Document number:	P0042
Date:	2015–09–27
To:	SC22/WG21 LEWG
Reply to:	David Krauss
	(david_work at me dot com)

`std::recover`: undoing type erasure

Type erasure is handling an object from a vague type such as a polymorphic base pointer, a `union`, or a `void *`. Such handles provide a subset of the functionality of the original value. Often the original value must be recovered for its full functionality (or any functionality at all). Classes like `std::function`, `any`, `variant`, and `optional` do more, but still provide some recovery interface. Unfortunately, these interfaces have not converged. The proposed functions `recover` and `try_recover` unify access to such "sum type" classes just as `std::get` unifies access to "product type" or `std::tuple`-like classes. RTTI semantics are applied to likewise treat pointers and smart pointers to polymorphic types. A common ErasureClass concept helps to simplify implementation of new client classes, and even to improve the performance of existing `std::function` classes.

1. Motivation

Safely recovering a type-erased value is the same regardless of the handle class semantics. The handle knows the value's original type, and the call context has an educated guess as to that type. An accessor function is called to verify the type and permit access to the object, or else signal a bad cast. Ideally, const-qualification and value category should be preserved. Non-throwing and non-checking alternatives are nice to have.

Several handle classes are either standardized or in the pipeline. The functionality and naming are similar, but their semantic details and level of functionality vary remarkably.

- `std::function` provides a member function template `T* target<T>()`. `const`-ness is propagated but rvalue-ness is cannot be. Failure is signaled by a null pointer; no exception is available.
- `fundamentals_v1::any` provides a non-member `T any_cast<T>(value)`, which can return a reference (lvalue or rvalue), a pointer, or a `T` prvalue. It essentially models recovery of the given object type, followed by a `static_cast` to the exact template argument type. Thus, it does not propagate `const` nor value category, but it can implement an lvalue-to-rvalue cast a la `std::move`. Failure is indicated by `bad_any_cast`, or `nullptr` in the pointer form.
- `variant` in N4542 overloads `std::get`, propagating `const` and value category as does `get<T>(tuple)` — but not both (as in `const&&`). Unlike `tuple`, it supports pointers and throws `bad_variant_access`. A bad pointer cast yields `nullptr`. Retrieving a reference type indicates that the erased type is a reference, not `static_cast` semantics like `any`.
- `fundamentals_v1::optional` may be considered as a limited case. It represents the "sum" of the given type and `void`. It provides a member function `value` propagating `const` and

value category (or even both). Failure is punishable by `bad_optional_access`. There is an unsafe, pointer-like accessor interface accessor in the form of `operator->` and `operator*`, with the latter also propagating `const` and value category (unlike pointers).

Most of the differences between these interfaces do not reflect differences between the classes. The library does not need a separate `bad_*_access` exception type per erasure class. Translation of a handle back to an object should follow a safe, uniform, easily-memorable pattern. Erasure types should be generically substitutable: `variant` is just a fast and restrictive `any`, `function` is merely `any` with a call operator, and `optional` is a sugared `variant` over one type. Although it's uncommon to need such generality in function templates, in refactoring programmers often switch one vocabulary type for another.

Besides inconvenience, the diverging interfaces represent bloat in the library implementation, since little functionality can be shared between them. They also reflect excessive design effort and development churn, which slow the standardization process.

2. Semantics

The underlying model is of a memory blob that may or may not hold a value of a given type. An unsafe handle to such a blob is a `void *`. Safety is regained by stating an expected type and determining whether it matches the type of the value last placed into the blob. Then, the type and the `void *` value may be combined with qualifiers (`const`, `volatile`, `&`, `&&`) to yield a contextually-appropriate value of object type. Overall, the function `recover` behaves as an accessor.

For value-semantic classes like `any`, the expected type is usually an object type. In such cases, `recover` yields a reference to the complete object directly and uniquely owned by the handle. Recovering an lvalue reference (`recover<T &>`) yields a value not owned by the handle, which may be a subobject. A reference-erasing type may offer a special guarantee, for example, that it recovers a most-derived object (e.g. polymorphic raw pointers or `observer_ptr<T>`), or it may simply fetch the glvalue assigned to it (e.g. `variant<T &>`). In any case, the const qualification of a recovered reference is unrelated to that of the non-owning handle. Recovering an rvalue reference expresses indirect ownership: it propagates value category as object types do, but not const qualification.

