

---

# Boost.Optional

Fernando Luis Cacciola Carballal

Copyright © 2003-2007 Fernando Luis Cacciola Carballal

Copyright © 2014 Andrzej Krzemiński

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE\_1\_0.txt or copy at [http://www.boost.org/LICENSE\\_1\\_0.txt](http://www.boost.org/LICENSE_1_0.txt))

## Table of Contents

Introduction .....	2
Quick Start .....	3
Optional return values .....	3
Optional automatic variables .....	4
Optional data members .....	4
Bypassing unnecessary default construction .....	5
Storage in containers .....	5
Tutorial .....	7
Motivation .....	7
Design Overview .....	8
When to use Optional .....	11
Relational operators .....	12
Optional references .....	12
Rebinding semantics for assignment of optional references .....	13
In-Place Factories .....	14
A note about optional<bool> .....	17
Exception Safety Guarantees .....	17
Type requirements .....	19
Reference .....	20
Synopsis .....	20
Detailed Semantics .....	22
Dependencies and Portability .....	44
Dependencies .....	44
Optional Reference Binding .....	44
Acknowledgements .....	46

# Introduction

Class template `optional` is a wrapper for representing 'optional' (or 'nullable') objects who may not (yet) contain a valid value. Optional objects offer full value semantics; they are good for passing by value and usage inside STL containers. This is a header-only library.

## Problem

Suppose we want to read a parameter from a config file which represents some integral value, let's call it "MaxValue". It is possible that this parameter is not specified; such situation is no error. It is valid to not specify the parameter and in that case the program is supposed to behave slightly different. Also suppose that any possible value of type `int` is a valid value for "MaxValue", so we cannot just use `-1` to represent the absence of the parameter in the config file.

## Solution

This is how you solve it with `boost::optional`:

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

boost::optional<int> getConfigParam(std::string name); // return either an int or a `not-an-
int`

int main()
{
    if (boost::optional<int> oi = getConfigParam("MaxValue")) // did I get a real int?
        runWithMax(*oi);                                     // use my int
    else
        runWithNoMax();
}
```

# Quick Start

## Optional return values

Let's write and use a converter function that converts an `std::string` to an `int`. It is possible that for a given string (e.g. "cat") there exist no value of type `int` capable of representing the conversion result. We do not consider such situation an error. We expect that the converter can be used only to check if the conversion is possible. A natural signature for this function can be:

```
#include <boost/optional.hpp>
boost::optional<int> convert(const std::string& text);
```

All necessary functionality can be included with one header `<boost/optional.hpp>`. The above function signature means that the function can either return a value of type `int` or a flag indicating that no value of `int` is available. This does not indicate an error. It is like one additional value of `int`. This is how we can use our function:

```
const std::string& text = /*... */;
boost::optional<int> oi = convert(text); // move-construct
if (oi)                                // contextual conversion to bool
    int i = *oi;                        // operator*
```

In order to test if `optional` contains a value, we use the contextual conversion to `bool`. Because of this we can combine the initialization of the optional object and the test into one instruction:

```
if (boost::optional<int> oi = convert(text))
    int i = *oi;
```

We extract the contained value with `operator*` (and with `operator->` where it makes sense). An attempt to extract the contained value of an uninitialized optional object is an *undefined behaviour* (UB). This implementation guards the call with `BOOST_ASSERT`. Therefore you should be sure that the contained value is there before extracting. For instance, the following code is reasonably UB-safe:

```
int i = *convert("100");
```

This is because we know that string value "100" converts to a valid value of `int`. If you do not like this potential UB, you can use an alternative way of extracting the contained value:

```
try {
    int j = convert(text).value();
}
catch (const boost::bad_optional_access&) {
    // deal with it
}
```

This version throws an exception upon an attempt to access a non-existent contained value. If your way of dealing with the missing value is to use some default, like 0, there exists a yet another alternative:

```
int k = convert(text).value_or(0);
```

This uses the `atoi`-like approach to conversions: if `text` does not represent an integral number just return 0. Finally, you can provide a callback to be called when trying to access the contained value fails:

```
int fallback_to_default()
{
    cerr << "could not convert; using -1 instead" << endl;
    return -1;
}

int l = convert(text).value_or_eval(fallback_to_default);
```

This will call the provided callback and return whatever the callback returns. The callback can have side effects: they will only be observed when the optional object does not contain a value.

Now, let's consider how function `convert` can be implemented.

