A proposal to add a utility class to represent optional objects (Revision 4)

ISO/IEC JTC1 SC22 WG21 N3672 2013-04-19

Fernando Cacciola, fernando.cacciola@gmail.com
Andrzej Krzemieński, akrzemi1@gmail.com

Introduction

Class template `optional<T>` proposed here is a type that may or may not store a value of type `T` in its storage space. Its interface allows to query if a value of type `T` is currently stored, and if so, to access it. The interface is based on Fernando Cacciola's Boost.Optional library[2], shipping since March, 2003, and widely used. It requires no changes to core language, and breaks no existing code.

Table of contents

Revision history

Changes since R4C

- Removed all relational operators except operator== and operator<.

Changes since N3527:

- Renamed tag `emplace` to `in_place` based on feedback from LEWG.
- Removed the Motivation section.
- Added description of `op = {}` syntax and explained why we had to apply some tricks to support it. See The `op = {}` syntax.
- Clarified why we do not require in function template `swap` that `T` be MoveAssignamle. See Requirements for `swap`.
- Added short discussion on the choice of the header. See Header `<utility>` or `<optional>`?.
- Optional's `operator<` now uses `less<T>` rather than `T::operator<`. See Relational operators for rationale.
- Provided a comparison between `optional` in this proposal and `boost::optional`. See Comparison with Boost.Optional.
- Added "open questions" section.
- Reduced the discussion in Rationale section by omitting some alternatives we considered but rejected.
- Removed namespace `experimental`, because we are not targetting a TS anymore.
- Applied fixes in the standardese as per LWG's feedback.
- Function `emplace` now returns `void` — LWG recommendation.
- All mixed ordering relations now use only `less<T>` in implementation — LWG recommendation.
- Clarified (in section Overview of `optional`) that Optional cannot be implemented with `aligned_storage` .
- Reworded complicated Effects elements for copy/move assignment and swap, so that they are easier to read — LWG recommendation.

Changes since N3406=12-0096:

- Wording changes are now relative to N3485.
- `optional<T>` is hashable for hashable `T`'s.
- Optional refereces are now an auxiliary proposal — this gives the possibility to accept optional values without references.
- Optional refereces are now assignamble and swappable.
- `get_value_or` is now `optional`-specific member function, renamed to `value_or`.
- Added member function `value` — an alternative to `operator*` that checks if the object is engaged.
- `optional<T>` is a literal type.
- Mixed relational operations between `optional<T>` and `T` are now allowed.
- Removed reference implementation. We now only provide the implementation of the parts that we consider non-trivial.

Changes since N1878=05-0138:

- Revised wording; the changes are now relative to N3290.
- Removed any form of assignment for optional references.
- Removed duplicate interface for accessing the value stored by the optional.
- Added in-place construction and assignment.
- Now using different tag `nullopt` instead of `nullptr` to indicate the 'disengaged' (uninitialized) optional.
- Included C++11 features: move semantics, `noexcept`, variadic templates, perfect forwarding, static initialization.
- Changed the motivation section.
- Changed the design rationale section.
- Added reference implementation

Impact on the Standard

This proposal depends on library proposal N3471: it requires that Standard Library components `move`, `forward` and member functions of `initializer_list` are `constexpr`. The paper has already been incorporated into the Working Draft of the Standard N3485. This proposal also depends on language proposal N2439 (Rvalue references for `*this`). While this feature proposal has been incorporated into C++11, we are aware of only two compilers that implemented it: Clang and GCC 4.8.1. There is a risk that if compiler vendors do not implement it, they will also not be able to fully implement this proposal. In that case, the signature of member function `optional<T>::value_or` from this proposal will need to be modified.

N3507 (A URI Library for C++) depends on this library.

Overview of `optional`

The primary purpose of `optional<T>`'s interface is to be able to answer the quesiotn "do you contain a value of type `T`?", and iff the answer is "yes", to provide access to the contained value. Conceptually, `optional` can be illustrated by the following structure.

```
template <typename T>
struct optional
{
  bool is_initialized_;
  typename aligned_storage<sizeof(T), alignof(T)>::type storage_;
};
```

Flag `is_initialized_` stores the information if `optional` has been assigned a value of type `T`. `storage_` is a raw fragment of memory (allocated within `optional` object) capable of storing an object of type `T`. Objects in this storage are created and destroyed using placement `new` and pseudo destructor call as member functions of `optional` are executed and based on the value of flag `is_initialized_`.

Note: the above representation is only conceptual. In fact, our aggressive requirements for `constexpr` constructors prevent the usage of `aligned_storage` in the implementation of `optional`. See section Implementability for an overview of a possible reference implementation.

Alternatively, one can think of `optional<T>` as `pair<bool, T>`, with one important difference: if `first` is `false`, member `second` has never been even initialized, even with default constructor or value-initialization.

The basic usage of `optional<T>` can be illustrated with the following example.

```
optional<int> str2int(string);    // converts int to string if possible

int get_int_form_user()
{
  string s;

  for (;;) {
    cin >> s;
    optional<int> o = str2int(s); // 'o' may or may not contain an int
    if (o) {                      // does optional contain a value?
      return *o;                  // use the value
    }
  }
}
```

Interface of `optional`

Default construction of `optional<T>` creates an an object that stores no value of type `T`. No default constructor of `T` is called. `T` doesn't even need to be `DefaultConstructible`. We say that thus created optional object is *disengaged*. We use a pair of somewhat arbitrary names, 'engaged' and 'disengaged'. A default-constructed optional object is initialized (the lifetime of `optional<T>` has started), but it is disengaged (the lifetime of the contained object has not yet started). Trying to access the value of `T` in this state causes undefined behavior. The only thing we can do with a disengaged optional object is to query whether it is engaged, copy it, compare it with another `optional<T>`, or *engage* it.

```
optional<int> oi;                 // create disengaged object
optional<int> oj = nullopt;       // alternative syntax
oi = oj;                          // assign disengaged object
optional<int> ok = oj;            // ok is disengaged

if (oi)  assert(false);           // 'if oi is engaged...'
if (!oi) assert(true);            // 'if oi is disengaged...'

if (oi != nullopt) assert(false); // 'if oi is engaged...'
if (oi == nullopt) assert(true);  // 'if oi is disengaged...'

assert(oi == ok);                 // two disengaged optionals compare equal
```

Tag `nullopt` represents the disengaged state of an optional object. This reflects our conceptual model of optional objects: `optional<T>` can be thought of as a `T` with one additional value `nullopt`. Being disengaged is just another value of `T`. Thus, `optional<unsigned>` can be thought of as a type with possible range of values `{nullopt, 0, 1, ...}`. Value `nullopt` is always picked by the default constructor.

