable` public static functions

1. ```
   static void apply(Os & out, const T & arg);
   ```

Struct template istreamable

boost::type_erasure::istreamable

Synopsis

```
// In header: <boost/type_erasure/operators.hpp>

template<typename Is = std::istream, typename T = _self>
struct istreamable {

  // public static functions
  static void apply(Is &, T &);
};
```

Description

The istreamable concept allows an any to be read from a `std::istream`.

`istreamable` public static functions

1. ```
   static void apply(Is & out, T & arg);
   ```
   ```

Header <boost/type_erasure/param.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Any, typename T> struct as_param;
  }
}
```

Struct template as_param

boost::type_erasure::as_param — Metafunction that creates a param.

Synopsis

```
// In header: <boost/type_erasure/param.hpp>

template<typename Any, typename T>
struct as_param {
  // types
  typedef unspecified type;
};
```

Description

If `T` is a (cv/reference qualifed) placeholder, returns param<concept_of<Any>::type, T>, otherwise, returns T. This metafunction is intended to be used for function arguments in specializations of concept_interface.

See Also:

derived, rebind_any

Header <boost/type_erasure/placeholder.hpp>

```
namespace boost {
  namespace type_erasure {
    struct placeholder;
    struct _a;
    struct _b;
    struct _c;
    struct _d;
    struct _e;
    struct _f;
    struct _g;
    struct _self;
  }
}
```

Struct placeholder

boost::type_erasure::placeholder

Synopsis

```
// In header: <boost/type_erasure/placeholder.hpp>


struct placeholder {
};
```

Description

Placeholders are used heavily throughout the library. Every placeholder must derive from placeholder. The library provides a number of placeholders, out of the box, but you are welcome to define your own, if you want more descriptive names. The placeholder _self is special in that it is used as the default wherever possible.

What exactly is a placeholder? Placeholders act as a substitute for template parameters in concepts. The library automatically replaces all the placeholders used in a concept with the actual types involved when it stores an object in an any.

For example, in the following,

```
any<copy_constructible<_a>, _a> x(1);
```

The library sees that we're constructing an any that uses the _a placeholder with an `int`. Thus it binds _a to int and instantiates copy_constructible<int>.

When there are multiple placeholders involved, you will have to use tuple, or pass the bindings explicitly, but the substitution still works the same way.

Struct _a

boost::type_erasure::_a

Synopsis

```
// In header: <boost/type_erasure/placeholder.hpp>


struct _a : public boost::type_erasure::placeholder {
};
```

Struct _b

boost::type_erasure::_b

Synopsis

```
// In header: <boost/type_erasure/placeholder.hpp>


struct _b : public boost::type_erasure::placeholder {
};
```

Struct _c

boost::type_erasure::_c

Synopsis

```
// In header: <boost/type_erasure/placeholder.hpp>


struct _c : public boost::type_erasure::placeholder {
};
```

Struct _d

boost::type_erasure::_d

Synopsis

```
// In header: <boost/type_erasure/placeholder.hpp>


struct _d : public boost::type_erasure::placeholder {
};
```

Struct _e

boost::type_erasure::_e

Synopsis

```
// In header: <boost/type_erasure/placeholder.hpp>


struct _e : public boost::type_erasure::placeholder {
};
```

Struct _f

boost::type_erasure::_f

Synopsis

```
// In header: <boost/type_erasure/placeholder.hpp>


struct _f : public boost::type_erasure::placeholder {
};
```

Struct _g

boost::type_erasure::_g

Synopsis

```
// In header: <boost/type_erasure/placeholder.hpp>


struct _g : public boost::type_erasure::placeholder {
};
```

Struct _self

boost::type_erasure::_self — The default placeholder.

Synopsis

```
// In header: <boost/type_erasure/placeholder.hpp>


struct _self : public boost::type_erasure::placeholder {
};
```

Description

_self is the default placeholder used by any. It should be used as a default by most concepts, so using concepts with no explicit arguments will "just work" as much as possible.

Header <boost/type_erasure/placeholder_of.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename T> struct placeholder_of;
  }
}
```

Struct template placeholder_of

boost::type_erasure::placeholder_of

Synopsis

```
// In header: <boost/type_erasure/placeholder_of.hpp>

template<typename T>
struct placeholder_of {
  // types
  typedef unspecified type;
};
```

Description

A metafunction returning the (const/reference qualified) placeholder corresponding to an any. It will also work for all bases of any, so it can be applied to the `Base` parameter of concept_interface.

Header <boost/type_erasure/rebind_any.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Any, typename T> struct rebind_any;
  }
}
```