```
boost::optional<int> convert(const std::string& text)
{
    std::stringstream s(text);
    int i;
    if ((s >> i) && s.get() == std::char_traits<char>::eof())
        return i;
    else
        return boost::none;
}
```

Observe the two return statements. `return i` uses the converting constructor that can create `optional<T>` from `T`. Thus constructed optional object is initialized and its value is a copy of `i`. The other return statement uses another converting constructor from a special tag `boost::none`. It is used to indicate that we want to create an uninitialized optional object.

## Optional automatic variables

We could write function `convert` in a slightly different manner, so that it has a single return-statement:

```
boost::optional<int> convert(const std::string& text)
{
    boost::optional<int> ans;
    std::stringstream s(text);
    int i;
    if ((s >> i) && s.get() == std::char_traits<char>::eof())
        ans = i;

    return ans;
}
```

The default constructor of `optional` creates an uninitialized optional object. Unlike with `ints` you cannot have an `optional<int>` in an indeterminate state. Its state is always well defined. Instruction `ans = i` initializes the optional object. It uses the assignment from `int`. In general, for `optional<T>`, when an assignment from `T` is invoked, it can do two things. If the optional object is not initialized our case here), it initializes it with `T`'s copy constructor. If the optional object is already initialized, it assigns the new value to it using `T`'s copy assignment.

## Optional data members

Suppose we want to implement a *lazy load* optimization. This is because we do not want to perform an expensive initialization of our `Resource` until (if at all) it is really used. We can do it this way:

```
class Widget
{
    mutable boost::optional<const Resource> resource_;

public:
    Widget() {}

    const Resource& getResource() const // not thread-safe
    {
        if (resource_ == boost::none)
            resource_.emplace("resource", "arguments");

        return *resource_;
    }
};
```

optional's default constructor creates an uninitialized optional. No call to Resource's default constructor is attempted. Resource doesn't have to be `DefaultConstructible`. In function `getResource` we first check if `resource_` is initialized. This time we do not use the contextual conversion to `bool`, but a comparison with `boost::none`. These two ways are equivalent. Function `emplace` initializes the optional in-place by perfect-forwarding the arguments to the constructor of `Resource`. No copy- or move-construction is involved here. `Resource` doesn't even have to be `MoveConstructible`.



### Note

Function `emplace` is only available on compilers that support rvalue references and variadic templates. If your compiler does not support these features and you still need to avoid any move-constructions, use [In-Place Factories](#).

## Bypassing unnecessary default construction

Suppose we have class `Date`, which does not have a default constructor: there is no good candidate for a default date. We have a function that returns two dates in form of a `boost::tuple`:

```
boost::tuple<Date, Date> getPeriod();
```

In other place we want to use the result of `getPeriod`, but want the two dates to be named: `begin` and `end`. We want to implement something like 'multiple return values':

```
Date begin, end; // Error: no default ctor!
boost::tie(begin, end) = getPeriod();
```

The second line works already, this is the capability of [Boost.Tuple](#) library, but the first line won't work. We could set some invented initial dates, but it is confusing and may be an unacceptable cost, given that these values will be overwritten in the next line anyway. This is where `optional` can help:

```
boost::optional<Date> begin, end;
boost::tie(begin, end) = getPeriod();
```

It works because inside `boost::tie` a move-assignment from `T` is invoked on `optional<T>`, which internally calls a move-constructor of `T`.

## Storage in containers

Suppose you want to ask users to choose some number (an `int`). One of the valid responses is to choose nothing, which is represented by an uninitialized `optional<int>`. You want to make a histogram showing how many times each choice was made. You can use an `std::map`:

```
std::map<boost::optional<int>, int> choices;

for (int i = 0; i < LIMIT; ++i) {
    boost::optional<int> choice = readChoice();
    ++choices[choice];
}
```

This works because `optional<T>` is `LessThanComparable` whenever `T` is `LessThanComparable`. In this case the state of being uninitialized is treated as a yet another value of `T`, which is compared less than any value of `T`.

# Tutorial

## 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 nonexistent 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.

## Design Overview

### 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*. (cf. 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* (cf. 8.5/11). `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*—where *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 precedent 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 uninitialized 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 nonexistent objects.

This meaning of the null pointer value allowed pointers to become 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 nonexistent) 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 contextual 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.

## When to use Optional

It is recommended to use `optional<T>` in situations where there is exactly one, clear (to all parties) reason for having no value of type `T`, and where the lack of value is as natural as having any regular value of `T`. One example of such situation is asking the user in some GUI form to optionally specify some limit on an `int` value, but the user is allowed to say 'I want the number not to be constrained by the maximum'. For another example, consider a config parameter specifying how many threads the application should launch. Leaving this parameter unspecified means that the application should decide itself. For yet another example, consider a function returning the index of the smallest element in a `vector`. We need to be prepared for the situation, where the `vector` is empty. Therefore a natural signature for such function would be:

```
template <typename T>
optional<size_t> find_smallest_elem(const std::vector<T>& vec);
```

Here, having received an empty `vec` and having no `size_t` to return is not a *failure* but a *normal*, albeit irregular, situation.

Another typical situation is to indicate that we do not have a value yet, but we expect to have it later. This notion can be used in implementing solutions like lazy initialization or a two-phase initialization.

`optional` can be used to take a non-`DefaultConstructible` type `T` and create a sibling type with a default constructor. This is a way to add a *null-state* to any type that doesn't have it already.

Sometimes type `T` already provides a built-in null-state, but it may still be useful to wrap it into `optional`. Consider `std::string`. When you read a piece of text from a GUI form or a DB table, it is hardly ever that the empty string indicates anything else but a missing text. And some data bases do not even distinguish between a null string entry and a non-null string of length 0. Still, it may be practical to use `optional<string>` to indicate in the returned type that we want to treat the empty string in a special dedicated program path:

```
if(boost::optional<std::string> name = ask_user_name()) {
    assert(*name != "");
    logon_as(*name);
}
else {
    skip_logon();
}
```

In the example above, the assertion indicates that if we choose to use this technique, we must translate the empty string state to an optional object with no contained value (inside function `ask_user_name`).

## Not recommended usages

It is not recommended to use `optional` to indicate that we were not able to compute a value because of a *failure*. It is difficult to define what a failure is, but it usually has one common characteristic: an associated information on the cause of the failure. This can be the type and member data of an exception object, or an error code. It is a bad design to signal a failure and not inform about the cause. If you do not want to use exceptions, and do not like the fact that by returning error codes you cannot return the computed value, you can use `Expected` library. It is sort of `Boost.Variant` that contains either a computed value or a reason why the computation failed.

Sometimes the distinction into what is a failure and what is a valid but irregular result is blurry and depends on a particular usage and personal preference. Consider a function that converts a `string` to an `int`. Is it a failure that you cannot convert? It might in some cases, but in other you may call it exactly for the purpose of figuring out if a given `string` is convertible, and you are not even interested in the resulting value. Sometimes when a conversion fails you may not consider it a failure, but you need to know why it cannot be converted; for instance at which character it is determined that the conversion is impossible. In this case returning `optional<T>` will not suffice. Finally, there is a use case where an input string that does not represent an `int` is not a failure condition, but during the conversion we use resources whose acquisition may fail. In that case the natural representation is to both return `optional<int>` and signal failure:

```
optional<int> convert1(const string& str); // throws
expected<ErrorT, optional<int>> convert2(const string& str); // return either optional or error
```

## Relational operators

Type `optional<T>` is [EqualityComparable](#) whenever `T` is [EqualityComparable](#). Two optional objects containing a value compare in the same as their contained values. The uninitialized state of `optional<T>` is treated as a distinct value, equal to itself, and unequal to any value of type `T`:

```
boost::optional<int> oN = boost::none;
boost::optional<int> o0 = 0;
boost::optional<int> o1 = 1;

assert(oN != o0);
assert(o1 != oN);
assert(o2 != o1);
assert(oN == oN);
assert(o0 == o0);
```

The converting constructor from `T` as well as from `boost::none` implies the existence and semantics of the mixed comparison between `T` and `optional<T>` as well as between `none_t` and `optional<T>`:

```
assert(oN != 0);
assert(o1 != boost::none);
assert(o2 != 1);
assert(oN == boost::none);
assert(o0 == 0);
```

This mixed comparison has a practical interpretation, which is occasionally useful:

```
boost::optional<int> choice = ask_user();
if (choice == 2)
    start_procedure_2();
```

In the above example, the meaning of the comparison is 'user chose number 2'. If user chose nothing, he didn't choose number 2.

In case where `optional<T>` is compared to `none`, it is not required that `T` be [EqualityComparable](#).

In a similar manner, type `optional<T>` is [LessThanComparable](#) whenever `T` is [LessThanComparable](#). The optional object containing no value is compared less than any value of `T`. To illustrate this, if the default ordering of `size_t` is `{0, 1, 2, ...}`, the default ordering of `optional<size_t>` is `{boost::none, 0, 1, 2, ...}`. This order does not have a practical interpretation. The goal is to have any semantically correct default ordering in order for `optional<T>` to be usable in ordered associative containers (wherever `T` is usable).

Mixed relational operators are the only case where the contained value of an optional object can be inspected without the usage of value accessing function (`operator*`, `value`, `value_or`).

## 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 occur:

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



### Warning

On compilers that do not conform to Standard C++ rules of reference binding, operations on optional references might give adverse results: rather than binding a reference to a designated object they may create an unexpected temporary and bind to it. For more details see [Dependencies and Portability section](#).

## Rvalue references

Rvalue references and lvalue references to `const` have the ability in C++ to extend the life time of a temporary they bind to. Optional references do not have this capability, therefore to avoid surprising effects it is not possible to initialize an optional references from a temporary. Optional rvalue references are disabled altogether. Also, the initialization and assignment of an optional reference to `const` from rvalue reference is disabled.