We can create engaged optional objects or engage existing optional objects by using converting constructor (from type `T`) or by assigning a value of type `T`.

```
optional<int> ol{1};              // ol is engaged; its contained value is 1
ok = 2;                           // ok becomes engaged; its contained value is 2
oj = ol;                          // oj becomes engaged; its contained value is 1

assert(oi != ol);                 // disengaged != engaged
assert(ok != ol);                 // different contained values
assert(oj == ol);                 // same contained value
assert(oi < ol);                  // disengaged < engaged
assert(ol < ok);                  // less by contained value
```

Copy constructor and copy assignment of `optional<T>` copies both the engaged/disengaged flag and the contained value, if it exists.

```
optional<int> om{1};              // om is engaged; its contained value is 1
optional<int> on = om;            // on is engaged; its contained value is 1
```

```
om = 2;                            // om is engaged; its contained value is 2
assert (on != om);                 // on still contains 1. They are not pointers
```

We access the contained value using the indirection operator.

```
int i = *ol;                       // i obtains the value contained in ol
assert(i == 1);
*ol = 9;                           // the valuesagety  contained in ol becomes 9
assert(*ol == 9);
```

We also provide consistent `operator->`. Even though `optional` provides operators `->` and `*` it is not a pointer. Unlike pointers, it has full value semantics: deep copy construction, deep copy assignmet, deep equality and less-than comparison, and constness propagation (from optional object to the contained value).

```
int p = 1;
optional<int> op = p;
assert(*op == 1);
p = 2;
assert(*op == 1);                  // value contained in op is separated from p
```

The typical usage of `optional` requires an if-statement.

```
if (ol)
  process(*ol);                    // use contained value if present
else
  processNil();                    // proceed without contained value

if (!om)
  processNil();
else
  process(*om);
```

If the action to be taken for disengaged `optional` is to proceed with the default value, we provide a convenient idiom:

```
process(ol.value_or(0));       // use 0 if ol is disengaged
```

Sometimes the initialization from `T` may not do. If we want to skip the copy/move construction for `T` because it is too expensive or simply not available, we can call an 'emplacing' constructor, or function `emplace`.

```
optional<Guard> oga;                      // Guard is non-copyable (and non-moveable)
optional<Guard> ogb(in_place, "res1");     // initializes the contained value with "res1"
optional<Guard> ogc(in_place);             // default-constructs the contained value

oga.emplace("res1");                      // initializes the contained value with "res1"
oga.emplace();                            // destroys the contained value and
                                          // default-constructs the new one
```

There are two ways to *disengage* a perhaps engaged optional object:

```
ok = nullopt;                       // if ok was engaged calls T's dtor
oj = {};                            // assigns a temporary disengaged optional
oga = nullopt;                      // OK: disengage the optional Guard
ogb = {};                           // ERROR: Guard is not Moveable
```

Optional propagates constness to its contained value:

```
const optional<int> c = 4;
int i = *c;                         // i becomes 4
*c = i;                             // ERROR: cannot assign to const int&
```

Exception safety

Type `optional<T>` is a wrapper over `T`, thus its exception safety guarantees depend on exception safety guarantees of `T`. We expect (as is the case for the entire C++ Standard Library) that destructor of `T` does not throw exceptions. Copy assignment of `optional<T>` provides the same exception guarantee as copy assignment of `T` and copy constructor of `T` (i.e., the weakest of the two). Move assignment of `optional<T>`

provides the same exception guarantee as move assignment of `T` and move constructor of `T`. Member function `emplace` provides basic guarantee: if exception is thrown, `optional<T>` becomes disengaged, regardless of its prior state.

Advanced use cases

With `optional<T>` problems described in Motivation section can be solved as follows. For lazy initialization:

```
class Car
{
  mutable mutex m_;
  mutable optional<const Engine> engine_;
  mutable optional<const int> mileage_

public:
  const Engine& engine() const
  {
    lock_guard<mutex> _(m_);
    if (engine_ == nullopt) engine_.emplace( engineParams() );
    return *engine_;
  }

  const int& mileage() const
  {
    lock_guard<mutex> _(m_);
    if (!mileage_) mileage_ = initMileage();
    return *mileage_;
  }
};
```

The algorithm for finding the greatest element in vector can be written as:

```
optional<int> find_biggest( const vector<int>& vec )
{
  optional<int> biggest;  // initialized to not-an-int
  for (int val : vec) {
    if (!biggest || *biggest < val) {
      biggest = val;
      // or: biggest.emplace(val);
    }
  }
  return biggest;
}
```

Missing return values are naturally modelled by optional values:

```
optional<char> c = stream.getNextChar();
optional<int> x = DB.execute("select ...");

storeChar( c.value_or('\0') );
storeCount( x.value_or(-1) );
```

Optional arguments can be implemented as follows:

```
template <typename T>
T getValue( optional<T> newVal = nullopt )
{
  if (newVal) {
    cached = *newVal;
  }
  return cached;
}
```

Manually controlling the life-time of guard-like objects can be achieved by emplacement operations and `nullopt` assignment:

```
{
  optional<Guard> grd1{in_place, "res1", 1};   // guard 1 initialized
```

```
      optional<Guard> grd2;

      grd2.emplace("res2", 2);                  // guard 2 initialized
      grd1 = nullopt;                           // guard 1 released

    }                                           // guard 2 released (in dtor)
```

It is possible to use `tuple` and `optional` to emulate multiple return valuse for types without default constructor:

```
    tuple<Date, Date, Date> getStartMidEnd();
    void run(Date const&, Date const&, Date const&);
    // ...

    optional<Date> start, mid, end;           // Date doesn't have default ctor (no good default dat

    tie(start, mid, end) = getStartMidEnd();
    run(*start, *mid, *end);
```

Comparison with `value_ptr`

N3339[7] proposes a smart pointer template `value_ptr`. In short, it is a smart pointer with deep copy semantics. It has a couple of features in common with `optional`: both contain the notion of optionality, both are deep-copyable. Below we list the most important differences.

`value_ptr` requires that the pointed-to object is allocated in the free store. This means that the `sizeof(value_ptr<T>)` is fixed irrespective of `T`. `value_ptr` is 'polymorphic': object of type `value_ptr<T>` can point to an object of type `DT`, derived from `T`. The deep copy preserves the dynamic type. `optional` requires no free store allocation: its creation is more efficient; it is not "polymorphic".

Relational operations on `value_ptr` are shallow: only addresses are compared. Relational operations on `optional` are deep, based on object's value. In general, `optional` has a well defined value: being disengaged and the value of contained value (if it exists); this value is expressed in the semantics of equality operator. This makes `optional` a full value-semantic type. Comparison of `value_ptr` does not have this property: copy semantics are incompatible with equality comparison semantics: a copy-constructed `value_ptr` does not compare equal to the original. `value_ptr` is not a value semantic type.

Comparison with Boost.Optional

This proposal basically tries to follow Boost.Optional's interface. Here we list the significant differences.

aspect	**this proposal**	**Boost.Optional**
Move semantics	yes	no
noexcept	yes	no
hash support	yes	no
a throwing value accessor	yes	no
literal type	partially	no
in place construction	`emplace`, tag `in_place`	utility `in_place_factory`
disengaged state tag	`nullopt`	`none`
optional references	no (optionally)	yes
conversion from `optional<U>` to `optional<T>`	no	yes
dupplicated interface functions (`is_initialized`, `reset`, `get`)	no	yes
explicit convert to ptr (`get_ptr`)	no	yes

Design rationale

The very minimum, ascetic interface for optional objects — apart from copy/move — could consist of two constructors and two functions:

```
// not proposed
int i = 9;
optional<int> oi{engaged, i};          // create engaged
optional<int> oj{disengaged};          // create disengaged
if (oi.is_engaged()) {                 // contains value?
  oi.get_value();                      // access the value
}
```

The reason we provide different and richer interface is motivated by users' convenience, performance improvements, secondary goals we want to acheive, and the attempt to standardize the existing practice.

Nearly every function in the interface of `optional` has risen some controversies. Different people have different expectations and different concerns and it is not possible to satisfy all conflicting requirements. Yet, we believe that `optional` is so universally useful, that it is worth standardizing it even at the expense of introducing a controversial interface. The current proposal reflects our arbitrary choice of balance between unambiguousity, genericity and flexibility of the interface. The interface is based on Fernanndo Cacciola's Boost.Optional library,[2] and the users' feedback. The library has been widely accepted, used and even occasionally recommended ever since.