If the actual type is not as expected, a `bad_recovery_access` exception is thrown or a null reference is returned. The latter allows efficiently testing a sequence of cases in a type-switch.

Each class specifies what types it can erase. For example, `function` allows certain object types, but never references, which are preempted by `reference_wrapper`. An observer such as `observer_ptr` may allow certain reference types, but never objects, because it cannot model ownership. Any assortment of types can be distinguished by `variant`.

Bare `void *` by itself falls short because it completely forgets the actual type of its referent. The model *does* allow a class to match several alternative expected types to one blob, but only as long as the expected types all certainly have objects at the same address. This is seldom satisfiable from the perspective of a generic wrapper. However, a user with knowledge of the erased type may fare better (see next). A class capable of reaching other (sub)objects of statically unknown, erased type, related to the object at the blob address, without first recovering that complete object, may expose the "children" as separate erasure handles. Such technique is beyond the scope of this paper, but see *Further work* (§7) for some elaboration.

Users occasionally want to provide for their own safety, when it can be independently assured and `recover` adds excessive cost or requires the wrong type. In addition to `recover`, separate access to the class' built-in `void *` value and type checker function enable this flexibility.

Other forms of type erasure are handled by cast operators such as `dynamic_cast` or specific functions like `use_facet`, and may be wrapped in adaptors like `dynamic_pointer_cast`. These vary in meaning, definedness, and existence depending on the semantics of the arguments. This proposal does not attempt to unify various preexisting facilities, but only the sizable and growing cluster of classes matching the model.

3. Proposal

This proposal comprises a common interface provided by type-erasure classes, and a common library facility for accessing type-safe erasures.

3.1. ErasureClass concept

An ErasureClass is a type conforming to the requirements in the following table. Given an ErasureClass type `E` (which may be a cv-qualified type) with objects `e` and `v`, where `v` represents a type-erased value of type `T`; and given `U`, a type not presently erased by `v`:

Expression	**Type**	**Semantics**
`v.complete_object_address()`	*`cv`* `void *`	Requires: `T` is an object type. Points to the type-erased object.
`v.referent_address()`	*`cv`* `void *`	Requires: `T` is a reference type. Points to the type-erased referent.
`e.verify_type<void>()`	`bool`	Equal to `true` if `e` does not have any type-erased value, else `false`.
`v.verify_type<T>()`	`bool`	Equal to `true`.
`v.verify_type<U>()`	`bool`	Requires: `E` is capable of erasing type `U`. Equal to `false`.

None of these expressions may throw an exception.

`complete_object_address()` shall point to an object owned by the ErasureClass handle object `*this`, which is not a subobject of, or owned by, any other object exposed by the handle class interface. In practical terms, it is the object uniquely managed by the handle.

`referent_address()` shall not point to an object that is `const` if the handle is a `const` object and the erased type is a reference to non-const-qualified type. It shall point to an object that is owned by the handle if the erased type is an rvalue reference type.

If one of the `*_address` functions is disallowed for all values of a class, it need not be declared.

The cv-qualification of a handle may affect the value of `verify_type<T>()`. This allows reference erasures a degree of `const` propagation, requiring the user to specify correct qualifiers. For a class type `volatile E` to conform to ErasureClass, `E` must provide `volatile`-qualified overloads. Cv-qualification shall not affect the value of `*_address()`.

Users should seldom want to use the ErasureClass interface directly. It is mainly for the benefit of `recover`. Directly calling `referent_address` or `complete_object_address` is akin to converting a handle to type `void const *`. They need not implement const propagation.

3.2. Training wheels

Some simple classes that allocate padding before the erased object cannot implement an `*_address` accessor per ErasureClass, but are able when given the erased type. (This is a fixable problem in general. See *Compatibility*, §4.1.) If an `*_address()` call is unimplemented for a class, then `recover<T>` will try the same call with an explicit template argument, `*_address<T>()`. Taking this `T` is considered as lower quality of implementation.

3.3. `recover`

`recover` verifies that a given type is one currently being stored in its argument. It retrieves the object address and returns a reference to the result with the given type, propagating const qualification and value category from the function argument to the template argument. If the verification fails, it throws a `bad_recovery_access` exception.

