Boost.Optional

Fernando Luis Cacciola Carballal

Table of Contents

Motivation

Consider these functions which should return a value but which might not have a value to return:

- (A) `double sqrt(double n );`
- (B) `char get_async_input();`
- (C) `point polygon::get_any_point_effectively_inside();`

There are different approaches to the issue of not having a value to return.

A typical approach is to consider the existence of a valid return value as a postcondition, so that if the function cannot compute the value to return, it has either undefined behavior (and can use assert in a debug build) or uses a runtime check and throws an exception if the postcondition is violated. This is a reasonable choice for example, for function (A), because the lack of a proper return value is directly related to an invalid parameter (out of domain argument), so it is appropriate to require the callee to supply only parameters in a valid domain for execution to continue normally.

However, function (B), because of its asynchronous nature, does not fail just because it can't find a value to return; so it is incorrect to consider such a situation an error and assert or throw an exception. This function must return, and somehow, must tell the callee that it is not returning a meaningful value.

A similar situation occurs with function (C): it is conceptually an error to ask a *null-area* polygon to return a point inside itself, but in many applications, it is just impractical for performance reasons to treat this as an error (because detecting that the polygon has no area might be too expensive to be required to be tested previously), and either an arbitrary point (typically at infinity) is returned, or some efficient way to tell the callee that there is no such point is used.

There are various mechanisms to let functions communicate that the returned value is not valid. One such mechanism, which is quite common since it has zero or negligible overhead, is to use a special value which is reserved to communicate this. Classical examples of such special values are `EOF`, `string::npos`, points at infinity, etc...

When those values exist, i.e. the return type can hold all meaningful values *plus* the *signal* value, this mechanism is quite appropriate and well known. Unfortunately, there are cases when such values do not exist. In these cases, the usual alternative is either to use a wider type, such as `int` in place of `char`; or a compound type, such as `std::pair<point,bool>`.

Returning a `std::pair<T,bool>`, thus attaching a boolean flag to the result which indicates if the result is meaningful, has the advantage that can be turned into a consistent idiom since the first element of the pair can be whatever the function would conceptually return. For example, the last two functions could have the following interface:

```
std::pair<char,bool> get_async_input();
std::pair<point,bool> polygon::get_any_point_effectively_inside();
```

These functions use a consistent interface for dealing with possibly inexistent results:

```
std::pair<point,bool> p = poly.get_any_point_effectively_inside();
if ( p.second )
    flood_fill(p.first);
```

However, not only is this quite a burden syntactically, it is also error prone since the user can easily use the function result (first element of the pair) without ever checking if it has a valid value.

Clearly, we need a better idiom.

Development

The models

In C++, we can *declare* an object (a variable) of type `T`, and we can give this variable an *initial value* (through an *initializer*. (c.f. 8.5)). When a declaration includes a non-empty initializer (an initial value is given), it is said that the object has been initialized. If the declaration uses an empty initializer (no initial value is given), and neither default nor value initialization applies, it is said that the object is **uninitialized**. Its actual value exist but has an *indeterminate initial value* (c.f. 8.5.9). `optional<T>` intends to formalize the notion of initialization (or lack of it) allowing a program to test whether an object has been initialized and stating that access to the value of an uninitialized object is undefined behavior. That is, when a variable is declared as `optional<T>` and no initial value is given, the variable is *formally* uninitialized. A formally uninitialized optional object has conceptually no value at all and this situation can be tested at runtime. It is formally *undefined behavior* to try to access the value of an uninitialized optional. An uninitialized optional can be assigned a value, in which case its initialization state changes to initialized. Furthermore, given the formal treatment of initialization states in optional objects, it is even possible to reset an optional to *uninitialized*.

In C++ there is no formal notion of uninitialized objects, which means that objects always have an initial value even if indeterminate. As discussed on the previous section, this has a drawback because you need additional information to tell if an object has been effectively initialized. One of the typical ways in which this has been historically dealt with is via a special value: `EOF`, `npos`, -1, etc... This is equivalent to adding the special value to the set of possible values of a given type. This super set of `T` plus some *nil_t*—were `nil_t` is some stateless POD-can be modeled in modern languages as a **discriminated union** of T and nil_t. Discriminated unions are often called *variants*. A variant has a *current type*, which in our case is either `T` or `nil_t`. Using the Boost.Variant library, this model can be implemented in terms of `boost::variant<T,nil_t>`. There is precedent for a discriminated union as a model for an optional value: the Haskell **Maybe** built-in type constructor. Thus, a discriminated union `T+nil_t` serves as a conceptual foundation.

A `variant<T,nil_t>` follows naturally from the traditional idiom of extending the range of possible values adding an additional sentinel value with the special meaning of *Nothing*. However, this additional *Nothing* value is largely irrelevant for our purpose since our goal is to formalize the notion of uninitialized objects and, while a special extended value can be used to convey that meaning, it is not strictly necessary in order to do so.

The observation made in the last paragraph about the irrelevant nature of the additional `nil_t` with respect to purpose of `optional<T>` suggests an alternative model: a *container* that either has a value of `T` or nothing.

As of this writing I don't know of any precedence for a variable-size fixed-capacity (of 1) stack-based container model for optional values, yet I believe this is the consequence of the lack of practical implementations of such a container rather than an inherent shortcoming of the container model.

In any event, both the discriminated-union or the single-element container models serve as a conceptual ground for a class representing optional—i.e. possibly uninitialized—objects. For instance, these models show the *exact* semantics required for a wrapper of optional values:

Discriminated-union:

- **deep-copy** semantics: copies of the variant implies copies of the value.
- **deep-relational** semantics: comparisons between variants matches both current types and values
- If the variant's current type is `T`, it is modeling an *initialized* optional.
- If the variant's current type is not `T`, it is modeling an *uninitialized* optional.
- Testing if the variant's current type is `T` models testing if the optional is initialized
- Trying to extract a `T` from a variant when its current type is not `T`, models the undefined behavior of trying to access the value of an uninitialized optional

Single-element container:

- **deep-copy** semantics: copies of the container implies copies of the value.
- **deep-relational** semantics: comparisons between containers compare container size and if match, contained value
- If the container is not empty (contains an object of type `T`), it is modeling an *initialized* optional.
- If the container is empty, it is modeling an *uninitialized* optional.
- Testing if the container is empty models testing if the optional is initialized
- Trying to extract a `T` from an empty container models the undefined behavior of trying to access the value of an uninitialized optional

The semantics

Objects of type `optional<T>` are intended to be used in places where objects of type `T` would but which might be uninitialized. Hence, `optional<T>`'s purpose is to formalize the additional possibly uninitialized state. From the perspective of this role, `optional<T>` can have the same operational semantics of `T` plus the additional semantics corresponding to this special state. As such, `optional<T>` could be thought of as a *supertype* of `T`. Of course, we can't do that in C++, so we need to compose the desired semantics using a different mechanism. Doing it the other way around, that is, making `optional<T>` a *subtype* of `T` is not only conceptually wrong but also impractical: it is not allowed to derive from a non-class type, such as a built-in type.