Struct template rebind_any

boost::type_erasure::rebind_any

Synopsis

```
// In header: <boost/type_erasure/rebind_any.hpp>

template<typename Any, typename T>
struct rebind_any {
  // types
  typedef unspecified type;
};
```

Description

A metafunction that changes the placeholder of an any. If `T` is not a placeholder, returns `T` unchanged. This class is intended to be used in concept_interface to deduce the argument types from the arguments of the concept.

```
rebind_any<any<Concept>, _a>::type -> any<Concept, _a>
rebind_any<any<Concept>, _b&>::type -> any<Concept, _b&>
rebind_any<any<Concept>, int>::type -> int
```

See Also:

derived, as_param

Header <boost/type_erasure/relaxed.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename T> struct is_relaxed;
    struct relaxed;
  }
}
```

Struct template is_relaxed

boost::type_erasure::is_relaxed

Synopsis

```
// In header: <boost/type_erasure/relaxed.hpp>

template<typename T>
struct is_relaxed {
};
```

Description

A metafunction indicating whether `Concept` includes relaxed.

Struct relaxed

boost::type_erasure::relaxed

Synopsis

```
// In header: <boost/type_erasure/relaxed.hpp>


struct relaxed : public boost::mpl::vector0<> {
};
```

Description

This special concept enables various useful default behavior that makes any act like an ordinary object. By default any forwards all operations to the underlying type, and provides only the operations that are specified in its `Concept`.

In detail, relaxed enables the following:

- A raw value can be assigned to an any. This will replace the value stored by the any. (But note that if assignable is present, it takes priority.)
- copy assignment of any uses the copy constructor if it can't use assignable (either because assignable is missing, or because the stored types do not match).
- default construction of any is allowed and creates a null any.
- equality_comparable: If the types do not match, it will return false.
- less_than_comparable: If the types do not match, the ordering will be according to `std::type_info::before`.
- if the arguments to any other function do not match, it will throw a bad_function_call exception instead of having undefined behavior.

Header <boost/type_erasure/require_match.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Concept, typename Op, class... U>
      void require_match(const binding< Concept > &, const Op &, U &&...);
    template<typename Op, class... U> void require_match(const Op &, U &&...);
  }
}
```

Function require_match

boost::type_erasure::require_match

Synopsis

```
// In header: <boost/type_erasure/require_match.hpp>


template<typename Concept, typename Op, class... U>
  void require_match(const binding< Concept > & binding, const Op & f,
                     U &&... args);
template<typename Op, class... U>
  void require_match(const Op & f, U &&... args);
```

Description

Checks that the actual types stored in all the `any` arguments match the types specified by `binding`. If they do not match then,

- If `relaxed` is in `Concept`, throws `bad_function_call`.
- Otherwise the behavior is undefined.

If `binding` is not specified, it will be deduced from the arguments.

Postconditions: `call(binding, f, args...)` is valid.

Header <boost/type_erasure/same_type.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename T, typename U> struct same_type;
  }
}
```

Struct template same_type

boost::type_erasure::same_type

Synopsis

```
// In header: <boost/type_erasure/same_type.hpp>

template<typename T, typename U>
struct same_type {
};
```

Description

A built in concept that indicates that two types are the same. Either T or U or both can be placeholders.

Warning

Any number of instances of deduced can be connected with same_type, but there should be at most one regular placeholder in the group. same_type<_a, _b> is not allowed. The reason for this is that the library needs to normalize all the placeholders, and in this context there is no way to decide whether to use _a or _b.

Header <boost/type_erasure/static_binding.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Map> struct static_binding;
    template<typename Map> static_binding< Map > make_binding();
  }
}
```

Struct template static_binding

boost::type_erasure::static_binding

Synopsis

```
// In header: <boost/type_erasure/static_binding.hpp>

template<typename Map>
struct static_binding {
};
```

Description

Represents a mapping from placeholders to the actual types that they bind to.

Function template make_binding

boost::type_erasure::make_binding

Synopsis

```
// In header: <boost/type_erasure/static_binding.hpp>


template<typename Map> static_binding< Map > make_binding();
```

Description

A convenience function to prevent constructor calls from being parsed as function declarations.

Header <boost/type_erasure/tuple.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Concept, class... T> class tuple;
    template<int N, typename Concept, class... T>
      any< Concept, TN > & get(tuple< Concept, T...> &);
    template<int N, typename Concept, class... T>
      const any< Concept, TN > & get(const tuple< Concept, T...> &);
  }
}
```

Class template tuple

boost::type_erasure::tuple

Synopsis

```
// In header: <boost/type_erasure/tuple.hpp>

template<typename Concept, class... T>
class tuple {
public:
  // construct/copy/destruct
  template<class... U> explicit tuple(U &&...);
};
```

Description

tuple is a Boost.Fusion Random Access Sequence containing anys. `Concept` specifies the Concept for each of the elements. The remaining arguments must be (possibly const and/or reference qualified) placeholders, which are the placeholders of the elements.

`tuple` public construct/copy/destruct

1. ```
   template<class... U> explicit tuple(U &&... args);
   ```

   Constructs a tuple. Each element of `args` will be used to initialize the corresponding `any` member. The `binding` for the tuple elements is determined by mapping the placeholders in `T` to the corresponding types in `U`.