```
const int& i = 1;           // legal
optional<const int&> oi = 1; // illegal
```

## 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; its 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 consistent.

If in your code rebinding to another object is not an option, then it is very likely that binding for the first time isn't either. In such case, assignment to an *uninitialized* `optional<T&>` shall be prohibited. It is quite possible that in such a scenario it is a precondition that the lvalue must be already initialized. If it isn't, then binding for the first time is OK while rebinding is not which is IMO very unlikely. In such a 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 [CopyConstructible](#), 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 where 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, although optional<> provides a contextual conversion to bool in C++11, this falls back to an implicit conversion on older compilers. 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).

Third, mixed comparisons with bool work differently than similar mixed comparisons between pointers and bool, so the results might surprise you:

```
optional<bool> oEmpty(none), oTrue(true), oFalse(false);

if (oEmpty == none); // renders true
if (oEmpty == false); // renders false!
if (oEmpty == true); // renders false!

if (oFalse == none); // renders false
if (oFalse == false); // renders true!
if (oFalse == true); // renders false

if (oTrue == none); // renders false
if (oTrue == false); // renders false
if (oTrue == true); // renders true
```

In other words, for optional<>, the following assertion does not hold:

```
assert((opt == false) == (!opt));
```

## Exception Safety Guarantees

This library assumes that T's destructor does not throw exceptions. If it does, the behaviour of many operations on optional<T> is undefined.

The following mutating operations never throw exceptions:

- optional<T>::operator= ( none\_t ) noexcept
- optional<T>::reset() noexcept

In addition, the following constructors and the destructor never throw exceptions:

- `optional<T>::optional() noexcept`
- `optional<T>::optional( none_t ) noexcept`

Regarding the following assignment functions:

- `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& )`

They forward calls to the corresponding T's constructors or assignments (depending on whether the optional object is initialized or not); so if both T's constructor and the assignment provide strong exception safety guarantee, `optional<T>`'s assignment also provides strong exception safety guarantee; otherwise we only get the basic guarantee. Additionally, if both involved T's constructor and the assignment never throw, `optional<T>`'s assignment also never throws.

Unless T's constructor or assignment throws, assignments to `optional<T>` do not throw anything else on its own. A throw during assignment never changes the initialization state of any optional object involved:

```
optional<T> opt1(val1);
optional<T> opt2(val2);
assert(opt1);
assert(opt2);

try
{
    opt1 = opt2; // throws
}
catch(...)
{
    assert(opt1);
    assert(opt2);
}
```

This also applies to move assignments/constructors. However, move operations are made no-throw more often.

Operation `emplace` provides basic exception safety guarantee. If it throws, the optional object becomes uninitialized regardless of its initial state, and its previous contained value (if any) is destroyed. It doesn't call any assignment or move/copy constructor on T.

## Swap

Unless `swap` on `optional` is customized, its primary implementation forwards calls to T's `swap` or move constructor (depending on the initialization state of the optional objects). Thus, if both T's `swap` and move constructor never throw, `swap` on `optional<T>` never throws. Similarly, if both T's `swap` and move constructor offer strong guarantee, `swap` on `optional<T>` also offers a strong guarantee.

In case `swap` on `optional` is customized, the call to T's move constructor are replaced with the calls to T's default constructor followed by `swap`. (This is more useful on older compilers that do not support move semantics, when one wants to achieve stronger exception safety guarantees.) In this case the exception safety guarantees for `swap` are reliant on the guarantees of T's `swap` and default constructor

## Type requirements

The very minimum requirement of `optional<T>` is that `T` is a complete type and that it has a publicly accessible destructor. `T` doesn't even need to be constructible. You can use a very minimum interface:

```
optional<T> o;           // uninitialized
assert(o == none);      // check if initialized
assert(!o);             //
o.value();               // always throws
```

But this is practically useless. In order for `optional<T>` to be able to do anything useful and offer all the spectrum of ways of accessing the contained value, `T` needs to have at least one accessible constructor. In that case you need to initialize the optional object with function `emplace()`, or if your compiler does not support it, resort to [In-Place Factories](#):

```
optional<T> o;
o.emplace("T", "ctor", "params");
```

If `T` is [MoveConstructible](#), `optional<T>` is also [MoveConstructible](#) and can be easily initialized from an rvalue of type `T` and be passed by value:

```
optional<T> o = make_T();
optional<T> p = optional<T>();
```

If `T` is [CopyConstructible](#), `optional<T>` is also [CopyConstructible](#) and can be easily initialized from an lvalue of type `T`:

```
T v = make_T();
optional<T> o = v;
optional<T> p = o;
```

If `T` is not [MoveAssignable](#), it is still possible to reset the value of `optional<T>` using function `emplace()`:

```
optional<const T> o = make_T();
o.emplace(make_another_T());
```

If `T` is [Moveable](#) (both [MoveConstructible](#) and [MoveAssignable](#)) then `optional<T>` is also [Moveable](#) and additionally can be constructed and assigned from an rvalue of type `T`.

Similarly, if `T` is [Copyable](#) (both [CopyConstructible](#) and [CopyAssignable](#)) then `optional<T>` is also [Copyable](#) and additionally can be constructed and assigned from an lvalue of type `T`.

`T` is not required to be [DefaultConstructible](#).

# Reference

## Synopsis

```
// In Header: <boost/optional/optional.hpp>

namespace boost {

template<class T>
class optional
{
    public :