We can draw from the purpose of `optional<T>` the required basic semantics:

- **Default Construction:** To introduce a formally uninitialized wrapped object.
- **Direct Value Construction via copy:** To introduce a formally initialized wrapped object whose value is obtained as a copy of some object.

- **Deep Copy Construction:** To obtain a new yet equivalent wrapped object.
- **Direct Value Assignment (upon initialized):** To assign a value to the wrapped object.
- **Direct Value Assignment (upon uninitialized):** To initialize the wrapped object with a value obtained as a copy of some object.
- **Assignment (upon initialized):** To assign to the wrapped object the value of another wrapped object.
- **Assignment (upon uninitialized):** To initialize the wrapped object with value of another wrapped object.
- **Deep Relational Operations (when supported by the type T):** To compare wrapped object values taking into account the presence of uninitialized states.
- **Value access:** To unwrap the wrapped object.
- **Initialization state query:** To determine if the object is formally initialized or not.
- **Swap:** To exchange wrapped objects. (with whatever exception safety guarantees are provided by `T`'s swap).
- **De-initialization:** To release the wrapped object (if any) and leave the wrapper in the uninitialized state.

Additional operations are useful, such as converting constructors and converting assignments, in-place construction and assignment, and safe value access via a pointer to the wrapped object or null.

The Interface

Since the purpose of optional is to allow us to use objects with a formal uninitialized additional state, the interface could try to follow the interface of the underlying `T` type as much as possible. In order to choose the proper degree of adoption of the native `T` interface, the following must be noted: Even if all the operations supported by an instance of type `T` are defined for the entire range of values for such a type, an `optional<T>` extends such a set of values with a new value for which most (otherwise valid) operations are not defined in terms of `T`.

Furthermore, since `optional<T>` itself is merely a `T` wrapper (modeling a `T` supertype), any attempt to define such operations upon uninitialized optionals will be totally artificial w.r.t. `T`.

This library chooses an interface which follows from `T`'s interface only for those operations which are well defined (w.r.t the type `T`) even if any of the operands are uninitialized. These operations include: construction, copy-construction, assignment, swap and relational operations.

For the value access operations, which are undefined (w.r.t the type `T`) when the operand is uninitialized, a different interface is chosen (which will be explained next).

Also, the presence of the possibly uninitialized state requires additional operations not provided by `T` itself which are supported by a special interface.

Lexically-hinted Value Access in the presence of possibly untitialized optional objects: The operators * and ->

A relevant feature of a pointer is that it can have a **null pointer value**. This is a *special* value which is used to indicate that the pointer is not referring to any object at all. In other words, null pointer values convey the notion of inexistent objects.

This meaning of the null pointer value allowed pointers to became a *de facto* standard for handling optional objects because all you have to do to refer to a value which you don't really have is to use a null pointer value of the appropriate type. Pointers have been used for decades—from the days of C APIs to modern C++ libraries—to *refer* to optional (that is, possibly inexistent) objects; particularly as optional arguments to a function, but also quite often as optional data members.

The possible presence of a null pointer value makes the operations that access the pointee's value possibly undefined, therefore, expressions which use dereference and access operators, such as: `( *p = 2 )` and `( p->foo() )`, implicitly convey the notion of optionality, and this information is tied to the *syntax* of the expressions. That is, the presence of operators `*` and `->` tell by themselves —without any additional context— that the expression will be undefined unless the implied pointee actually exist.

Such a *de facto* idiom for referring to optional objects can be formalized in the form of a concept: the OptionalPointee concept. This concept captures the syntactic usage of operators `*`, `->` and conversion to `bool` to convey the notion of optionality.

However, pointers are good to refer to optional objects, but not particularly good to handle the optional objects in all other respects, such as initializing or moving/copying them. The problem resides in the shallow-copy of pointer semantics: if you need to effectively move or copy the object, pointers alone are not enough. The problem is that copies of pointers do not imply copies of pointees. For example, as was discussed in the motivation, pointers alone cannot be used to return optional objects from a function because the object must move outside from the function and into the caller's context.

A solution to the shallow-copy problem that is often used is to resort to dynamic allocation and use a smart pointer to automatically handle the details of this. For example, if a function is to optionally return an object `X`, it can use `shared_ptr<X>` as the return value. However, this requires dynamic allocation of `X`. If `X` is a built-in or small POD, this technique is very poor in terms of required resources. Optional objects are essentially values so it is very convenient to be able to use automatic storage and deep-copy semantics to manipulate optional values just as we do with ordinary values. Pointers do not have this semantics, so are inappropriate for the initialization and transport of optional values, yet are quite convenient for handling the access to the possible undefined value because of the idiomatic aid present in the OptionalPointee concept incarnated by pointers.

Optional<T> as a model of OptionalPointee

For value access operations `optional<>` uses operators `*` and `->` to lexically warn about the possibly uninitialized state appealing to the familiar pointer semantics w.r.t. to null pointers.

Warning

However, it is particularly important to note that `optional<>` objects are not pointers. `optional<>` is not, and does not model, a pointer.

For instance, `optional<>` does not have shallow-copy so does not alias: two different optionals never refer to the *same* value unless `T` itself is a reference (but may have *equivalent* values). The difference between an `optional<T>` and a pointer must be kept in mind, particularly because the semantics of relational operators are different: since `optional<T>` is a value-wrapper, relational operators are deep: they compare optional values; but relational operators for pointers are shallow: they do not compare pointee values. As a result, you might be able to replace `optional<T>` by `T*` on some situations but not always. Specifically, on generic code written for both, you cannot use relational operators directly, and must use the template functions `equal_pointees()` and `less_pointees()` instead.