Conceptual model for `optional<T>`

Optional objects serve a number of purposes and a couple of conceptual models can be provided to answer the question what `optional<T>` really is and what interface it should provide. The three most common models are:

1. Just a `T` with deferred initialization (and additional interface to check if the object has already been initialized).
2. A discriminated union of types `nullopt_t` and `T`.
3. A container of `T`'s with the maximum size of 1.

While (1) was the first motivation for `optional`, we do not choose to apply this model, because type `optional<T>` would not be a value semantic type: it would not model concept `Regular` (if C++ had concepts). In particular, it would be not clear whether being engaged or disengaged is part of the object's state. Programmers who wish to adopt this view, and don't mind the mentioned difficulties, can still use `optional` this way:

```
optional<int> oi;
initializeSomehow(oi);

int i = (*oi);
use(*oi);
(*oi) = 2;
cout << (*oi);
```

Note that this usage does not even require to check for engaged state, if one is sure that the object is engaged. One just needs to use indirection operator consistently anywhere one means to use the initialized value.

Model (2) treats `optional<T>` as either a value of type `T` or value `nullopt`, allocated in the same storage, along with the way of determining which of the two it is. The interface in this model requires operations such as comparison to `T`, comparison to `nullopt`, assignment and creation from either. It is easy to determine what the value of the optional object is in this model: the type it stores (`T` or `nullopt_t`) and possibly the value of `T`. This is the model that we propose.

Model (3) treats `optional<T>` as a special case container. This begs for a container-like interface: `empty` to check if the object is disengaged, `emplace` to engage the object, and `clear` to disengage it. Also, the value of optional object in this model is well defined: the size of the container (0 or 1) and the value of the element if the size is 1. This model would serve our pupose equally well. The choice between models (2) and (3) is to a certain degree arbitrary. One argument in favour of (2) is that it has been used in practice for a while in Boost.Optional.

Additionally, within the affordable limits, we propose the view that `optional<T>` just extends the set of the values of `T` by one additional value `nullopt`. This is reflected in initialization, assignment, ordering, and equality comparison with both `T` and `nullopt`.

```
optional<int> oi = 0;
optional<int> oj = 1;
optional<int> ok = nullopt;

oi = 1;
oj = nullopt;
ok = 0;

oi == nullopt;
oj == 0;
ok == 1;
```

Initialization of `optional<T>`

In cases `T` is a value semantic type capable of storing *n* distinct values, `optional<T>` can be seen as an extended `T` capable of storing *n* + 1 values: these that `T` stores and `nullopt`. Any valid initialization scheme must provide a way to put an optional object to any of these states. In addition, some `T`s (like scope guards) are not `MoveConstructible` and their optional variants still should constructible with any set of arguments that work for `T`. Two models have been identified as feasible.

The first requires that you initialize either by providing either an already constructed `T` or the tag `nullopt`.

```
string s{"STR"};

optional<string> os{s};                    // requires Copyable<T>
optional<string> ot = s;                   // requires Copyable<T>
optional<string> ou{"STR"};                // requires Movable<T>
optional<string> ov = string{"STR"};       // requires Movable<T>

optional<string> ow;                       // disengaged
optional<string> ox{};                     // disengaged
optional<string> oy = {};                  // disengaged
optional<string> oz = optional<string>{};  // disengaged
optional<string> op{nullopt};              // disengaged
optional<string> oq = {nullopt};           // disengaged
```

In order to avoid calling move/copy constructor of `T`, we use a 'tagged' placement constructor:

```
optional<Guard> og;                        // disengaged
optional<Guard> oh{};                      // disengaged
optional<Guard> oi{in_place};               // calls Guard{} in place
optional<Guard> oj{in_place, "arg"};        // calls Guard{"arg"} in place
```

The in-place constructor is not strictly necessary. It could be dropped because one can always achieve the same effect with a two-liner:

```
optional<Guard> oj;                        // start disengaged
oj.emplace("arg");                         // now engage
```

Notably, there are two ways to create a disengaged optional object: either by using the default constructor or by calling the 'tagged constructor' that takes `nullopt`. One of these could be safely removed and `optional<T>` could still be initialized to any state.

The default constructor

This proposal provides a default constructor for `optional<T>` that creates a disengaged optional. We find this feature convenient for a couple of reasons. First, it is because this behaviour is intuitive as shown in the above example of function `readChar`. It avoids a certain kind of bugs. Also, it satisfies other expectations. If I declare `optional<T>` as a non-static member, without any initializer, I may expect it is already initialized to the most natural, disengaged, state regardless of whether `T` is `DefaultConstructible` or not. Also when declaring a global object, one could expect that default constructor would be initialized during static-initialization (this proposal guarantees that). One could argue that the tagged constructor could be used for that puropse:

```
optional<int> global = nullopt;

struct X
{
  optional<M> m = nullopt;
};
```

However, sometimes not providing the tag may be the result of an inadvertent omission rather than conscious decision. Because of our default constructor semantics we have to reject the initialization scheme that uses a perfect forwarding constructor. Even if this is fine, one could argue that we do not need a default constructor if we have a tagged constructor. We find this redundancy convenient. For instance, how do you resize a `vector<optional<T>>` if you do not have the default constructor? You could type:

```
vec.resize(size, nullopt);
```

However, that causes first the creation of disengaged optional, and then copying it multiple times. The use of copy constructor may incur run-time overhead and not be available for non-copyable `T`s. Also, it would be not possible to use subscript operator in maps that hold optional objects.

Also, owing to this constructor, `optional` has a nice side-effect feature: it can make "almost `Regular`" types fully `Regular` if the lack of default constructor is the only thing they are missing. For instance consider type `Date` for representing calendar days: it is copyable movable, comparable, but is not `DefaultConstructible` because there is no meaningful default date. However, `optional<Date>` is `Regular` with a meaningful not-a-date state created by default.

Converting constructor (from `T`)

An object of type `T` is convertible to an engaged object of type `optional<T>`:

```
optional<int> oi = 1; // works
```

This convenience feature is not strictly necessary because you can achieve the same effect by using tagged forwarding constructor:

```
optional<int> oi{in_place, 1};
```

If the latter appears too inconvenient, one can always use function `make_optional` described below:

```
optional<int> oi = make_optional(1);
auto oj = make_optional(1);
```

The implicit converting constructor comes in handy in case of optional function arguments:

```
void fun(std::string s, optional<int> oi = nullopt);

fun("dog", 2);
fun("dog");
fun("dog", nullopt); // just to be explicit
```

It has been argued that the constructor from `T` should be made explicit. It is not trivial to decide whether `T` should be convertiblle to `optional<T>`. This is not a clear situation where value of one type is stored in another type with greater resolution, or the situation where the same abstract value is stored in a type with different internal representation. On the other hand, given our conceptual model, `optional<T>` can store all values of `T`, so it is possible to apply a "lossless conversion". We decided to provide the conversion, in order to (1) adhere to our conceptual model, and (2) to enable the above convenience for function argument passing. The implicit conversion naturally implies that `optional<T>`'s can be compared with `T`'s. This is discussed further down.

At some point we considered the possibility to make this constructor conditionally explicit: make it explicit if `T` has an explicit copy/move constructor, and make it non-explicit if `T` has a normal, non-explicit constructor. In the end, we find explicit copy constructor so unusual that we do not find it worthwile to addressing it at the expense of complicating the design.

Contextual conversion to `bool` for checking engaged state

Objections have been risen to this decision. When using `optional<bool>`, contextual conversion to `bool` (used for checking the engaged state) might be confused with accessing the stored value. While such mistake is possible, it is not the first such case in the standard: types `bool*`, `unique_ptr<bool>`, `shared_ptr<bool>` suffer from the same potential problem, and it was never considered a show-stopper. Some have suggested that a special case in the interface should be made for `optional<bool>` specialization. This was however rejected because it would break the generic use of `optional`.