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

    optional () noexcept ; R

    optional ( none_t ) noexcept ; R

    optional ( T const& v ) ; R

    optional ( T&& v ) ; R

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

    optional ( optional const& rhs ) ; R

    optional ( optional&& rhs ) noexcept(see below) ; R

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

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

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

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

    optional& operator = ( none_t ) noexcept ; R

    optional& operator = ( T const& v ) ; R

    optional& operator = ( T&& v ) ; R

    optional& operator = ( optional const& rhs ) ; R

    optional& operator = ( optional&& rhs ) noexcept(see below) ; R

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

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

    template<class... Args> void emplace ( Args...&& args ) ; R

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

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

    T const& get() const ; R
}
```

```

T&      get() ; R

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

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

T const& value() const& ; R
T&      value() & ; R
T&&     value() && ; R

template<class U> T value_or( U && v ) const& ; R
template<class U> T value_or( U && v ) && ; R

template<class F> T value_or_eval( F f ) const& ; R
template<class F> T value_or_eval( F f ) && ; R

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

explicit operator bool() const noexcept ; R

bool operator!() const noexcept ; R

// deprecated methods

// (deprecated)
void reset() noexcept ; R

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









// (deprecated)
bool is_initialized() const ; R

// (deprecated)
T const& get_value_or( T const& default ) const ; R
};

template<class T> inline bool operator == ( optional<T> const& x, optional<T> const& y ) ; R
template<class T> inline bool operator != ( optional<T> const& x, optional<T> const& y ) ; R
template<class T> inline bool operator < ( optional<T> const& x, optional<T> const& y ) ; R
template<class T> inline bool operator > ( optional<T> const& x, optional<T> const& y ) ; R
template<class T> inline bool operator <= ( optional<T> const& x, optional<T> const& y ) ; R
template<class T> inline bool operator >= ( optional<T> const& x, optional<T> const& y ) ; R
template<class T> inline bool operator == ( optional<T> const& x, none_t ) noexcept ; R
template<class T> inline bool operator != ( optional<T> const& x, none_t ) noexcept ; R
template<class T> inline optional<T> make_optional ( T const& v ) ; R
template<class T> inline optional<T> make_optional ( bool condition, T const& v ) ; R

```

```

template<class T> inline T const& get_optional_value_or ( optional<T> const& opt, T const& default_value ) ; 
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() noexcept;
```

- **Effect:** Default-Constructs an optional.
- **Postconditions:** `*this` is uninitialized.
- **Notes:** T's default constructor is not called.
- **Example:**

```

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

```

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

- **Effect:** Constructs an optional uninitialized.
- **Postconditions:** `*this` is uninitialized.
- **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 )
```

- **Requires:** `is_copy_constructible<T>::value` is true.
- **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( T&& v )`

- **Requires:** `is_move_constructible<T>::value` is true.
- **Effect:** Directly-Move-Constructs an optional.
- **Postconditions:** `*this` is initialized and its value is move-constructed from `v`.
- **Throws:** Whatever `T::T( T&& )` throws.
- **Notes:** `T::T( T&& )` is called.
- **Exception Safety:** Exceptions can only be thrown during `T::T( T&& )`; in that case, the state of `v` is determined by exception safety guarantees for `T::T(T&&)`.
- **Example:**

```
T v1, v2;  
optional<T> opt(std::move(v1));  
assert ( *opt == v2 ) ;
```

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

- **Notes:** This constructor is deleted

`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 );
```

- **Requires:** `is_copy_constructible<T>::value` is true.
- **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> uninit2 ( 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 refer 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 ) ;
```

`optional<T (not a ref)>::optional( optional&& rhs ) noexcept(see below);`

- **Requires:** `is_move_constructible<T>::value` is true.
- **Effect:** Move-constructs an `optional`.
- **Postconditions:** If `rhs` is initialized, `*this` is initialized and its value is move constructed from `rhs`; else `*this` is uninitialized.
- **Throws:** Whatever `T::T( T&& )` throws.
- **Remarks:** The expression inside `noexcept` is equivalent to `is_nothrow_move_constructible<T>::value`.
- **Notes:** If `rhs` is initialized, `T::T( T && )` is called.
- **Exception Safety:** Exceptions can only be thrown during `T::T( T&& )`; in that case, `rhs` remains initialized and the value of `*rhs` is determined by exception safety of `T::T(T&&)`.
- **Example:**

```
optional<std::unique_ptr<T>> uninit ;
assert (!uninit);

optional<std::unique_ptr<T>> uinit2 ( std::move(uninit) ) ;
assert ( uninit2 == uninit );

optional<std::unique_ptr<T>> init( std::unique_ptr<T>(new T(2)) );
assert ( **init == T(2) ) ;

optional<std::unique_ptr<T>> init2 ( std::move(init) ) ;
assert ( init );
assert ( *init == nullptr );
assert ( init2 );
assert ( **init2 == T(2) ) ;
```

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

- **Effect:** Move-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 refer to the same object (they alias).
- **Example:**