If the function argument is an ErasureClass or it conforms to the "training wheels" interface, its members are used to verify the type and obtain the address. If the argument is a pointer to polymorphic class type, the `typeid` operator and `dynamic_cast<void*>` are used to get a handle on the most-derived object.

The return value category is determined by reference collapsing, like accessing a reference element of a `std::tuple`. The result is an xvalue if both the explicit template type parameter and the function parameter are not lvalue references. Const qualification does not propagate given a reference type parameter; the cv-qualification within the reference type is used verbatim.

Note that cv-qualifications are part of the erased type, so `recover<foo const>(e)` may fail if the erased type of `e` is `foo`, even if `e` is a genuinely constant object and the `foo` is allocated within its read-only bytes. The cast follows semantics defined by the class, and const propagation ensures safety in all cases. Conversely a non-const ErasureClass may recover a const-qualified type, provided it grants permission to do so.

`try_recover` avoids the potential exception by returning a nullable reference. Unfortunately, the complete specification of its return type is beyond this proposal. It behaves like a pointer in implementing an `operator*` which yields the referent, and a conversion to `bool` which reveals whether dereferencing is well-defined. The difference from a simple pointer is that the dereference operator propagates value category. For example:

```
if ( auto r = std::try_recover< std::string >
                ( std::forward< E >( e ) ) ) {        // If e contains a string…
    my_list.push_back( * std::move( r ) );    //… forward it to the end of the list.
}
```

Support for `*std::move(r)` instead of only `std::move(*r)` enables forwarding through the recovered nullable reference.

```
template< typename T, typename ErasureClass >
unspecified try_recover( ErasureClass cv ref e ) noexcept;   // ref is & or &&.
```

The proposed function template signatures are informative: the second template type parameter may include cv- and reference qualifiers, such that e.g. `static_cast<int const& (*)(any const&)>(&std::try_recover< int, any >)` may not be a valid expression.

Requires: `ErasureClass` is capable of erasing `T`. (Thus, `T` shall not be `void`. A class may not assign semantics to "erased values of type `void`," and the `try_recover` implementation should diagnose an attempt to use `recover< void >( & e )` as an accessor.)

Returns: A value of unspecified type defining a contextual conversion to `bool` and a dereference operator. If `e->verify_type< T >()` is `false`, the Boolean conversion yields `false`. Otherwise, the Boolean conversion yields `true` and the dereference operator yields `static_cast< T cv ref >( * p )`, where *p* is obtained as follows:
• If `T` is an object type, let *p* be `(T cv *) e->complete_object_address()`.
• Else, let *p* be `(remove_reference_t< T > *) e->referent_address()`.
Note that these pointer cast expressions may implement `const_cast`, and *cv* is ignored in `T cv ref` if `T` is a reference type.

```
template< typename T, typename Poly >
unspecified try_recover( Poly cv * e ) noexcept;
```

Requires: `T` is of the form `U cv &` where `U` is an object type and *cv* is the same as in the signature. `Poly` is a polymorphic class type. `is_base_of<Poly, U>::value` is `true`. (Note that this function disregards base class access qualification and tolerates base class ambiguity.)

Returns: A value of unspecified type defining a contextual conversion to `bool` and a dereference operator. Unlike the foregoing overload, this type may be simply `T cv *`. If `typeid(U) != typeid(*e)`, the Boolean conversion yields `false`. Otherwise, the Boolean conversion yields `true` and the dereference operator yields `* static_cast<T cv *>(dynamic_cast<void cv *>( e ))`.

```
template< typename T, typename E >
auto && recover( E && e );
```

Requires: `try_recover< T >( forward< E >( e ) )` is well formed.

Effects: Let *p* be `try_recover< T >( forward< E >( e ) )`. If `!p`, throw an object of class `bad_recovery_access`. Otherwise, return `*p` (which may be an xvalue).

```
class bad_recovery_access : public exception {
public:
    virtual const char* what() const noexcept;
};
```

The class `bad_recovery_access` represents an exception, in the usual fashion. It is suitable for use by classes such as `optional`, `variant`, `function`, etc., directly or as a base class of exceptions defined for failures of their specific functionalities.