Synopsis

```
namespace boost {

template<class T>
class optional
{
    public :

    // (If T is of reference type, the parameters and results by reference are by value)

    optional () ;

    optional ( none_t ) ;

    optional ( T const& v ) ;

    // [new in 1.34]
    optional ( bool condition, T const& v ) ;

    optional ( optional const& rhs ) ;

    template<class U> explicit optional ( optional<U> const& rhs ) ;

    template<class InPlaceFactory> explicit optional ( InPlaceFactory const& f ) ;

    template<class TypedInPlaceFactory> explicit optional ( TypedInPlaceFactory const& f ) ;

    optional& operator = ( none_t ) ;

    optional& operator = ( T const& v ) ;

    optional& operator = ( optional const& rhs ) ;

    template<class U> optional& operator = ( optional<U> const& rhs ) ;

    template<class InPlaceFactory> optional& operator = ( InPlaceFactory const& f ) ;

    template<class TypedInPlaceFactory> optional& operator = ( TypedInPlaceFactory const& f ) ;

    T const& get() const ;
    T&       get() ;

    // [new in 1.34]
    T const& get_value_or( T const& default ) const ;

    T const* operator ->() const ;
    T*       operator ->() ;

    T const& operator *() const ;
    T&       operator *() ;

    T const* get_ptr() const ;
    T*       get_ptr() ;

    operator unspecified-bool-type() const ;

    bool operator!() const ;

    // deprecated methods
```

```
    // (deprecated)
    void reset() ;

    // (deprecated)
    void reset ( T const& ) ;

    // (deprecated)
    bool is_initialized() const ;

};

template<class T> inline bool operator == ( optional<T> const& x, optional<T> const& y ) ;

template<class T> inline bool operator != ( optional<T> const& x, optional<T> const& y ) ;

template<class T> inline bool operator <  ( optional<T> const& x, optional<T> const& y ) ;

template<class T> inline bool operator >  ( optional<T> const& x, optional<T> const& y ) ;

template<class T> inline bool operator <= ( optional<T> const& x, optional<T> const& y ) ;

template<class T> inline bool operator >= ( optional<T> const& x, optional<T> const& y ) ;

// [new in 1.34]
template<class T> inline optional<T> make_optional ( T const& v ) ;

// [new in 1.34]
template<class T> inline optional<T> make_optional ( bool condition, T const& v ) ;

// [new in 1.34]
template<class T> inline T const& get_optional_value_or ( optional<T> const& opt, T const& de↵
fault ) ;

template<class T> inline T const& get ( optional<T> const& opt ) ;

template<class T> inline T& get ( optional<T> & opt ) ;

template<class T> inline T const* get ( optional<T> const* opt ) ;

template<class T> inline T* get ( optional<T>* opt ) ;

template<class T> inline T const* get_pointer ( optional<T> const& opt ) ;

template<class T> inline T* get_pointer ( optional<T> & opt ) ;

template<class T> inline void swap( optional<T>& x, optional<T>& y ) ;

} // namespace boost
```

Detailed Semantics

Because `T` might be of reference type, in the sequel, those entries whose semantic depends on `T` being of reference type or not will be distinguished using the following convention:

- If the entry reads: `optional<T` *(not a ref)*`>`, the description corresponds only to the case where `T` is not of reference type.
- If the entry reads: `optional<T&>`, the description corresponds only to the case where `T` is of reference type.
- If the entry reads: `optional<T>`, the description is the same for both cases.

Note

The following section contains various `assert()` which are used only to show the postconditions as sample code. It is not implied that the type `T` must support each particular expression but that if the expression is supported, the implied condition holds.

optional class member functions

`optional<T>::optional();`

- **Effect:** Default-Constructs an `optional`.
- **Postconditions:** `*this` is uninitialized.
- **Throws:** Nothing.
- Notes: T's default constructor is not called.
- **Example:**

```
optional<T> def ;
assert ( !def ) ;
```

`optional<T>::optional( none_t );`

- **Effect:** Constructs an `optional` uninitialized.
- **Postconditions:** `*this` is uninitialized.
- **Throws:** Nothing.
- **Notes:** `T`'s default constructor is not called. The expression `boost::none` denotes an instance of `boost::none_t` that can be used as the parameter.
- **Example:**

```
#include <boost/none.hpp>
optional<T> n(none) ;
assert ( !n ) ;
```

`optional<T` *(not a ref)*`>::optional( T const& v )`

- **Effect:** Directly-Constructs an `optional`.

- **Postconditions:** `*this` is initialized and its value is a *copy* of `v`.
- **Throws:** Whatever `T::T( T const& )` throws.
- **Notes:** `T::T( T const& )` is called.
- **Exception Safety:** Exceptions can only be thrown during `T::T( T const& );` in that case, this constructor has no effect.
- **Example:**

```
T v;
optional<T> opt(v);
assert ( *opt == v ) ;
```

`optional<T&>::optional( T& ref )`

- **Effect:** Directly-Constructs an `optional`.
- **Postconditions:** `*this` is initialized and its value is an instance of an internal type wrapping the reference `ref`.
- **Throws:** Nothing.
- **Example:**

```
T v;
T& vref = v ;
optional<T&> opt(vref);
assert ( *opt == v ) ;
++ v ; // mutate referee
assert (*opt == v);
```

`optional<T` *(not a ref)*`>::optional( bool condition, T const& v ) ;`

`optional<T&> ::optional( bool condition, T& v ) ;`

- If condition is true, same as:

  `optional<T` *(not a ref)*`>::optional( T const& v )`

  `optional<T&> ::optional( T& v )`

- otherwise, same as:

  `optional<T ['(not a ref)]>::optional()`

  `optional<T&> ::optional()`

`optional<T` *(not a ref)*`>::optional( optional const& rhs );`

- **Effect:** Copy-Constructs an `optional`.
- **Postconditions:** If rhs is initialized, `*this` is initialized and its value is a *copy* of the value of `rhs`; else `*this` is uninitialized.
- **Throws:** Whatever `T::T( T const& )` throws.
- **Notes:** If rhs is initialized, `T::T(T const& )` is called.
- **Exception Safety:** Exceptions can only be thrown during `T::T( T const& );` in that case, this constructor has no effect.
- **Example:**

```
optional<T> uninit ;
assert (!uninit);

optional<T> uinit2 ( uninit ) ;
assert ( uninit2 == uninit );

optional<T> init( T(2) );
assert ( *init == T(2) ) ;

optional<T> init2 ( init ) ;
assert ( init2 == init ) ;
```