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

optional<std::unique_ptr<T>&> uninit2 ( std::move(uninit) ) ;
assert ( uninit2 == uninit );

std::unique_ptr<T> v(new T(2)) ;
optional<std::unique_ptr<T>&> init(v);
assert ( *init == v ) ;

optional<std::unique_ptr<T>&> init2 ( std::move(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<U> explicit optional<T(not a ref)>::optional( optional<U>&& rhs );
```

- **Effect:** Move-constructs an optional.

- **Postconditions:** If `rhs` is initialized, `*this` is initialized and its value is move-constructed from `*rhs`; else `*this` is uninitialized.
- **Throws:** Whatever `T::T( U&& )` throws.
- **Notes:** `T::T( U&& )` 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&& )`; in that case, `rhs` remains initialized and the value of `*rhs` is determined by exception safety guarantee of `T::T( U&& )`.
- **Example:**

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

optional<int> y(std::move(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( TypedInPlace-
Factory 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>::operator= ( none_t ) noexcept;
```

- **Effect:** If `*this` is initialized destroys its contained value.
- **Postconditions:** `*this` is uninitialized.

```
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& rhs ) ;
```

- **Effect:** (Re)binds the wrapped reference.
- **Postconditions:** `*this` is initialized and it references the same object referenced by `rhs`.
- **Notes:** If `*this` was initialized, it 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= ( T&& rhs ) ;
```

- **Effect:** Moves the value `rhs` to an `optional`.
- **Postconditions:** `*this` is initialized and its value is moved from `rhs`.
- **Throws:** Whatever `T::operator=( T&& )` or `T::T( T && )` throws.
- **Notes:** If `*this` was initialized, `T`'s move-assignment operator is used, otherwise, its move-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 *move constructor* fails, `*this` is left properly uninitialized.
- **Example:**

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

T y1, y2, yR;
def = std::move(y1) ;
assert ( *def == yR ) ;
opt = std::move(y2) ;
assert ( *opt == yR ) ;
```

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

- **Notes:** This assignment operator is deleted.

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

- **Requires:** `T` is [CopyConstructible](#) and [CopyAssignable](#).
- **Effects:**
  - If `!*this && !rhs` no effect, otherwise
  - if `bool(*this) && !rhs`, destroys the contained value by calling `val->T::~~T()`, otherwise
  - if `!*this && bool(rhs)`, initializes the contained value as if direct-initializing an object of type `T` with `*rhs`, otherwise
  - (if `bool(*this) && bool(rhs)`) assigns `*rhs` to the contained value.
- **Returns:** `*this`;
- **Postconditions:** `bool(rhs) == bool(*this)`.
- **Exception Safety:** If any exception is thrown, the initialization state of `*this` and `rhs` remains unchanged. If an exception is thrown during the call to `T`'s copy constructor, no effect. If an exception is thrown during the call to `T`'s copy assignment, the state of its contained value is as defined by the exception safety guarantee of `T`'s copy assignment.

- **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 the 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 *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 ) ;
```

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

- **Requires:** T is MoveConstructible and MoveAssignable.
- **Effects:**
  - If *!\*this && !rhs* no effect, otherwise
  - if *bool(\*this) && !rhs*, destroys the contained value by calling *val->T::~T()*, otherwise
  - if *!\*this && bool(rhs)*, initializes the contained value as if direct-initializing an object of type T with *std::move(\*rhs)*, otherwise

- (if `bool(*this) && bool(rhs)`) assigns `std::move(*rhs)` to the contained value.
- **Returns:** `*this`;
- **Postconditions:** `bool(rhs) == bool(*this)`.
- **Remarks:** The expression inside `noexcept` is equivalent to `is_nothrow_move_constructible<T>::value && is_nothrow_move_assignable<T>::value`.
- **Exception Safety:** If any exception is thrown, the initialization state of `*this` and `rhs` remains unchanged. If an exception is thrown during the call to `T`'s move constructor, the state of `*rhs` is determined by the exception safety guarantee of `T`'s move constructor. If an exception is thrown during the call to `T`'s move-assignment, the state of `**this` and `*rhs` is determined by the exception safety guarantee of `T`'s move assignment.
- **Example:**

```
optional<T> opt(T(2)) ;
optional<T> def ;

opt = def ;
assert ( def ) ;
assert ( opt ) ;
assert ( *opt == T(2) ) ;
```

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

- **Effect:** Same as `optional<T&>::operator= ( optional<T&> const& rhs )`.

```
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) ) ;
```



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

- **Effect:** Move-assigns another convertible optional to an optional.
- **Postconditions:** If `rhs` is initialized, `*this` is initialized and its value is moved from the value of `rhs`; else `*this` is uninitialized.
- **Throws:** Whatever `T::operator=( U&& )` or `T::T( U&& )` 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 = std::move(opt0) ;
assert ( opt0 );
assert ( opt1 );
assert ( *opt1 == static_cast<U>(v) ) ;
```

```
template<class... Args> void optional<T (not a ref)>::emplace( Args...&& args );
```