Some have also suggested that a member function like `is_initialized` would more clearly indicate the intent than explicit conversion to `bool`. However, we believe that the latter is a well established idiom in C++ community as well as in the C++ Standard Library, and `optional` appears so fundamental a type that a short and familiar notation appears more appropriate. It also allows us to combine the construction and checking for being engaged in a condition:

```
if (optional<char> ch = readNextChar()) {
  // ...
}
```

Using tag `nullopt` for indicating disengaged state

The proposed interface uses special tag `nullopt` to indicate disengaged `optional` state. It is used for construction, assignment and relational operations. This might rise a couple of objections. First, it introduces redundancy into the interface:

```
optional<int> opt1 = nullopt;
optional<int> opt2 = {};

opt1 = nullopt;
opt2 = {};

if (opt1 == nullopt) ...
if (!opt2) ...
if (opt2 == optional<int>{}) ...
```

On the other hand, there are usages where the usage of `nullopt` cannot be replaced with any other convenient notation:

```
void run(complex<double> v);
void run(optional<string> v);

run(nullopt);              // pick the second overload
run({});                   // ambiguous

if (opt1 == nullopt) ...   // fine
if (opt2 == {}) ...        // illegal

bool is_engaged( optional<int> o)
{
  return bool(o);          // ok, but unclear
  return o != nullopt;     // familiar
}
```

While some situations would work with `{}` syntax, using `nullopt` makes the programmer's intention more clear. Compare these:

```
optional<vector<int>> get1() {
  return {};
}

optional<vector<int>> get2() {
  return nullopt;
}

optional<vector<int>> get3() {
  return optional<vector<int>>{};
```

```
}
```

The usage of `nullopt` is also a consequence of the adapted model for optional: a discriminated union of `T` and `nullopt_t`. Also, a similar redundancy in the interface already exists in a number of components in the standard library: `unique_ptr`, `shared_ptr`, `function` (which use literal `nullptr` for the same purpose); in fact, type requirements `NullablePointer` require of types this redundancy.

Name "nullopt" has been chosen because it clearly indicates that we are interested in creating a null (disengaged) `optional<T>` (of unspecified type `T`). Other short names like "null", "naught", "nothing" or "none" (used in Boost.Optional library) were rejected because they were too generic: they did not indicate unambiguously that it was `optional<T>` that we intend to create. Such a generic tag `nothing` could be useful in many places (e.g., in types like `variant<nothing_t, T, U>`), but is outside the scope of this proposal.

Note also that the definition of tag struct `nullopt` is more complicated than that of other, similar, tags: it has explicitly deleted default constructor. This is in order to enable the reset idiom (`opt2 = {};`), which would otherwise not work because of ambiguuity when deducing the right-hand side argument.

Why not `nullptr`

One could argue that since we have keyword `nullptr`, which already indicates a 'null-state' for a number of Standard Library types, not necessarily pointers (class template `function`), it could be equally well used for `optional`. In fact, the previous revision of this proposal did propose `nullptr`, however there are certain difficulties that arise when the null-pointer literal is used.

First, the interface of `optional` is already criticized for resembling too much the interface of a (raw or smart) pointer, which incorrectly suggests external heap storage and shallow copy and comparison semantics. The "ptr" in "nullptr" would only increase this confusion. While `std::function` is not a pointer either, it also does not provide a confusing `operator->`, or equality comparison, and in case it stores a function pointer it does shallow copying.

Second, using literal `nullptr` in `optional` would make it impossible to provide some of the natural and expected initialization and assignment semantics for types that themselves are nullable:

- `optional<int*>`,
- `optional<const char*>`,
- `optional<M C::*>`,
- `optional<function<void(int)>>`,
- `optional<NullableInteger>`,
- `optional<nullptr_t>`.

Should the following initialization render an engaged or a disengaged optional?

```
optional<int*> op = nullptr;
```

One could argue that if we want to initialize an engaged optional we should indicate that explicitly:

```
optional<int*> op{in_place, nullptr};
```

But this argument would not work in general. One of the goals of the design of `optional` is to allow a seamless "optionalization" of function arguments. That is, given the folowing function signature:

```
void fun(T v) {
  process(v);
}
```

It should be possible to change the signature and the implementation to:

```
void fun(optional<T> v) {
  if (v) process(*v);
  else   doSthElse();
}
```

and expect that all the places that call function `fun` are not affected. But if `T` happens to be `int*` and we occasionally pass value `nullptr` to it, we will silently change the intended behavior of the refactoring: because it will not be the pointer that we null-initialize anymore but a disengaged optional.

Note that this still does not save us from the above problem with refactoring function `fun` in case where `T` happens to be `optional<U>`, but we definately limit the amount of surprises.

In order to avoid similar problems with tag `nullopt`, instantiating template `optional` with types `nullopt_t` and `in_place_t` is prohibited.

There exist, on the other hand, downsides of introducing a special token in place of `nullptr`. The number of ways to indicate the 'null-state' for different library components will grow: you will have `NULL`, `nullptr`, `nullopt`. New C++ programmers will ask "which of these should I use now?" What guidelines should be provided? Use only `nullptr` for pointers? But does it mean that we should use `nullopt` for `std::function`? Having only one way of denoting null-state, would make the things easier, even if "ptr" suggests a pointer.

Why not a tag dependent on `T`?

It has been suggested that instead of 'typeless' `nullopt` a tag nested in class `optional` be used instead:

```
optional<int> oi = optional<int>::nullopt;
```

This has several advantages. Namespace `std` is not polluted with an additional `optional`-specific name. Also, it resolves certain ambiguities when types like `optional<optional<T>>` are involved:

```
optional<optional<int>> ooi = optional<int>::nullopt;           // engaged
optional<optional<int>> ooj = optional<optional<int>>::nullopt; // disengaged

void fun(optional<string>);
void fun(optional<int>);

fun(optional<string>::nullopt); // unambiguous: a typeless nullopt would not do
```

Yet, we choose to propose a typeless tag because we consider the above problems rare and a typeless tag offers a very short notation in other cases:

```
optional<string> fun()
{
  optional<int> oi = nullopt;  // no ambiguity
  oi = nullopt;                // no ambiguity
  // ...
  return nullopt;              // no ambiguity
}
```

If the typeless tag does not work for you, you can always use the following construct, although at the expense of invoking a (possibly elided) move constructor:

```
optional<optional<int>> ooi = optional<int>{};           // engaged
optional<optional<int>> ooj = optional<optional<int>>{}; // disengaged

void fun(optional<string>);
void fun(optional<int>);

fun(optional<string>{}); // unambiguous
```

Accessing the contained value

It was chosen to use indirection operator because, along with explicit conversion to `bool`, it is a very common pattern for accessing a value that might not be there:

```
if (p) use(*p);
```

This pattern is used for all sort of pointers (smart or dumb), and it clearly indicates the fact that the value may be missing and that we return a reference rather than a value. The indirection operator has risen some objections because it may incorrectly imply that `optional` is a (possibly smart) pointer, and thus provides shallow copy and comparison semantics. All library components so far use indirection operator to return an object that is not part of the pointer's/iterator's value. In contrast, `optional` indirects to the part of its own state. We do not consider it a problem in the design; it is more like an unprecedented usage of indirection operator. We believe that the cost of potential confusion is overweighed by the benefit of an easy to grasp and intuitive interface for accessing the contained value.

We do not think that providing an implicit conversion to `T` would be a good choice. First, it would require different way of checking for the empty state; and second, such implicit conversion is not perfect and still requires other means of accessing the contained value if we want to call a member function on it.