`optional<T&>::optional( optional const& rhs );`

- **Effect:** Copy-Constructs an `optional`.
- **Postconditions:** If `rhs` is initialized, `*this` is initialized and its value is another reference to the same object referenced by `*rhs`; else `*this` is uninitialized.
- **Throws:** Nothing.
- **Notes:** If `rhs` is initialized, both `*this` and `*rhs` will reefer to the same object (they alias).
- **Example:**

```
optional<T&> uninit ;
assert (!uninit);

optional<T&> uinit2 ( uninit ) ;
assert ( uninit2 == uninit );

T v = 2 ; T& ref = v ;
optional<T> init(ref);
assert ( *init == v ) ;

optional<T> init2 ( init ) ;
assert ( *init2 == v ) ;

v = 3 ;

assert ( *init  == 3 ) ;
assert ( *init2 == 3 ) ;
```

`template<U> explicit optional<T` *(not a ref)*`>::optional( optional<U> const& rhs );`

- **Effect:** Copy-Constructs an `optional`.
- **Postconditions:** If `rhs` is initialized, `*this` is initialized and its value is a *copy* of the value of rhs converted to type `T`; else `*this` is uninitialized.
- **Throws:** Whatever `T::T( U const& )` throws.
- **Notes:** `T::T( U const& )` is called if `rhs` is initialized, which requires a valid conversion from `U` to `T`.
- **Exception Safety:** Exceptions can only be thrown during `T::T( U const& );` in that case, this constructor has no effect.
- **Example:**

```
optional<double> x(123.4);
assert ( *x == 123.4 ) ;

optional<int> y(x) ;
assert( *y == 123 ) ;
```

`template<InPlaceFactory> explicit optional<T` *(not a ref)*`>::optional( InPlaceFactory const& f );`

`template<TypedInPlaceFactory> explicit optional<T` *(not a ref)*`>::optional( TypedInPlaceFactory const& f );`

- **Effect:** Constructs an `optional` with a value of `T` obtained from the factory.
- **Postconditions:** `*this` is initialized and its value is *directly given* from the factory `f` (i.e., the value is not copied).
- **Throws:** Whatever the `T` constructor called by the factory throws.
- **Notes:** See In-Place Factories
- **Exception Safety:** Exceptions can only be thrown during the call to the `T` constructor used by the factory; in that case, this constructor has no effect.
- **Example:**

```
class C { C ( char, double, std::string ) ; } ;

C v('A',123.4,"hello");

optional<C> x( in_place   ('A', 123.4, "hello") ); // InPlaceFactory used
optional<C> y( in_place<C>('A', 123.4, "hello") ); // TypedInPlaceFactory used

assert ( *x == v ) ;
assert ( *y == v ) ;
```

`optional& optional<T (not a ref)>::operator= ( T const& rhs ) ;`

- **Effect:** Assigns the value `rhs` to an `optional`.
- **Postconditions:** `*this` is initialized and its value is a *copy* of `rhs`.
- **Throws:** Whatever `T::operator=( T const& )` or `T::T(T const&)` throws.
- **Notes:** If `*this` was initialized, `T`'s assignment operator is used, otherwise, its copy-constructor is used.
- **Exception Safety:** In the event of an exception, the initialization state of `*this` is unchanged and its value unspecified as far as `optional` is concerned (it is up to `T`'s `operator=()`). If `*this` is initially uninitialized and `T`'s *copy constructor* fails, `*this` is left properly uninitialized.
- **Example:**

```
T x;
optional<T> def ;
optional<T> opt(x) ;

T y;
def = y ;
assert ( *def == y ) ;
opt = y ;
assert ( *opt == y ) ;
```

`optional<T&>& optional<T&>::operator= ( T& const& rhs ) ;`

- **Effect:** (Re)binds thee wrapped reference.
- **Postconditions:** `*this` is initialized and it references the same object referenced by `rhs`.
- **Notes:** If `*this` was initialized, is is *rebound* to the new object. See here for details on this behavior.
- **Example:**

```
int a = 1 ;
int b = 2 ;
T& ra = a ;
T& rb = b ;
optional<int&> def ;
optional<int&> opt(ra) ;

def = rb ; // binds 'def' to 'b' through 'rb'
assert ( *def == b ) ;
*def = a ; // changes the value of 'b' to a copy of the value of 'a'
assert ( b == a ) ;
int c = 3;
int& rc = c ;
opt = rc ; // REBINDS to 'c' through 'rc'
c = 4 ;
assert ( *opt == 4 ) ;
```

`optional& optional<T (not a ref)>::operator= ( optional const& rhs ) ;`

- **Effect:** Assigns another `optional` to an `optional`.
- **Postconditions:** If `rhs` is initialized, `*this` is initialized and its value is a *copy* of the value of `rhs`; else `*this` is uninitialized.
- **Throws:** Whatever `T::operator( T const&)` or `T::T( T const& )` throws.
- **Notes:** If both `*this` and `rhs` are initially initialized, `T`'s *assignment operator* is used. If `*this` is initially initialized but `rhs` is uninitialized, `T`'s [destructor] is called. If `*this` is initially uninitialized but `rhs` is initialized, `T`'s *copy constructor* is called.
- **Exception Safety:** In the event of an exception, the initialization state of `*this` is unchanged and its value unspecified as far as optional is concerned (it is up to `T`'s `operator=()`). If `*this` is initially uninitialized and `T`'s *copy constructor* fails, `*this` is left properly uninitialized.
- **Example:**

```
T v;
optional<T> opt(v);
optional<T> def ;

opt = def ;
assert ( !def ) ;
// previous value (copy of 'v') destroyed from within 'opt'.
```

`optional<T&> & optional<T&>::operator= ( optional<T&> const& rhs ) ;`

- **Effect:** (Re)binds thee wrapped reference.
- **Postconditions:** If `*rhs` is initialized, `*this` is initialized and it references the same object referenced by `*rhs`; otherwise, `*this` is uninitialized (and references no object).
- **Notes:** If `*this` was initialized and so is *rhs, this is is *rebound* to the new object. See here for details on this behavior.
- **Example:**

```
int a = 1 ;
int b = 2 ;
T& ra = a ;
T& rb = b ;
optional<int&> def ;
optional<int&> ora(ra) ;
optional<int&> orb(rb) ;

def = orb ; // binds 'def' to 'b' through 'rb' wrapped within 'orb'
assert ( *def == b ) ;
*def = ora ; // changes the value of 'b' to a copy of the value of 'a'
assert ( b == a ) ;
int c = 3;
int& rc = c ;
optional<int&> orc(rc) ;
ora = orc ; // REBINDS ora to 'c' through 'rc'
c = 4 ;
assert ( *ora == 4 ) ;
```