- **Requires:** The compiler supports rvalue references and variadic templates.
- **Effect:** If `*this` is initialized calls `*this = none`. Then initializes in-place the contained value as if direct-initializing an object of type `T` with `std::forward<Args>(args)...`
- **Postconditions:** `*this` is initialized.
- **Throws:** Whatever the selected `T`'s constructor throws.
- **Notes:** `T` need not be `MoveConstructible` or `MoveAssignable`. On compilers that do not support variadic templates, the signature falls back to single-argument: `template<class Arg> void emplace(Arg&& arg)`. On compilers that do not support rvalue references, the signature falls back to two overloads: taking `const` and non-`const` lvalue reference.
- **Exception Safety:** If an exception is thrown during the initialization of `T`, `*this` is *uninitialized*.
- **Example:**

```
T v;
optional<const T> opt;
opt.emplace(0); // create in-place using ctor T(int)
opt.emplace(); // destroy previous and default-construct another T
opt.emplace(v); // destroy and copy-construct in-place (no assignment called)
```

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

```
template<TypedInPlaceFactory> optional<T>& optional<T (not a ref)>::operator=( TypedIn-
PlaceFactory const& f );
```

- **Effect:** Assigns 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, the optional object will be reset to be *uninitialized*.

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

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

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

- **Deprecated:** Same as operator=( none\_t ) ;

```
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)> & ) ;
```

- **Requires:** \*this is initialized
- **Returns:** A reference to the contained value
- **Throws:** Nothing.
- **Notes:** The requirement is asserted via BOOST\_ASSERT( ).

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

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

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

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

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

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

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

- **Requires:** `*this` is initialized
- **Returns:** A reference to the contained value
- **Throws:** Nothing.
- **Notes:** The requirement is asserted via `BOOST_ASSERT()`. On compilers that do not support ref-qualifiers on member functions these two overloads are replaced with the classical two: a `const` and non-`const` member functions.
- **Example:**

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

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

- **Requires:** `*this` contains a value.
- **Effects:** Equivalent to `return std::move(*val);`.
- **Notes:** The requirement is asserted via `BOOST_ASSERT()`. On compilers that do not support ref-qualifiers on member functions this overload is not present.

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

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

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

- **Requires:** `*this` is initialized
- **Returns:** The reference contained.

- **Throws:** Nothing.
- **Notes:** The requirement is asserted via `BOOST_ASSERT()`. On compilers that do not support ref-qualifiers on member functions these three overloads are replaced with the classical two: a `const` and non-`const` member functions.
- **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>::value() const& ;
```

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

- **Effects:** Equivalent to `return bool(*this) ? *val : throw bad_optional_access();`.
- **Notes:** On compilers that do not support ref-qualifiers on member functions these two overloads are replaced with the classical two: a `const` and non-`const` member functions.
- **Example:**

```
T v ;
optional<T> o0, o1 ( v );
assert ( o1.value() == v );

try {
    o0.value(); // throws
    assert ( false );
}
catch(bad_optional_access&) {
    assert ( true );
}
```

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

- **Effects:** Equivalent to `return bool(*this) ? std::move(*val) : throw bad_optional_access();`.
- **Notes:** On compilers that do not support ref-qualifiers on member functions this overload is not present.

```
template<class U> T optional<T>::value_or(U && v) const& ;
```

- **Effects:** Equivalent to `if (*this) return **this; else return std::forward<U>(v);`.
- **Remarks:** If `T` is not `CopyConstructible` or `U &&` is not convertible to `T`, the program is ill-formed.

- **Notes:** On compilers that do not support ref-qualifiers on member functions this overload is replaced with the `const`-qualified member function. On compilers without rvalue reference support the type of `v` becomes `U const&`.

```
template<class U> T optional<T>::value_or(U && v) && ;
```

- **Effects:** Equivalent to `if (*this) return std::move(**this); else return std::forward<U>(v);`.
- **Remarks:** If `T` is not `MoveConstructible` or `U &&` is not convertible to `T`, the program is ill-formed.
- **Notes:** On compilers that do not support ref-qualifiers on member functions this overload is not present.

```
template<class F> T optional<T>::value_or_eval(F f) const& ;
```

- **Requires:** `T` is `CopyConstructible` and `F` models a `Generator` whose result type is convertible to `T`.
- **Effects:** `if (*this) return **this; else return f();`.
- **Notes:** On compilers that do not support ref-qualifiers on member functions this overload is replaced with the `const`-qualified member function.
- **Example:**

```
int complain_and_0()
{
    clog << "no value returned, using default" << endl;
    return 0;
}

optional<int> o1 = 1;
optional<int> oN = none;

int i = o1.value_or_eval(complain_and_0); // fun not called
assert (i == 1);

int j = oN.value_or_eval(complain_and_0); // fun called
assert (i == 0);
```

```
template<class F> T optional<T>::value_or_eval(F f) && ;
```

- **Requires:** `T` is `MoveConstructible` and `F` models a `Generator` whose result type is convertible to `T`.
- **Effects:** `if (*this) return std::move(**this); else return f();`.
- **Notes:** On compilers that do not support ref-qualifiers on member functions this overload is not present.