Using the indirection operator for a disengaged object is an undefined behavior. This behavior offers maximum runtime performance. In addition to indirection operator, we provide member function `value` that returns a reference to to the contained value if one exists or throws an exception (derived from `logic_error`) otherwise:

```
void interact()
{
  std::string s;
  cout << "enter number ";
  cin >> s;
  optional<int> oi = str2int(s);

  try {
    process_int(oi.value());
  }
  catch(bad_optional_access const&) {
    cout << "this was not a number";
  }
}
```

Relational operators

One of the design goals of `optional` is that objects of type `optional<T>` should be valid elements in STL containers and usable with STL algorithms (at least if objects of type `T` are). Equality comparison is essential for `optional<T>` to model concept `Regular`. C++ does not have concepts, but being regular is still essential for the type to be effectively used with STL. Ordering is essential if we want to store optional values in ordered associative containers. A number of ways of including the disengaged state in comparisons have been suggested. The ones proposed, have been crafted such that the axioms of equivalence and strict weak ordering are preserved: disengaged `optional<T>` is simply treated as an additional and unique value of `T` equal only to itself; this value is always compared as less than any value of `T`:

```
optional<unsigned> o0{0};
optional<unsigned> o1{1};
optional<unsigned> oN{nullopt};

assert (oN < o0);
assert (o0 < o1);
assert (!(oN  < oN));
assert (!(o1 < o1));

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

Value `nullopt` could have been as well considered greater than any value of `T`. The choice is to a great degree arbitrary. We choose to stick to what `boost::optional` does.

Behind the scenes, optional is using utility `std::less<T>` rather than `T::operator<`. This is almost the same thing except for one detail. Quoting 20.8.5 [comparisons] paragraph 8, "For templates `greater`, `less`, `greater_equal`, and `less_equal`, the specializations for any pointer type yield a total order, even if the built-in operators `<`, `>`, `<=`, `>=` do not."

Given that both `nullopt_t` and `T` are implicitly convertible to `optional<T>`, this implies the existence and semantics of mixed comparison between `optional<T>` and `T`, as well as between `optional<T>` and `nullopt_t`:

```
assert (oN == nullopt);
assert (o0 != nullopt);
assert (oN != 1);
assert (o1 == 1);

assert (oN < 1);
assert (o0 > nullopt);
```

Although it is difficult to imagine any practical use case of ordering relation between `optional<T>` and `nullopt_t`, we still provide it for completness's sake

The mixed relational operators, especially these representing order, between `optional<T>` and `T` have been accused of being dangerous. In code examples like the following, it may be unclear if the author did not really intend to compare two `T`'s.

```
auto count = get_optional_count();
if (count < 20) {}                         // or did you mean: *count < 20 ?
if (count == nullopt || *count < 20) {}   // verbose, but unambiguous
```

Given that `optional<T>` is comparable and implicitly constructible from `T`, the mixed comparison is there already. We would have to artificially create the mixed overloads only for them to cause controlled compilation errors. A consistent approach to prohibiting mixed relational operators would be to also prohibit the convesion from `T` or to also prohibit homogenous relational operators for `optional<T>`; we do not want to do either, for other reasons discussed in this proposal. Also, mixed relational operations are available in Boost.Optional and were found useful by the users. Mixed operators come as something natural when we consider the model "`T` with one additional value".

For completeness sake, we also provide ordering relations between `optional<T>` and `nullopt_t`, even though we see no practical use case for them:

```
bool test(optional<int> o)
{
  assert (o >= nullopt);    // regardless of o's state
  assert (!(o < nullopt));  // regardless of o's state
  assert (nullopt <= o);    // regardless of o's state
  return (o > nullopt);     // o != nullopt is cleaner
}
```

This is similar to comparing values of type `unsigned int` with `0`:

```
bool test(unsigned int i)
{
  assert (i >= 0);          // regardless of i's state
  assert (0 <= i);          // regardless of i's state
  return (i > 0);           // i != 0 is cleaner
}
```

There exist two ways of implementing `operator>` for optional objects:

```
bool operator>(const optional<T>& x, const optional<T>& y)
{
  return (!x) ? false : (!y) ? true : greater{}(*x, *y);  // use T::operator>
}

bool operator>(const optional<T>& x, const optional<T>& y)
{
  return y < x;                                 // use optional<T>::operator<
}
```

In case `T::operator>` and `T::operator<` are defined consistently, both above implementations are equivalent. If the two operators are not consistent, the choice of implementation makes a difference. For homogenous relational operations (between two `optional<T>`s), we chose the former specification. This is consistent with a similar choice for `std::tuple`. The only downside of this solution would be visible if some type defined the two predicates inconsistently. We have never seen such a case. The only examples of "clever" abusage of relational operators in domain-speciffic languages we are aware of are expression templates. But in this case operators `<` and `>` do not return type `bool` and we cannot imagine, you would use `optional` for them. On the other hand, there is one small benefit that comes with our proposal: `T::operator>` may not even be defined in order for `optional<T>::operator<` to work.

For heterogenous relational operations (between `optional<T>` and `T`), we also choose to implement all in terms of `std::less`. This way we are consistent with the rest of the standard library. This is a change compared to the previous revisions, where mixed relops were implemented in terms of corresponding relational operators of `T`.

Resetting the optional value

Assigning the value of type `T` to `optional<T>` object results in doing two different things based on whether the optional object is engaged or not. If optional object is engaged, the contained value is assigned a new value. If optional object is disengaged, it becomes engaged using `T`'s copy/move constructor. This behavior is based on a silent assumption that `T`'s copy/move constructor is copying a value in a similar way to copy/move assignment. A similar logic applies to `optional<T>`'s copy/move assignment, although the situation here is more complicated because we have two engaged/disengaged states to be considered. This means that `optional<T>`'s assignment does not work (does not compile) if `T` is not assignable:

```
optional<const int> oi = 1;  // ok
oi = 2;                      // error
oi = oi;                     // error
oi = nullopt;                // ok
```

There is an option to reset the value of optional object without resorting to `T`'s assignment:

```
optional<const int> oj = 1;  // ok
oj.emplace(2);               // ok
```

Function `emplace` disengages the optional object if it is engaged, and then just engages the object anew by copy-constructing the contained value. It is similar to assignment, except that it is guaranteed not to use `T`'s assignment and provides only a basic exception safety guarantee. In contrast, assignment may provide a stronger guarantee if `T`'s assignment does.

To sumarize, this proposal offers three ways of assigning a new contained value to an optional object:

```
optional<int> o;
o = make_optional(1);         // copy/move assignment
o = 1;                        // assignment from T
o.emplace(1);                 // emplacement
```

The first form of assignment is required to make `optional` a regular object, useable in STL. We need the second form in order to reflect the fact that `optional<T>` is a wrapper for `T` and hence it should behave as `T` as much as possible. Also, when `optional<T>` is viewed as `T` with one additional value, we want the values of `T` to be directly assignable to `optional<T>`. In addition, we need the second form to allow the interoperability with function `std::tie` as shown above. The third option is required to be able to reset an optional non-assignable `T`.

Tag `in_place`

This proposal provides an 'in-place' constructor that forwards (perfectly) the arguments provided to `optional`'s constructor into the constructor of `T`. In order to trigger this constructor one has to use the tag struct `in_place`. We need the extra tag to disambiguate certain situations, like calling `optional`'s default constructor and requesting `T`'s default construction:

```
optional<Big> ob{in_place, "1"}; // calls Big{"1"} in place (no moving)
optional<Big> oc{in_place};      // calls Big{} in place (no moving)
optional<Big> od{};             // creates a disengaged optional
```

We no longer propose name `emplace` for the tag. We follow the recomendation from LEWG. Name `emplace` should be reserved in namespace `std` in case an algorithm with that name needs to be added in the future. Also it might be confusing to see a member function and a tag with the same name?

Requirements on `T`

Class template `optional` imposes little requirements on `T`: it has to be either an lvalue reference type, or a complete object type satisfying the requirements of `Destructible`. It is the particular operations on `optional<T>` that impose requirements on `T`: `optional<T>`'s move constructor requires that `T` is `MoveConstructible`, `optional<T>`'s copy constructor requires that `T` is `CopyConstructible`, and so on. This is because `optional<T>` is a wrapper for `T`: it should resemble `T` as much as possible. If `T` is `EqualityComparable` then (and only then) we expect `optional<T>` to be `EqualityComparable`.

Optional references

In this revision, optional references are presented as an auxiliary proposal. The intention is that the Committee should have an option to accept optional values without optional references, if it finds the latter concept inacceptable. Users that in generic contexts require to also store optional lvalue references can achieve this effect, even without direct support for optional references, with a bit of meta-programming.