`template<U> optional& optional<T (not a ref)>::operator= ( optional<U> const& rhs ) ;`

- **Effect:** Assigns another convertible optional to an optional.
- **Postconditions:** If `rhs` is initialized, `*this` is initialized and its value is a *copy* of the value of `rhs` *converted* to type `T`; else `*this` is uninitialized.
- **Throws:** Whatever `T::operator=( U const& )` or `T::T( U const& )` throws.
- **Notes:** If both `*this` and rhs are initially initialized, `T`'s *assignment operator* (from `U`) is used. If `*this` is initially initialized but `rhs` is uninitialized, `T`'s *destructor* is called. If `*this` is initially uninitialized but rhs is initialized, `T`'s *converting constructor* (from `U`) is called.
- **Exception Safety:** In the event of an exception, the initialization state of `*this` is unchanged and its value unspecified as far as optional is concerned (it is up to `T`'s `operator=()`). If `*this` is initially uninitialized and `T`'s converting constructor fails, `*this` is left properly uninitialized.
- **Example:**

```
T v;
optional<T> opt0(v);
optional<U> opt1;

opt1 = opt0 ;
assert ( *opt1 == static_cast<U>(v) ) ;
```

`void optional<T (not a ref)>::reset( T const& v ) ;`

- **Deprecated:** same as `operator= ( T const& v) ;`

`void optional<T>::reset() ;`

- **Deprecated:** Same as `operator=( detail::none_t );`

`T const& optional<T (not a ref)>::operator*() const ;`

`T& optional<T (not a ref)>::operator*();`

`T const& optional<T (not a ref)>::get() const ;`

`T& optional<T (not a ref)>::get() ;`

`inline T const& get ( optional<T (not a ref)> const& ) ;`

`inline T& get ( optional<T (not a ref)> &) ;`

- **Requirements:** `*this` is initialized

- **Returns:** A reference to the contained value
- **Throws:** Nothing.
- **Notes:** The requirement is asserted via `BOOST_ASSERT()`.
- **Example:**

```
T v ;
optional<T> opt ( v );
T const& u = *opt;
assert ( u == v ) ;
T w ;
*opt = w ;
assert ( *opt == w ) ;
```

```
T const& optional<T (not a ref)>::get_value_or( T const& default) const ;

T& optional<T (not a ref)>::get_value_or( T& default ) ;

inline T const& get_optional_value_or ( optional<T (not a ref)> const& o, T const& default
) ;

inline T& get_optional_value_or ( optional<T (not a ref)>& o, T& default ) ;
```

- **Returns:** A reference to the contained value, if any, or `default`.
- **Throws:** Nothing.
- **Example:**

```
T v, z ;
optional<T> def;
T const& y = def.get_value_or(z);
assert ( y == z ) ;

optional<T> opt ( v );
T const& u = get_optional_value_or(opt,z);
assert ( u == v ) ;
assert ( u != z ) ;
```

```
T const& optional<T&>::operator*() const ;

T & optional<T&>::operator*();

T const& optional<T&>::get() const ;

T& optional<T&>::get() ;

inline T const& get ( optional<T&> const& ) ;

inline T& get ( optional<T&> &) ;
```

- **Requirements:** `*this` is initialized
- **Returns:**The reference contained.
- **Throws:** Nothing.
- **Notes:** The requirement is asserted via `BOOST_ASSERT()`.
- **Example:**

```
T v ;
T& vref = v ;
optional<T&> opt ( vref );
T const& vref2 = *opt;
assert ( vref2 == v ) ;
++ v ;
assert ( *opt == v ) ;
```

`T const* optional<T` *(not a ref)*`>::get_ptr() const ;`

`T* optional<T` *(not a ref)*`>::get_ptr() ;`

`inline T const* get_pointer ( optional<T` *(not a ref)*`> const& ) ;`

`inline T* get_pointer ( optional<T` *(not a ref)*`> &) ;`

- **Returns:** If `*this` is initialized, a pointer to the contained value; else `0` (*null*).
- **Throws:** Nothing.
- **Notes:** The contained value is permanently stored within `*this`, so you should not hold nor delete this pointer
- **Example:**

```
T v;
optional<T> opt(v);
optional<T> const copt(v);
T* p = opt.get_ptr() ;
T const* cp = copt.get_ptr();
assert ( p == get_pointer(opt) );
assert ( cp == get_pointer(copt) ) ;
```

`T const* optional<T` *(not a ref)*`>::operator ->() const ;`

`T* optional<T` *(not a ref)*`>::operator ->() ;`

- **Requirements:** `*this` is initialized.
- **Returns:** A pointer to the contained value.
- **Throws:** Nothing.
- **Notes:** The requirement is asserted via `BOOST_ASSERT()`.

- **Example:**

```
struct X { int mdata ; } ;
X x ;
optional<X> opt (x);
opt->mdata = 2 ;
```

`optional<T>::operator` *unspecified-bool-type*`() const ;`

- **Returns:** An unspecified value which if used on a boolean context is equivalent to (`get() != 0`)
- **Throws:** Nothing.
- **Example:**

```
optional<T> def ;
assert ( def == 0 );
optional<T> opt ( v ) ;
assert ( opt );
assert ( opt != 0 );
```

`bool optional<T>::operator!() ;`

- **Returns:** If `*this` is uninitialized, `true`; else `false`.
- **Throws:** Nothing.
- **Notes:** This operator is provided for those compilers which can't use the *unspecified-bool-type operator* in certain boolean contexts.
- **Example:**

```
optional<T> opt ;
assert ( !opt );
*opt = some_T ;

// Notice the "double-bang" idiom here.
assert ( !!opt ) ;
```

`bool optional<T>::is_initialized() const ;`

- **Returns:** `true` if the `optional` is initialized, `false` otherwise.
- **Throws:** Nothing.
- **Example:**

```
optional<T> def ;
assert ( !def.is_initialized() );
optional<T> opt ( v ) ;
assert ( opt.is_initialized() );
```