Function get

boost::type_erasure::get

Synopsis

```
// In header: <boost/type_erasure/tuple.hpp>


template<int N, typename Concept, class... T>
  any< Concept, TN > & get(tuple< Concept, T...> & arg);
template<int N, typename Concept, class... T>
  const any< Concept, TN > & get(const tuple< Concept, T...> & arg);
```
```

Description

Returns the Nth any in the tuple.

Header <boost/type_erasure/typeid_of.hpp>

```
namespace boost {
  namespace type_erasure {
    template<typename Concept, typename T>
      const std::type_info & typeid_of(const any< Concept, T > &);
    template<typename T, typename Concept>
      const std::type_info & typeid_of(const binding< Concept > &);
  }
}
```

Function typeid_of

boost::type_erasure::typeid_of

Synopsis

```
// In header: <boost/type_erasure/typeid_of.hpp>


template<typename Concept, typename T>
  const std::type_info & typeid_of(const any< Concept, T > & arg);
template<typename T, typename Concept>
  const std::type_info & typeid_of(const binding< Concept > & binding);
```

Description

The first form returns the type currently stored in an any.

The second form returns the type corresponding to a placeholder in `binding`.

Requires: `Concept` includes typeid_<T>.

`T` is a non-reference, CV-unqualified placeholder.

Rationale

Why do I have to specify the presence of a destructor explicitly?

When using references the destructor isn't needed. By not assuming it implicitly, we allow capturing types with private or protected destructors by reference. For the sake of consistency, it must be specified when capturing by value as well.

Why non-member functions?

The members of `any` can be customized. By using free functions, we guarantee that we don't interfere with anything that a user might want.

Why are the placeholders called `_a`, `_b` and not `_1` `_2`

An earlier version of the library used the names `_1`, `_2`, etc. instead of `_a`, `_b`, etc. This caused a certain amount of confusion because the numbered placeholders are already used with a somewhat different meaning by several other libraries including Boost/Std Bind, Boost.Phoenix, and Boost.MPL. I eventually decided that since the placeholders represented named parameters instead of positional parameters, letters were more appropriate than numbers.

Why not use `boost::ref` for references?

Boost.Function allows you to use `boost::ref` to store a reference to a function object. However, in the general case treating references and values in the same way causes inconsistent behavior that is difficult to reason about. If Boost.TypeErasure handled references like this, then, when you copy an `any`, you would have no idea whether the new object is a real copy or just a new reference to the same underlying object. Boost.Function can get away with it, because it doesn't expose any mutating operations on the stored function object.

Another method that has been proposed is only to keep a reference the first time.

```
int i = 2;
any x = ref(i);
any y = x; // makes a copy
```

Unfortunately, this doesn't handle all use cases, as there is no reliable way to return such a reference from a function. In addition it adds overhead whether it's needed or not, as we would have to add a flag to any to keep track of whether or not it is storing a reference. (The alternate method of storing this in the "`clone`" method in the vtable is impossibly complex to implement given the decoupled vtables that Boost.TypeErasure uses and it still adds overhead.).

Future Work

These are just some ideas. There is absolutely no guarantee that any of them will ever be implemented.

- Use SBO.
- Allow more control over vtable layout.
- Attempt to reuse sub-tables in conversions.
- Allow "dynamic_cast". This requires creating a global registry of concept mappings.
- Optimize the compile-time cost.

Acknowledgements

The name `any` and an early ancestor of my placeholder system were taken from Alexander Nasonov's DynamicAny library.

Thanks to review manager, Lorenzo Caminiti and all who participated in the formal review:

- Christophe Henry
- Paul Bristow
- Karsten Ahnert
- Pete Bartlett
- Sebastian Redl
- Hossein Haeri
- Trigve Siver
- Julien Nitard
- Eric Niebler
- Fabio Fracassi
- Joel de Guzman
- Alec Chapman
- Larry Evans
- Vincente J. Botet Escriba
- Marcus Werle
- Andrey Semashev
- Dave Abrahams
- Thomas Jordan

Related Work

There are a number of similar libraries in existence. I'm aware of at least three.

- Boost.Interfaces by Jonathan Turkanis
- Adobe Poly
- Boost.dynamic_any by Alexander Nasonov