3.4. Applicability

`std::function`, and also `any` and `optional` from the Fundamentals TS v.2 will conform to the ErasureClass interface. Before its adoption, `variant` should also be adjusted. Third-party libraries like Boost.TypeErasure are encouraged to adopt the interface. The concept is designed to be easy to retrofit to user libraries, but more than this, to serve as a basis for new classes.

Classes such as `function` and `any` that identify erased types by `typeid` shall not allow it to strip reference-and cv-qualification from the explicit template argument to `verify_type`. Since they cannot manage const objects or references, their implementations of `verify_type` should diagnose such type arguments by `static_assert`.

Pointers observing polymorphic objects such as `shared_ptr<`*`cv`*` T>` and `observer_ptr<`*`cv`*` T>` support ErasureClass by implementing `verify_type<`*`cv`*` U &>` as `typeid(ref) == typeid(U)`, and `referent_address` by `dynamic_cast<void *>(&ref)`. `verify_type` should statically assert `is_base_of<T, U>`. Neither function is implemented by their specializations over non-polymorphic classes.

On the other hand, `unique_ptr<T>` type-erases both `U` and `U&&`, since it typically knows its own complete object, and since it is a non-const-propagating handle, respectively. For polymorphic `T`, its ErasureClass interface works as for `observer_ptr`, but with rvalue references instead of lvalue. For non-polymorphic `T`, `unique_ptr<T>::verify_type<U>` determines an exact match as by using `std::is_same`.

4. Rationale

A new interface is proposed because none of the existing class interfaces can be suitably extended. The maximum extent of common functionality is brought into the generic `recover`. ErasureClass is designed to provide useful primitives at trivial implementation cost.

ErasureClass accessors

“Internal” `void *` values are exposed because they are a primitive, common to erasure classes based on memory blobs, and in some cases they are useful to programmers. Not only do `complete_object_address` and `referent_address` provide the fastest way to recover the contents of an erasure when the type is certainly known beforehand, they allow a degree of auxiliary polymorphism when the erased object or referent of *unknown* type is known to share its address with an object of known type, for example the initial member of a standard-layout class, or a singly-inherited base class subobject under common ABI guarantees.

For example, several functor classes might contain a bound argument value of a given type. If they use a common layout, the argument may be found given only a `std::function` wrapper.

Separate “object” and “referent” functions are provided to reflect whether or not a handle owns its blob, which is significant to deciding the safety of object representation access in the absence of confirmed ownership and const-qualification. Classes which erase arbitrary reference and object values may need to implement them differently — see *Examples*, §5.

Const propagation is not specified for members, to simplify the implementation of conforming classes, and because it is not helpful to the implementation of `recover`. Note that a non-`const` reference value may be retrieved from a `const` erasure object.

Type verification

No `type_info` accessor is provided because it would be both expensive and restrictive. It would require `variant` to build a lookup table. It would muddle the semantics of references and cv-qualification, which are stripped by `typeid`. It would preclude multiple simultaneous values, for example a class capable of erasing a standard-layout derived object and recovering a base subobject [1]. On the other hand, `variant::verify_type` may be implemented without RTTI.

Reliably determining the success of an access is what `verify_type` does, no more, no less. Compile-time diagnosis of impossible type checks is encouraged to avoid unexpected negative results at runtime. It is an error to ask an object-type eraser like `any` to recover a reference.

Disengagement is checked by `verify_type<void>()`, as all the surveyed classes have varying degrees of such a service. Absent other type-erased functionality, knowing that an erasure is not empty is unhelpful without knowing the erased type. This is why no more specialized interface is required (such as `operator bool`). Nevertheless, the caller may well have narrowed the possibilities down to one particular type or the empty state. The intent is that `verify_type<void>()` should be faster than the general `verify_type<T>()`, such as by avoiding `type_info` comparison in `function` or `any`. On the other hand, it is also valid to implement it uniformly using RTTI, and those classes do already use the value `typeid(void)`.

Member interface vs. non-members

The class interface comprises only member functions, to simplify implementation and to reduce namespace pollution. If ADL were used, the interface names would effectively be reserved in user namespaces containing ErasureClasses. Also, ADL can be fragile if a user defines an indiscriminate template, expecting it to be found only when the immediate argument is their own class, and then an unexpected association occurs e.g. via class template arguments. A class member named `recover` would hide the unqualified name. It is better to avoid ADL.