Free functions

`optional<T` *(not a ref)*`> make_optional( T const& v )`

- **Returns:** `optional<T>(v)` for the *deduced* type `T` of `v`.
- **Example:**

```
template<class T> void foo ( optional<T> const& opt ) ;

foo ( make_optional(1+1) ) ; // Creates an optional<int>
```

`optional<T` *(not a ref)*`> make_optional( bool condition, T const& v )`

- **Returns:** `optional<T>(condition,v)` for the *deduced* type `T` of `v`.
- **Example:**

```
optional<double> calculate_foo()
{
  double val = compute_foo();
  return make_optional(is_not_nan_and_finite(val),val);
}

optional<double> v = calculate_foo();
if ( !v )
  error("foo wasn't computed");
```

`bool operator == ( optional<T> const& x, optional<T> const& y );`

- **Returns:** If both `x` and `y` are initialized, `(*x == *y)`. If only `x` or `y` is initialized, `false`. If both are uninitialized, `true`.
- **Throws:** Nothing.
- **Notes:** Pointers have shallow relational operators while `optional` has deep relational operators. Do not use `operator ==` directly in generic code which expect to be given either an `optional<T>` or a pointer; use `equal_pointees()` instead
- **Example:**

```
T x(12);
T y(12);
T z(21);
optional<T> def0 ;
optional<T> def1 ;
optional<T> optX(x);
optional<T> optY(y);
optional<T> optZ(z);

// Identity always hold
assert ( def0 == def0 );
assert ( optX == optX );

// Both uninitialized compare equal
assert ( def0 == def1 );

// Only one initialized compare unequal.
assert ( def0 != optX );

// Both initialized compare as (*lhs == *rhs)
assert ( optX == optY ) ;
assert ( optX != optZ ) ;
```

```
bool operator < ( optional<T> const& x, optional<T> const& y );
```

- **Returns:** If `y` is not initialized, `false`. If `y` is initialized and `x` is not initialized, `true`. If both `x` and `y` are initialized, `(*x < *y)`.
- **Throws:** Nothing.
- **Notes:** Pointers have shallow relational operators while `optional` has deep relational operators. Do not use `operator <` directly in generic code which expect to be given either an `optional<T>` or a pointer; use `less_pointees()` instead.
- **Example:**

```
T x(12);
T y(34);
optional<T> def ;
optional<T> optX(x);
optional<T> optY(y);

// Identity always hold
assert ( !(def < def) );
assert ( optX == optX );

// Both uninitialized compare equal
assert ( def0 == def1 );

// Only one initialized compare unequal.
assert ( def0 != optX );

// Both initialized compare as (*lhs == *rhs)
assert ( optX == optY ) ;
assert ( optX != optZ ) ;
```

```
bool operator != ( optional<T> const& x, optional<T> const& y );
```

- **Returns:** `!( x == y );`
- **Throws:** Nothing.

```
bool operator > ( optional<T> const& x, optional<T> const& y );
```

- **Returns:** `( y < x );`
- **Throws:** Nothing.

```
bool operator <= ( optional<T> const& x, optional<T> const& y );
```

- **Returns:** `!( y<x );`
- **Throws:** Nothing.

```
bool operator >= ( optional<T> const& x, optional<T> const& y );
```

- **Returns:** `!( x<y );`
- **Throws:** Nothing.

```
void swap ( optional<T>& x, optional<T>& y );
```

- **Effect:** If both `x` and `y` are initialized, calls `swap(*x,*y)` using `std::swap`. If only one is initialized, say `x`, calls: `y.reset(*x); x.reset();` If none is initialized, does nothing.
- **Postconditions:** The states of `x` and `y` interchanged.
- **Throws:** If both are initialized, whatever `swap(T&,T&)` throws. If only one is initialized, whatever `T::T ( T const& )` throws.
- **Notes:** If both are initialized, `swap(T&,T&)` is used unqualified but with `std::swap` introduced in scope. If only one is initialized, `T::~T()` and `T::T( T const& )` is called.
- **Exception Safety:** If both are initialized, this operation has the exception safety guarantees of `swap(T&,T&)`. If only one is initialized, it has the same basic guarantee as `optional<T>::reset( T const& )`.
- **Example:**

```
T x(12);
T y(21);
optional<T> def0 ;
optional<T> def1 ;
optional<T> optX(x);
optional<T> optY(y);

boost::swap(def0,def1); // no-op

boost::swap(def0,optX);
assert ( *def0 == x );
assert ( !optX );

boost::swap(def0,optX); // Get back to original values

boost::swap(optX,optY);
assert ( *optX == y );
assert ( *optY == x );
```

Examples

Optional return values

```
optional<char> get_async_input()
{
    if ( !queue.empty() )
        return optional<char>(queue.top());
    else return optional<char>(); // uninitialized
}

void receive_async_message()
{
    optional<char> rcv ;
    // The safe boolean conversion from 'rcv' is used here.
    while ( (rcv = get_async_input()) && !timeout() )
        output(*rcv);
}
```

Optional local variables

```
optional<string> name ;
if ( database.open() )
{
    name.reset ( database.lookup(employer_name) ) ;
}
else
{
    if ( can_ask_user )
        name.reset ( user.ask(employer_name) ) ;
}

if ( name )
    print(*name);
else print("employer's name not found!");
```

Optional data members

```
class figure
{
    public:

    figure()
    {
        // data member 'm_clipping_rect' is uninitialized at this point.
    }

    void clip_in_rect ( rect const& rect )
    {
        ....
        m_clipping_rect.reset ( rect ) ; // initialized here.
    }

    void draw ( canvas& cvs )
    {
        if ( m_clipping_rect )
            do_clipping(*m_clipping_rect);
```

```
        cvs.drawXXX(..);
    }

    // this can return NULL.
    rect const* get_clipping_rect() { return get_pointer(m_clipping_rect); }

    private :

    optional<rect> m_clipping_rect ;

};
```

Bypassing expensive unnecessary default construction

```
class ExpensiveCtor { ... } ;
class Fred
{
    Fred() : mLargeVector(10000) {}

    std::vector< optional<ExpensiveCtor> > mLargeVector ;
} ;
```

Optional references

This library allows the template parameter `T` to be of reference type: `T&`, and to some extent, `T const&`.