```
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 ) ;
```

- **Deprecated:** Use `value_or()` instead.
- **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(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 ->() ;
```

- **Requires:** `*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 ;
```

```
explicit optional<T>::operator bool() const noexcept ;
```

- **Returns:** `get_ptr() != 0`.
- **Notes:** On compilers that do not support explicit conversion operators this falls back to safe-bool idiom.
- **Example:**

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

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

- **Returns:** If `*this` is uninitialized, true; else false.
- **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 ;
```

- **Deprecated:** Same as `explicit operator bool () ;`

## 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 );
```

- **Requires:** T shall meet requirements of [EqualityComparable](#).
- **Returns:** If both x and y are initialized, (\*x == \*y). If only x or y is initialized, false. If both are uninitialized, true.
- **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:**



```
optional<T> oN, oN_;
optional<T> o1(T(1)), o1_(T(1));
optional<T> o2(T(2));

assert ( oN == oN_ ); // Identity implies equality
assert ( o1 == o1_ ); //

assert ( oN == oN_ ); // Both uninitialized compare equal

assert ( oN != o1 ); // Initialized unequal to initialized.

assert ( o1 == o1_ ); // Both initialized compare as (*lhs == *rhs)
assert ( o1 != o2 ); //
```

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

- **Requires:** Expression `*x < *y` shall be well-formed and its result shall be convertible to `bool`.
- **Returns:** `(!y) ? false : (!x) ? true : *x < *y`.
- **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. `T` need not be `LessThanComparable`. Only single `operator<` is required. Other relational operations are defined in terms of this one. If `T`'s `operator<` satisfies the axioms of `LessThanComparable` (transitivity, antisymmetry and irreflexivity), `optional<T>` is `LessThanComparable`.
- **Example:**

```
optional<T> oN, oN_;
optional<T> o0(T(0));
optional<T> o1(T(1));

assert ( !(oN < oN) ); // Identity implies equivalence
assert ( !(o1 < o1) );

assert ( !(oN < oN_) ); // Two uninitialized are equivalent
assert ( !(oN_ < oN) );

assert ( oN < o0 ); // Uninitialized is less than initialized
assert ( !(o0 < oN) );

assert ( o1 < o2 ); // Two initialized compare as (*lhs < *rhs)
assert ( !(o2 < o1) );
assert ( !(o2 < o2) );
```

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

- **Returns:** `!( x == y );`

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

- **Returns:** ( y < x );

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

- **Returns:** !( y < x );

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

- **Returns:** !( x < y );

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

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

- **Returns:** !x.
- **Notes:** T need not meet requirements of [EqualityComparable](#).

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

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

- **Returns:** !( x == y );

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

- **Requires:** Lvalues of type T shall be swappable and T shall be MoveConstructible.
- **Effects:**
  - If !\*this && !rhs, no effect, otherwise
  - if bool(\*this) && !rhs, initializes the contained value of rhs as if direct-initializing an object of type T with the expression `std::move(*(*this))`, followed by `val->T::~~T()`, \*this does not contain a value and rhs contains a value, otherwise
  - if !\*this && bool(rhs), initializes the contained value of \*this as if direct-initializing an object of type T with the expression `std::move(*rhs)`, followed by `rhs.val->T::~~T()`, \*this contains a value and rhs does not contain a value, otherwise
  - (if bool(\*this) && bool(rhs)) calls `swap(*(*this), *rhs)`.
- **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&& )` throws.

- **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 );
```

# Dependencies and Portability

## Dependencies

The implementation uses the following other Boost modules:

1. assert
2. config
3. core
4. detail
5. move
6. mpl
7. static\_assert
8. throw\_exception
9. type\_traits
10. utility

## Optional Reference Binding

On compilers that do not conform to Standard C++ rules of reference binding, operations on optional references might give adverse results: rather than binding a reference to a designated object they may create an unexpected temporary and bind to it. Compilers known to have these deficiencies include GCC versions 4.2, 4.3, 4.4, 4.5; QCC 4.4.2; MSVC versions 8.0, 9.0, 10.0, 11.0, 12.0. On these compilers prefer using direct-initialization and copy assignment of optional references to copy-initialization and assignment from T&:

```
const int i = 0;
optional<const int&> or1;
optional<const int&> or2 = i; // not portable
or1 = i;                     // not portable

optional<const int&> or3(i); // portable
or1 = optional<const int&>(i); // portable
```

In order to check if your compiler correctly implements reference binding use this test program.

```
#include <cassert>

const int global_i = 0;

struct TestingReferenceBinding
{
    TestingReferenceBinding(const int& ii)
    {
        assert(&ii == &global_i);
    }

    void operator=(const int& ii)
    {
        assert(&ii == &global_i);
    }

    void operator=(int&&) // remove this if your compiler doesn't have rvalue refs
    {
        assert(false);
    }
};

int main()
{
    const int& iref = global_i;
    assert(&iref == &global_i);

    TestingReferenceBinding ttt = global_i;
    ttt = global_i;

    TestingReferenceBinding ttt2 = iref;
    ttt2 = iref;
}
```

# Acknowledgements

## 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".
- Vladimir Batov's design of Boost.Convert library motivated the development of value accessors for optional: functions value, value\_or, value\_or\_eval.