Accessors in `namespace std` for the ErasureClass interface are not proposed because they would add bloat but little value. However, they may be useful for raw pointer types, so the door is open for a follow-on proposal.

The primary function, `recover`

ADL is also not used for `std::recover`, so it may (and should) be used as a qualified name. The complicated process of defining it separately for individual classes is the problem that inspired this proposal. If necessary in the future, platforms (or the standard library) may define compatibility with additional classes or generic interfaces. For example, `variant` in N4542 supports retrieval by numeric index to `std::get`, an interface that could map directly to `recover`.

[1] Reliance on common ABI guarantees on single inheritance may yield more practical examples.

`recover` approximates member access semantics in the same way as `std::get`. Typecasts should be orthogonal to each other, so it is inappropriate to combine erasure recovery with user-specified adjustment to cv-qualification and value category, as does `any_cast`. The name `recover_cast` was considered and rejected.

The return type of `try_recover`

A nullable reference with rvalue propagation support is not very exotic, but no such thing is yet standardized. N4542's `variant` is capable of implementing this, but an `optional<T&&>` class would be more appropriate. However, no `optional` for reference types has yet been accepted. For the sake of efficiency and ABI forward-compatibility, implementations are encouraged to use a type with the same layout as a single `T*` pointer.

Although the raw polymorphic pointer overload may return `T` *`cv`* `*` in this proposal, uniformity with the other overload (which handles `observer_ptr` using equivalent constraints and semantics) may be preferable.

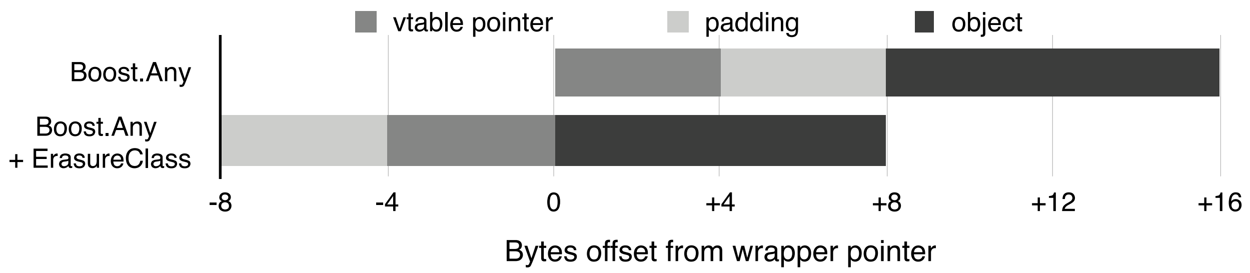
4.1. Compatibility

ErasureClass is the fundamental basis for `recover`. It accommodates a wide variety of classes, although its constraints are narrow. Generally speaking, an owning handle class should be able to find the storage of its object, without being given its type, because it should be able to destroy that object. In most cases, the storage address is fixed by a `union` or a `void *`, or a virtual (or similarly indirect) function already exists to find it.

The only less-capable classes known to this author are ones that place the type-erased object in a `struct` but not at its beginning: for example, after a discriminating value for an enclosing `union`. Or, in a polymorphic erasure object with no accessor function, and no small-object-optimization space in the wrapper for a shortcut pointer. These can be worked around, at the expense of changing the ABI but not losing any efficiency (in effect, only padding is shuffled):

In the first case, move the discriminating value out of the union.

In the second case (which happens to describe Boost.Any), move the erased object out of the polymorphic control class. Call `operator new` and use placement new so the target object immediately follows the emptied control object, which invariably has the size of one vtable pointer. Point the wrapper at the erased object, and when the control object is desired, find it by a negative array index. On destruction, call `operator delete` on the address of a more-negative array index, to account for the padding at the start of the allocation block.



Note that `experimental::any` in libstdc++ and libc++ have no such issue. Their architecture is completely different.

Considering that ordinary users may encounter such problems, the “training wheels” accessor function templates are specified. However, they are not part of the ErasureClass concept. Standard library classes are still required to provide the object or referent address without being given its type.