However, since references are not real objects some restrictions apply and some operations are not available in this case:

- Converting constructors
- Converting assignment
- InPlace construction
- InPlace assignment
- Value-access via pointer

Also, even though `optional<T&>` treats it wrapped pseudo-object much as a real value, a true real reference is stored so aliasing will ocurr:

- Copies of `optional<T&>` will copy the references but all these references will nonetheless reefer to the same object.
- Value-access will actually provide access to the referenced object rather than the reference itself.

Rebinding semantics for assignment of optional references

If you assign to an *uninitialized* `optional<T&>` the effect is to bind (for the first time) to the object. Clearly, there is no other choice.

```
int x = 1 ;
int& rx = x ;
optional<int&> ora ;
optional<int&> orb(x) ;
ora = orb ; // now 'ora' is bound to 'x' through 'rx'
*ora = 2 ; // Changes value of 'x' through 'ora'
assert(x==2);
```

If you assign to a bare C++ reference, the assignment is forwarded to the referenced object; it's value changes but the reference is never rebound.

```
int a = 1 ;
int& ra = a ;
int b = 2 ;
int& rb = b ;
ra = rb ; // Changes the value of 'a' to 'b'
assert(a==b);
b = 3 ;
assert(ra!=b); // 'ra' is not rebound to 'b'
```

Now, if you assign to an *initialized* `optional<T&>`, the effect is to **rebind** to the new object instead of assigning the referee. This is unlike bare C++ references.

```
int a = 1 ;
int b = 2 ;
int& ra = a ;
int& rb = b ;
optional<int&> ora(ra) ;
optional<int&> orb(rb) ;
ora = orb ; // 'ora' is rebound to 'b'
*ora = 3 ; // Changes value of 'b' (not 'a')
assert(a==1);
assert(b==3);
```

Rationale

Rebinding semantics for the assignment of *initialized* `optional` references has been chosen to provide **consistency among initialization states** even at the expense of lack of consistency with the semantics of bare C++ references. It is true that `optional<U>` strives to behave as much as possible as `U` does whenever it is initialized; but in the case when `U` is `T&`, doing so would result in inconsistent behavior w.r.t to the lvalue initialization state.

Imagine `optional<T&>` forwarding assignment to the referenced object (thus changing the referenced object value but not rebinding), and consider the following code:

```
optional<int&> a = get();
int x = 1 ;
int& rx = x ;
optional<int&> b(rx);
a = b ;
```

What does the assignment do?

If `a` is *uninitialized*, the answer is clear: it binds to `x` (we now have another reference to `x`). But what if `a` is already *initialized*? it would change the value of the referenced object (whatever that is); which is inconsistent with the other possible case.

If `optional<T&>` would assign just like `T&` does, you would never be able to use Optional's assignment without explicitly handling the previous initialization state unless your code is capable of functioning whether after the assignment, `a` aliases the same object as `b` or not.

That is, you would have to discriminate in order to be consistency.

If in your code rebinding to another object is not an option, then is very likely that binding for the fist time isn't either. In such case, assignment to an *uninitialized* `optional<T&>` shall be prohibited. It is quite possible that in such scenario the precondition that the lvalue must be already initialized exist. If it doesn't, then binding for the first time is OK while rebinding is not which is IMO very unlikely. In such scenario, you can assign the value itself directly, as in:

```
assert(!!opt);
*opt=value;
```

In-Place Factories

One of the typical problems with wrappers and containers is that their interfaces usually provide an operation to initialize or assign the contained object as a copy of some other object. This not only requires the underlying type to be Copy Constructible, but also requires the existence of a fully constructed object, often temporary, just to follow the copy from:

```
struct X
{
    X ( int, std:::string ) ;
} ;

class W
{
    X wrapped_ ;

    public:

    W ( X const& x ) : wrapped_(x) {}
} ;

void foo()
{
    // Temporary object created.
    W ( X(123,"hello") ) ;
}
```

A solution to this problem is to support direct construction of the contained object right in the container's storage. In this scheme, the user only needs to supply the arguments to the constructor to use in the wrapped object construction.

```
class W
{
    X wrapped_ ;

    public:

    W ( X const& x ) : wrapped_(x) {}
    W ( int a0, std::string a1) : wrapped_(a0,a1) {}
} ;

void foo()
{
    // Wrapped object constructed in-place
    // No temporary created.
    W (123,"hello") ;
}
```

A limitation of this method is that it doesn't scale well to wrapped objects with multiple constructors nor to generic code were the constructor overloads are unknown.

The solution presented in this library is the family of **InPlaceFactories** and **TypedInPlaceFactories**. These factories are a family of classes which encapsulate an increasing number of arbitrary constructor parameters and supply a method to construct an object of a given type using those parameters at an address specified by the user via placement new.

For example, one member of this family looks like:

```
template<class T,class A0, class A1>
class TypedInPlaceFactory2
{
    A0 m_a0 ; A1 m_a1 ;

    public:

    TypedInPlaceFactory2( A0 const& a0, A1 const& a1 ) : m_a0(a0), m_a1(a1) {}

    void construct ( void* p ) { new (p) T(m_a0,m_a1) ; }
 } ;
```

A wrapper class aware of this can use it as:

```
class W
{
    X wrapped_ ;

    public:

    W ( X const& x ) : wrapped_(x) {}
    W ( TypedInPlaceFactory2 const& fac ) { fac.construct(&wrapped_) ; }
} ;

void foo()
{
    // Wrapped object constructed in-place via a TypedInPlaceFactory.
    // No temporary created.
    W ( TypedInPlaceFactory2<X,int,std::string>(123,"hello")) ;
}
```

The factories are divided in two groups:

- TypedInPlaceFactories: those which take the target type as a primary template parameter.
- InPlaceFactories: those with a template `construct(void*)` member function taking the target type.

Within each group, all the family members differ only in the number of parameters allowed.

This library provides an overloaded set of helper template functions to construct these factories without requiring unnecessary template parameters:

```
template<class A0,...,class AN>
InPlaceFactoryN <A0,...,AN> in_place ( A0 const& a0, ..., AN const& aN) ;

template<class T,class A0,...,class AN>
TypedInPlaceFactoryN <T,A0,...,AN> in_place ( T const& a0, A0 const& a0, ..., AN const& aN) ;
```

In-place factories can be used generically by the wrapper and user as follows:

```
class W
{
    X wrapped_ ;

    public:

    W ( X const& x ) : wrapped_(x) {}

    template< class InPlaceFactory >
    W ( InPlaceFactory const& fac ) { fac.template <X>construct(&wrapped_) ; }

} ;

void foo()
{
    // Wrapped object constructed in-place via a InPlaceFactory.
    // No temporary created.
    W ( in_place(123,"hello") ) ;
}
```

The factories are implemented in the headers: in_place_factory.hpp and typed_in_place_factory.hpp

A note about optional<bool>

`optional<bool>` should be used with special caution and consideration.