5. Examples

5.1. Usage

Here is a basic `optional`-like class, offering less proxy-like behavior than `experimental::optional` — the wrapper values are consistently held distinct from the wrapped. Assignment is implemented as disengagement and reengagement; in this way references are supported. Its reference support is sufficient for use as a `try_recover` return type (assuming that `recover` is not implemented in terms of `try_recover`).

```
template< typename ref >
class refwrap {
    void const * pointer;
protected:
    typedef refwrap wrapper;

    refwrap() : pointer( nullptr ) {}
    explicit refwrap( ref r ) : pointer( & r ) {}
    refwrap( refwrap && ) = delete;
public:
    void const * referent_address() const           { return pointer; }
    explicit operator bool () const                 { return pointer; }
};

template< typename obj >
struct objwrap {
    std::aligned_storage_t< sizeof (obj), alignof (obj) > storage;
    bool engaged;
protected:
    typedef objwrap wrapper;

    objwrap() : engaged( false ) {}
    explicit objwrap( obj o ) : engaged( true )
        { new (& storage) obj( std::move( o ) ); }
    objwrap( objwrap && ) = delete;
    ~ objwrap()
        { if (engaged) ((obj &) storage).obj:: ~ obj(); }
public:
    void const * complete_object_address() const  { return & storage; }
    explicit operator bool () const                 { return engaged; }
};

template< typename value_t >
```

```
class opt
    : public std::conditional_t< std::is_reference< value_t >::value,
                              refwrap< value_t >, objwrap< value_t > >
{
    using typename opt::wrapper::wrapper;
    void init( value_t v ) {
        new ((wrapper *) this) wrapper( std::forward<value_t>( v ) );
    }
public:
    typedef value_t value_type; // May be a reference!

    template< typename want >
    bool verify_type() const {
        static_assert ( std::is_void< want >::value
            || std::is_same< want, value_t >::value, "Invalid type query." );
        return bool{ *this } == std::is_same< want, value_t >::value;
    }

    using wrapper::wrapper; // Inherit conversion from value_t.
    opt() = default;
    opt( opt const & o )
        { if ( o ) init( std::recover< value_t >( o ) ); }
    opt( opt && o ) {
        if ( o ) init( std::recover< value_t >( std::move( o ) ) );
        o = {};
    }
    opt & operator = ( opt o ) {
        this-> ~ wrapper();
        new ((wrapper *) this) wrapper;
        if ( o ) init( std::recover< value_t >( std::move( o ) ) );
        return * this;
    }
};

template< typename opt_r >          // Non-member dereference. This is missing SFINAE.
auto && operator * ( opt_r && r ) {
    return std::recover< typename std::decay_t< opt_r >::value_type >
        ( std::forward< opt_r >( r ) );
}
```

5.2. Implementation

Here is the core of an implementation of `recover`, including reference and `const` propagation. “Training wheels” interface support, `try_recover`, `volatile` propagation, and class `bad_recovery_access` are not shown.

```
template< typename ErasureClass >
void const * recover_address( ErasureClass & e, std::false_type )
        { return e.complete_object_address(); }
```

```
template< typename ErasureClass >
void const * recover_address( ErasureClass & e, std::true_type )
        { return e.referent_address(); }

template< typename T, typename ErasureClass >
constexpr auto && recover( ErasureClass && e ) {
    using prop_const = std::conditional_t<
        std::is_const< std::remove_reference_t<ErasureClass> >::value,
            T const, T >;
    using prop_cref = std::conditional_t<
        std::is_lvalue_reference< ErasureClass >::value,
            prop_const &, prop_const >;
    using object = std::remove_reference_t< prop_const >;

    if ( ! e.template verify_type< T >() )
        throw bad_recovery_access{};
    return std::forward< prop_cref >
        ( * (object *) recover_address( e, std::is_reference<T>{} ) );
}

template< typename T, typename Polymorphic >
constexpr auto & recover( Polymorphic * e ) noexcept {
    using object = std::remove_reference_t< T >;
    static_assert ( std::is_polymorphic< Polymorphic >::value,
        "This function does not support non-polymorphic pointers." );
    static_assert ( std::is_lvalue_reference< T >::value &&
        std::is_const< object >::value
        == std::is_const< Polymorphic >::value,
        "Only an lvalue reference may be recovered from a pointer, and its const must match." );
    static_assert ( std::is_base_of< Polymorphic, object >::value,
        "The requested type is not derived from the given pointer." );

    if ( ! e || typeid (*e) != typeid (T) )
        throw bad_recovery_access{};
    return * (object *) dynamic_cast< void const * >( e );
}
```

6. Adoption

Redundancy between `recover` and class-specific interfaces should be minimized. At present, the only standard erasure class is `std::function`; its accessor `function::target<T>()` is entirely redundant. `function::target_type()` adds some value, but it does not come for free: Unlike it, `verify_type<T>()` can be implemented without `type_info` objects, which potentially have non-negligible equality comparison complexity and executable file overhead. This optimization (and `recover` itself) have been prototyped in the *cxx_function* library.