First, it is functionally similar to a tristate boolean (false,maybe,true) —such as boost::tribool— except that in a tristate boolean, the maybe state represents a valid value, unlike the corresponding state of an uninitialized `optional<bool>`. It should be carefully considered if an `optional<bool>` instead of a `tribool` is really needed.

Second, `optional<>` provides an implicit conversion to `bool`. This conversion refers to the initialization state and not to the contained value. Using `optional<bool>` can lead to subtle errors due to the implicit `bool` conversion:

```
void foo ( bool v ) ;
void bar()
{
    optional<bool> v = try();

    // The following intended to pass the value of 'v' to foo():
    foo(v);
    // But instead, the initialization state is passed
    // due to a typo: it should have been foo(*v).
}
```

The only implicit conversion is to `bool`, and it is safe in the sense that typical integral promotions don't apply (i.e. if `foo()` takes an `int` instead, it won't compile).

Exception Safety Guarantees

Because of the current implementation (see Implementation Notes), all of the assignment methods:

- `optional<T>::operator= ( optional<T> const& )`
- `optional<T>::operator= ( T const& )`
- `template<class U> optional<T>::operator= ( optional<U> const& )`
- `template<class InPlaceFactory> optional<T>::operator= ( InPlaceFactory const& )`

- `template<class TypedInPlaceFactory> optional<T>::operator= ( TypedInPlaceFactory const& )`
- `optional<T>:::reset ( T const&)`

Can only *guarantee* the basic exception safety: The lvalue optional is left uninitialized if an exception is thrown (any previous value is *first* destroyed using `T::~T()`)

On the other hand, the *uninitializing* methods:

- `optional<T>::operator= ( detail::none_t )`
- `optional<T>::reset()`

Provide the no-throw guarantee (assuming a no-throw `T::~T()`)

However, since `optional<>` itself doesn't throw any exceptions, the only source for exceptions here are `T`'s constructor, so if you know the exception guarantees for `T::T ( T const& )`, you know that `optional`'s assignment and reset has the same guarantees.

```
//
// Case 1: Exception thrown during assignment.
//
T v0(123);
optional<T> opt0(v0);
try
{
    T v1(456);
    optional<T> opt1(v1);
    opt0 = opt1 ;

    // If no exception was thrown, assignment succeeded.
    assert( *opt0 == v1 ) ;
}
catch(...)
{
    // If any exception was thrown, 'opt0' is reset to uninitialized.
    assert( !opt0 ) ;
}

//
// Case 2: Exception thrown during reset(v)
//
T v0(123);
optional<T> opt(v0);
try
{
    T v1(456);
    opt.reset ( v1 ) ;

    // If no exception was thrown, reset succeeded.
    assert( *opt == v1 ) ;
}
catch(...)
{
    // If any exception was thrown, 'opt' is reset to uninitialized.
    assert( !opt ) ;
}
```

Swap

`void swap( optional<T>&, optional<T>& )` has the same exception guarantee as `swap(T&,T&)` when both optionals are initialized. If only one of the optionals is initialized, it gives the same *basic* exception guarantee as `optional<T>::reset( T`

`const& )` (since `optional<T>::reset()` doesn't throw). If none of the optionals is initialized, it has no-throw guarantee since it is a no-op.

Type requirements

In general, `T` must be Copy Constructible and have a no-throw destructor. The copy-constructible requirement is not needed if **InPlaceFactories** are used.

`T` is not required to be Default Constructible.

Implementation Notes

`optional<T>` is currently implemented using a custom aligned storage facility built from `alignment_of` and `type_with_alignment` (both from Type Traits). It uses a separate boolean flag to indicate the initialization state. Placement new with `T`'s copy constructor and `T`'s destructor are explicitly used to initialize,copy and destroy optional values. As a result, `T`'s default constructor is effectively by-passed, but the exception guarantees are basic. It is planned to replace the current implementation with another with stronger exception safety, such as a future `boost::variant`.

Dependencies and Portability

The implementation uses `type_traits/alignment_of.hpp` and `type_traits/type_with_alignment.hpp`

Acknowledgments

Pre-formal review

- Peter Dimov suggested the name 'optional', and was the first to point out the need for aligned storage.
- Douglas Gregor developed 'type_with_alignment', and later Eric Friedman coded 'aligned_storage', which are the core of the optional class implementation.
- Andrei Alexandrescu and Brian Parker also worked with aligned storage techniques and their work influenced the current implementation.
- Gennadiy Rozental made extensive and important comments which shaped the design.
- Vesa Karvonen and Douglas Gregor made quite useful comparisons between optional, variant and any; and made other relevant comments.
- Douglas Gregor and Peter Dimov commented on comparisons and evaluation in boolean contexts.
- Eric Friedman helped understand the issues involved with aligned storage, move/copy operations and exception safety.
- Many others have participated with useful comments: Aleksey Gurotov, Kevlin Henney, David Abrahams, and others I can't recall.

Post-formal review

- William Kempf carefully considered the originally proposed interface and suggested the new interface which is currently used. He also started and fueled the discussion about the analogy optional<>/smart pointer and about relational operators.
- Peter Dimov, Joel de Guzman, David Abrahams, Tanton Gibbs and Ian Hanson focused on the relational semantics of optional (originally undefined); concluding with the fact that the pointer-like interface doesn't make it a pointer so it shall have deep relational operators.
- Augustus Saunders also explored the different relational semantics between optional<> and a pointer and developed the Optional-Pointee concept as an aid against potential conflicts on generic code.

- Joel de Guzman noticed that optional<> can be seen as an API on top of variant<T,nil_t>.
- Dave Gomboc explained the meaning and usage of the Haskell analog to optional<>: the Maybe type constructor (analogy originally pointed out by David Sankel).
- Other comments were posted by Vincent Finn, Anthony Williams, Ed Brey, Rob Stewart, and others.
- Joel de Guzman made the case for the support of references and helped with the proper semantics.
- Mat Marcus shown the virtues of a value-oriented interface, influencing the current design, and contributed the idea of "none".