`std::experimental::any` is architecturally similar to `function`. Its accessor `any_cast` is in widespread use deriving from Boost.Any. Unfortunately, `any_cast` is unsafe with respect to value category. It happily retrieves an lvalue from an rvalue, making its deprecation is somewhat urgent. Although `recover` is not a drop-in replacement, `static_cast<T>(std::recover< std::remove_reference_t<T>>(value))` covers all the safe cases, and such a function could be kept permanently under the name `experimental::any_cast`.

`recover` should be a drop-in replacement for N4542's `get<T>(variant)`.

`bad_recovery_access` should be used as a replacement or a common base class of `bad_any_cast`, `bad_optional_access`, `bad_variant_access`, `bad_function_call`, etc. Note that [res.on.exception.handling] §17.6.5.12/2 allows `recover` to throw subclasses of `bad_recovery_access` when operating on standard types.

Since `recover` does not use ADL, it may be implemented in `namespace experimental`.

7. Further work

7.1. Conceptification

The same approach of building on a minimal concept could possibly be applied to smart pointers, so `dynamic_pointer_cast` etc. may work with types other than `shared_ptr`. This may anticipate formalized Concepts; the library does not need to wait for the language to catch up.

7.2. Constant expressions

Whereas `optional` and `variant` work as literal classes, `recover` cannot be `constexpr` as it relies on casting from `void*` to object type. Compatibility may be enabled by further relaxation of constant expression restrictions, specifically the case of casting from an object pointer type `T*` to `void*` and then back to `T*`.

7.3. Shared control blocks

In addition to supporting `std::shared_ptr` as an observer, it may be useful to expose its internal owning pointer. `recover` could expose the argument to the deleter in the same way that `get_deleter` exposes the deleter itself. It would recover the complete shared object more reliably than `dynamic_pointer_cast<void>(p)` or `recover<T>(p)`, which only work on polymorphic base subobjects. This deserves a separate proposal. First, granting write access to the shared internal pointer seems risky. Second, it may require additional overhead, as current libraries do not expose the type of the pointer. Third, the `get_deleter` erasure interface would lag behind this new update. It may be preferable to address the pointer and the deleter together, with a thorough proposal and perhaps a new accessor.

Example (not currently proposed):

```
struct s { int i; };
std::shared_ptr< int > si;
{
    std::shared_ptr< s > ss( new s{ 5 }, make_my_deleter() );
    si = std::shared_ptr< int >( & ss->i, ss );
} // Access to the complete s object is now irrecoverably lost (without an extension).
```

```
auto origin = si.get_owner();      // Further work: shared control block representation.
s & recovered = std::recover< s >( origin->object );         // Got it back!
my_deleter && d                                       // Handle deleter uniformly as well.
    = std::recover< my_deleter >( std::move( origin->deleter ) );
```

7.4. Calling without explicit template argument

Defaulting the initial template parameter `T` to `ErasureClass::reference_type` or `ErasureClass::value_type` could be useful to `optional` or Ranges, enabling syntax like `recover(rng)` instead of `*rng`. (Default parameters are ignored unless they are needed.) This would need some investigation, especially regarding rvalue range objects.

8. Conclusion and kudos

Type erasure is a common technique deserving a common interface. The divergence of standard-track facilities, and their many remaining bugs in value category handling, prove that this wheel is not one to be reinvented. Standardized `recover` is a win for usability, extensibility, implementability, and performance.

Kudos to Ville Voutilainen for helpful review and suggesting the connections to smart- and polymorphic pointers, and to Vicente Botet for helping to refactor and separate `try_recover` from `recover`.

Thanks to Andrey Semashev for patient discussion, and raising important concerns.