```
template <class T>
struct generic
{
  typedef T type;
};

template <class U>
struct generic<U&>
{
  typedef std::reference_wrapper<U> type;
};

template <class T>
using Generic = typename generic<T>::type;

template <class X>
void generic_fun()
{
  std::optional<Generic<X>> op;
  // ...
}
```

Although the behavior of such "emulated" optional references will be slightly different than that of "normal" optional references.

Exception specifications

First draft of this revision required an aggressive usage of conditional `noexcept` specifications for nearly every, member- or non-member-, function in the interface. For instance equality comparison was to be declared as:

```
template <class T>
  bool operator==(const optional<T>& lhs, const optional<T>& rhs)
  noexcept(noexcept(*lhs == *rhs));
```

This was based on one of our goals: that we want `optional<T>` to be applicable wherever `T` is applicable in as many situations as reasonably possible. One such situation occurs where no-throw operations of objects of type `T` are used to implement a strong exception safety guarantee of some operations. We would like objects of type `optional<T>` to be also useable in such cases. However, we do not propose this aggressive conditional no-throw guarantees at this time in order for the proposed library component to adhere to the current Library guidelines for conditional `noexcept`: it is currently only used in move constructor, move assignment and `swap`. One exception to this rule, we think could be made for optional's move constructor and assignment from type `T&&`, however we still do not propose this at this time in order to avoid controversy.

Constructors and mutating functions that disengage an optional object are required to be `noexcept(true)`: they only call `T`'s destructor and impose no precondition on optional object's or contained value's state. The same applies to the observers that check the disengaged/engaged state.

The observers that access the contained value — `operator*` and `operator->` — are not declared as `noexcept(true)` even though they have no good reason to throw. This is because they impose a precondition that optional object shall be engaged, and as per observations from N3248[6], library vendors may need to use exceptions to test if the implementation has all the necessary precondition-checking code inside. These observer functions are still required not to throw exceptions.

In general, operations on optional objects only throw, when operations delegated to the contained value throw.

Making `optional` a literal type

We propose that `optional<T>` be a literal type for trivially destructible `T`'s.

```
constexpr optional<int> oi{5};
static_assert(oi, "");            // ok
static_assert(oi != nullopt, ""); // ok
static_assert(oi == oi, "");      // ok
int array[*oi];                   // ok: array of size 5
```

Making `optional<T>` a literal-type in general is impossible: the destructor cannot be trivial because it has to execute an operation that can be conceptually described as:

```
~optional() {
  if (is_engaged()) destroy_contained_value();
}
```

It is still possible to make the destructor trivial for `T`'s which provide a trivial destructor themselves, and we know an efficient implementation of such `optional<T>` with compile-time interface — except for copy constructor and move constructor — is possible. Therefore we propose that for trivially destructible `T`'s all `optional<T>`'s constructors, except for move and copy constructors, as well as observer functions are `constexpr`. The sketch of reference implementation is provided in this proposal.

We need to make a similar exception for `operator->` for types with overloaded `operator&`. The common pattern in the Library is to use function `addressof` to avoid the surprise of overloaded `operator&`. However, we know of no way to implement `constexpr` version of function template `addressof`. The best approach we can take is to require that for normal types the non-overloaded (and `constexpr`) `operator&` is used to take the address of the contained value, and for the tricky types, implementations can use the normal (non-`constexpr`) `addressof`. Similar reasoning applies to operations on optional references in the auxiliary proposal below.

Moved-from state

When a disengaged optional object is moved from (i.e., when it is the source object of move constructor or move assignment) its state does not change. When an engaged object is moved from, we move the contained value, but leave the optional object engaged. A moved-from contained value is still valid (although possibly not specified), so it is fine to consider such optional object engaged. An alternative approach would be to destroy the contained value and make the moved-from optional object disengaged. However, we do not propose this for performance reasons.

In contexts, like returning by value, where you need to call the destructor the second after the move, it does not matter, but in cases where you request the move explicitly and intend to assign a new value in the next step, and if `T` does not provide an efficient move, the chosen approach saves an unnecessary destructor and constructor call:

```
optional<array<Big, 1000>> oo = ... // array doesn't have efficient move
op = std::move(oo);
oo = std::move(tmp);
```

The following is an even more compelling reason. In this proposal `std::optional<int>` is allowed to be implemented as a `TriviallyCopyable` type. Therefore, the copy constructor of type `std::array<std::optional<int>, 1000>` can be implemented using `memcpy`. With the additional requirement that `optional`'s move constructor should not be trivial, we would be preventing the described optimization.

The fact that the moved-from optional is not disengaged may look "uncomfortable" at first, but this is an invalid expectation. The requirements of library components expressed in 17.6.5.15 (moved-from state of library types) only require that moved-from objects are in a valid but unspecified state. We do not need to guarantee anything above this minimum.

The `op = {}` syntax

We put the extra requirements in the standardese to make sure that the following syntax works for resetting the optional:

```
op = {};
```

We consider that this will become a common idiom for resetting (putting into default-constructed state) values in C++. While you get that syntax for free for POD types, in `optional` we have to take extra care to enable it; this is because `optional` provides three assignment operators: copy/move assignment, assignment from `T` and from `nullopt_t`. If we just provided the "intuitive" declaration of assignment from `const T&` and `T&&`, the expression above would become ambiguous. The expression above is processed as:

```
op = DEDUCED{};
```

where `DEDUCED` needs to be deduced from all available overloads. We would have two candidates: move assignment and assignemnt from `T&&`, which would cause an ambiguity. Therefore, we require that the assignment from `T&&` is declared in a more convoluted way — as a template:

```
template <class U> optional& optional<T>::operator=(U&&);
// enable if decay<U> == T
```

The additional requirement that `decay<U> == T` says that the only valid instantiations from this template are these for `const T&` and `T&&` (and some other less relevant variations of references to `T`). But it is still a template, and templates do not participate in the resolution of type `DEDUCED`.

For the same reason, we require that tag `nullopt_t` is not DefaultConstructible. Otherwise, because `optional` provides an assignment from `nullopt_t`, `DEDUCED` might also have been deduced as `nullopt_t`.

Note that it is not the only way to disengage an optional object. You can also use:

```
op = std::nullopt;
```

Requirements for `swap`

For function `swap`, we require that `T` is "swappable for lvalues and `is_move_constructible<T>::value` is `true`". Similarly, we require that it is declared with the following excption specification:

```
noexcept(is_nothrow_move_constructible<T>::value && noexcept(swap(declval<T&>(), declval<T&>()))
```

You can see that we require things of the move constructor but not of move assignment. This is the consequence of the behaviour of function `swap`, which does three different things based on the engagement states of the two optional objects in question. Conceptually, the behaviour could be described as:

```
void swap(optional<T>& lhs, optional<T>& rhs)
{
  if ( lhs &&  rhs) swap(*lhs, *rhs);
  if (!lhs &&  rhs) move-construct in lhs, destroy in rhs;
  if ( lhs && !rhs) move-construct in rhs, destroy in lhs;
  if (!lhs && !rhs) /* no-op */;
}
```

No move-assignment is involved in swap. This is part of the function's contract.

IO operations

The proposed interface for optional values does not contain IO operations: `operator<<` and `operator>>`. While we believe that they would be a useful addition to the interface of optional objects, we also observe that there are some technical obstacles in providing them, and we choose not to propose them at this time. Library components like containers, pairs, tuples face the same issue. At present IO operations are not provided for these types. Our preference for `optional` is to provide an IO solution compatible with this for containers, pairs and tuples, therefore at this point we refrain from proposing a solution for `optional` alone.

Function `value_or`

This function template returns a value stored by the `optional` object if it is engaged, and if not, it falls back to the default value specified in the second argument. It used to be called `get_value_or` in the previous revisions, but we decided to rename it, as a consequence of disscussions, so that it is similar to another new member function `value`. This method for specifying default values on the fly rather than tying the default values to the type is based on the observation that different contexts or usages require different default values for the same type. For instance the default value for `int` can be `0` or `-1`. The callee might not know what value the caller considers special, so it returns the lack of the requested value explicitly. The caller may be better suited to make the choice what special value to use.

```
optional<int> queryDB(std::string);
void setPieceCount(int);
void setMaxCount(int);

setPieceCount( queryDB("select piece_count from ...").value_or(0) );
setMaxCount( queryDB("select max_count from ...").value_or(numeric_limits<int>::max()) );
```

The decision to provide this function is controversial itself. As pointed out by Robert Ramey, the goal of the `optional` is to make the lack of the value explicit. Its syntax forces two control paths; therefore we will typically see an `if`-statement (or similar branching instruction) wherever `optional` is used. This is considered an improvement in correctness. On the other hand, using the default value appears to conflict with the above idea. One other argument against providing it is that in many cases you can use a ternary conditional operator instead:

```
auto&& cnt = queryDB("select piece_count from ...");
setPieceCount(cnt ? *cnt : 0);

auto&& max = queryDB("select max_count from ...");
setMaxCount(max ? std::move(*max) : numeric_limits<int>::max());
```

However, in case optional objects are returned by value and immediately consumed, the ternary operator syntax requires introducing an lvalue. This requires more typing and explicit `move`. This in turn makes the code less safe because a moved-from lvalue is still accessible and open for inadvertent misuse.

There are reasons to make it a free-standing function. (1) It can be implemented by using only the public interface of `optional`. (2) This function template could be equally well be applied to any type satisfying some requirements `NullableProxy`. In this proposal, function `value_or` is defined as a member function and only for `optional`s. Making a premature generalization would risk standardizing a function with suboptimal performance/utility. While we know what detailed semantics (e.g., the return type) `value_or` should have for `optional`, we cannot claim to know the ideal semantics for a generic function. Also, it is not clear to us if this convenience function is equally useful for pointers, as it is for optional objects. By making `value_or` a member function we leave the room for this name in namespace `std` for a possible future generalization.

The second argument in the function template's signature is not `T` but any type convertible to `T`:

```
template <class T, class V>
  typename decay<T>::type optional<T>::value_or(V&& v) const&;
template <class T, class V>
  typename decay<T>::type optional<T>::value_or(V&& v) &&;
```

This allows for a certain run-time optimization. In the following example:

```
optional<string> op{"cat"};
string ans = op.value_or("dog");
```

Because the optional object is engaged, we do not need the fallback value and therefore to convert the string literal `"dog"` into type `string`.

It has been argued that the function should return by constant reference rather than value, which would avoid copy overhead in certain situations:

```
void observe(const X& x);

optional<X> ox { /* ... */ };
observe( ox.value_or(X{args}) );    // unnecessary copy
```

However, the benefit of the function `value_or` is only visible when the optional object is provided as a temporary (without the name); otherwise, a ternary operator is equally useful:

```
optional<X> ox { /* ... */ };
observe(ox ? *ok : X{args});            // no copy
```

Also, returning by reference would be likely to render a dangling reference, in case the optional object is disengaged, because the second argument is typically a temporary:

```
optional<X> ox {nullopt};
auto&& x = ox.value_or(X{args});
cout << x;                              // x is dangling!
```

There is also one practical problem with returning a reference. The function takes two arguments by reference: the optional object and the default value. It can happen that one is deduced as lvalue reference and the other as rvalue reference. In such case we would not know what kind of reference to return. Returning lvalue reference might prevent move optimization; returning an rvalue reference might cause an unsafe move from lvalue. By returning by value we avoid these problems by requiring one unnecessary move in some cases.

We also do not want to return a constant lvalue reference because that would prevent a copy elision in cases where optional object is returned by value.

It has also been suggested (by Luc Danton) that function `optional<T>::value_or<V>` should return type `decay<common_type<T, V>::type>::type` rather than `decay<T>::type`. This would avoid certain problems, such as loss of accuracy on arithmetic types:

```
// not proposed
std::optional<int> op = /* ... */;
long gl = /* ... */;

auto lossless = op.value_or(gl);   // lossless deduced as long rather than int
```

However, we did not find many practical use cases for this extension, so we do not propose it at this time.

Together with function `value`, `value_or` makes a set of similarly called functions for accessing the contained value that do not cause an undefined behavior when invoked on a disengaged optional (at the expense of runtime overhead). They differ though, in the return type: one returns a value, the other a reference.

One other similar convenience function has been suggested. Sometimes the default value is not given, and computing it takes some time. We only want to compute it, when we know the optional object is disengaged:

```
optional<int> oi = /* ... */;

if (oi) {
  use(*oi);
}
else {
  int i = painfully_compute_default();
  use(i);
}
```

The solution to that situation would be another convenience function which rather than taking a default value takes a callable object that is capable of computing a default value if needed:

```
use( oi.value_or_call(&painfully_compute_default) );
// or
use( oi.value_or_call([&]{return painfully_compute_default();} );
```

We do not propose this, as we prefer to standardize the existing practice. Also, it is not clear how often the above situations may occur, and the tool prove useful.

Function `make_optional`

We also propose a helper function `make_optional`. Its semantics is closer to that of `make_pair` or `make_tuple` than that of `make_shared`. You can use it in order for the type of the optional to be deduced:

```
int i = 1;
auto oi = make_optional(i);          // decltype(oi) == optional<int>
```

This may occasionally be useful when you need to pick the right overload and not type the type of the optional by hand:

```
void fun(optional<complex<double>>);
void fun(Current);                   // complex is convertible to Current

complex<double> c{0.0, 0.1};
fun(c);                              // ambiguous
fun({c});                            // ambiguous
fun(make_optional(c));               // picks first overload
```

This is not very useful in return statements, as long as the converting constructor from `T` is implicit, because you can always use the brace syntax:

```
optional<complex<double>> findC()
{
  complex<double> c{0.0, 0.1};
  return {c};
}
```

`make_shared`-like function does not appear to be useful at all: it is no different than manually creating a temporary optional object:

```
// not proposed
fun( make_optional<Rational>(1, 2) );
fun( optional<Rational>{1, 2} );     // same as above
```

It would also not be a good alternative for tagged placement constructor, because using it would require type `T` to be `MoveConstructible`:

```
// not proposed
auto og = make_optional<Guard>("arg1"); // ERROR: Guard is not MoveConstructible
```

Such solution works for `shared_ptr` only because its copy constructor is shallow. One useful variant of `shared_ptr`-like `make_optional` would be a function that either creates an engaged or a disengaged optional based on some boolean condition:

```
// not proposed
return make_optional_if<Rational>(good(i) && not_bad(j), i, j);

// same as:
if (good(i) && not_bad(j)) {
  return {i, j};
}
else {
  return nullopt;
}

// same as:
optional<Rational> or = nullopt;
if (good(i) && not_bad(j)) or.emplace(i, j);
return or; // move-construct on return
```

Since this use case is rare, and the function call not that elegant, and a two-liner alternative exists, we do not propose it.

Header `<utility>` or `<optional>`?

Should `optional` be packed into header file `<utility>` or does it deserve its own header? The answer to this question is arbitrary. To great extent (where we believed it made sense) we try to follow what `tuple` does. The latter has its own header. Optional is pretty much a "stand-alone" library with vast usage. Also, we do not want to overload `<utility>` too much. There is no problem, on the other hand, with putting `optional` to `<utility>`, if this is the feedback from LWG.

Handling `initializer_list`

Another feature worth considering is a "sequence constructor" (one that takes `initializer_list` as its argument). It would be enabled (in `enable_if` sense) only for these `T`s that themself provide a sequence constructor. This would be useful to fully support two features we already mentioned above (but chose not to propose).

First, our goal of "copy initialization forwarding" for `optional` also needs to address the following usages of `initializer_list`:

```
vector<int> v = {1, 2, 4, 8};
optional<vector<int>> ov = {1, 2, 4, 8};

assert (v == *ov);
```

This is not only a syntactical convenience. It also avoids subtle bugs. When perfect forwarding constructor is implemented naively with one variadic constructor, optional vector initialization may render surprising result:

```
optional<vector<int>> ov = {3, 1};

assert (*ov == vector{3, 1});    // FAILS!
assert (*ov == vector{1, 1, 1}); // TRUE!
```

However this sequence constructor feature is incompatible with another one: default constructor creating a disengaged optional. This is because, as outlined in the former example, initializer `{}`, that looks like 0-element list, is in fact interpreted as the request for value-initialization (default constructor call). This may hit programmers that use initializer list in "generic" context:

```
template <class ...A> // enable_if: every A is int
void fwd(const A&&... a)
{
  optional<vector<int>> o = {a...};
  assert (bool(o)); // not true for empty a
}
```

If this feature were to be added, we would need to provide an assignment from initializer list and variadic 'emplacement' constructor with the first forwarded argument being `initializer_list`:

```
ov = {1, 2, 4, 8};

allocator<int> a;
optional<vector<int>> ou { in_place, {1, 2, 4, 8}, a };

assert (ou == ov);
```

Since we are not proposing neither perfect forwarding constructor, nor the "copy initialization forwarding", we are also not proposing the sequence constructor. However, in this proposal, the following constructs work:

```
optional<vector<int>> ov{in_place, {3, 1}};
assert (*ov == vector{3, 1});

ov.emplace({3, 1});
assert (*ov == vector{3, 1});
```

`optional<optional<T>>`

The necessity to create a "double" optional explicitly does not occur often. Such type may appear though in generic contexts where we create `optional<V>` and `V` only happens to be `optional<T>`. Some special behavior to be observed in this situation is the following. When copy-initializing with `nullopt`, the "outermost" optional is initialized to disengaged state. Thus, changing function argument from `optional<T>` to `optional<optional<T>>` will silently break the code in places where the argument passed to function happens to be of type `nullopt_t`:

```
// before change
void fun(optional<T> v) {
  process(v);
}

fun(nullopt); // process() called

// after change
void fun(optional<optional<T>> v) {
  if (v) process(*v);
  else   doSthElse();
}

fun(nullopt); // process() not called!
```

This issue would not arise if `nullopt` were `T`-specific:

```
fun(optional<T>::nullopt);            // process() called
fun(optional<optional<T>>::nullopt);  // process() not called
```

Since `T`-dependent `nullopt` is not proposed, in order to create an engaged optional containing a disengaged optional, one needs to use one of the following constructs:

```
optional<optional<T>> ot {in_place};
optional<optional<T>> ou {in_place, nullopt};
optional<optional<T>> ov {optional<T>{}};
```

Also note that `make_optional` will create a "double" optional when called with optional argument:

```
optional<int> oi;
auto ooi = make_optional(oi);
static_assert( is_same<optional<optional<int>>, decltype(ooi)>::value, "");
```

Open questions

Allocator support

Optional does not allocate memory. So it can do without allocators. However, it can be useful in compound types like:

```
typedef vector< optional<vector<int, MyAlloc>>, MyAlloc>; MyVec;
MyVec v{ v2, MyAlloc{} };
```

One could expect that the allocator argument is forwarded in this constructor call to the nested vectors that use the same allocator. Allocator support would enable this. `std::tuple` offers this functionality.

Proposed wording

After subclause 17.3.10 ([defns.default.behavior.func]), insert a new subclause. (Subclause [defns.handler] becomes 17.3.12.)

17.3.11 **[defns.direct-non-list-init]**

dierect-non-list-initialization

A direct-initialization that is not list-initialization.

Add new header in Table 14 (C++ library headers).

Table 14 — C++ library headers

`<algorithm>`	`<fstream>`	`<list>`	`<ratio>`	`<tuple>`
`<array>`	`<functional>`	`<locale>`	`<regex>`	`<typeindex>`
`<atomic>`	`<future>`	`<map>`	`<set>`	`<typeinfo>`
`<bitset>`	`<initializer_list>`	`<memory>`	`<sstream>`	`<type_traits>`
`<chrono>`	`<iomanip>`	`<mutex>`	`<stack>`	`<unordered_map>`
`<codecvt>`	`<ios>`	`<new>`	`<stdexcept>`	`<unordered_set>`
`<complex>`	`<iosfwd>`	`<numeric>`	`<streambuf>`	`<utility>`
`<condition_variable>`	`<iostream>`	`<optional>`	`<string>`	`<valarray>`
`<dequeue>`	`<istream>`	`<ostream>`	`<strstream>`	`<vector>`
`<exception>`	`<iterator>`	`<queue>`	`<system_error>`	
`<forward_list>`	`<limits>`	`<random>`	`<thread>`	

After chapter 20.4 Tuples [tuple], insert a new paragraph. (Chapter [template.bitset] (Class template `bitset`) becomes 20.6.)

20.5 Optional objects [optional]

20.5.1 In general [optional.general]

This subclause describes class template `optional` that represents *optional objects*. An *optional object for object types* is an object that contains the storage for another object and manages the lifetime of this contained object, if any. The contained object may be initialized after the optional object has been initialized, and may be destroyed before the optional object has been destroyed. The initialization state of the contained object is tracked by the optional object.

20.5.2 Header `<optional>` synopsis [optional.synop]

```
namespace std {
  // 20.5.4, optional for object types
  template <class T> class optional;

  // 20.5.5, In-place construction
  struct in_place_t{};
  constexpr in_place_t in_place{};

  // 20.5.6, Disengaged state indicator
  struct nullopt_t{see below};
  constexpr nullopt_t nullopt(unspecified);

  // 20.5.7, class bad_optional_access
  class bad_optional_access;

  // 20.5.8, Relational operators
  template <class T>
    constexpr bool operator==(const optional<T>&, const optional<T>&);
  template <class T>
    constexpr bool operator<(const optional<T>&, const optional<T>&);

  // 20.5.9, Comparison with nullopt
  template <class T> constexpr bool operator==(const optional<T>&, nullopt_t) noexcept;
  template <class T> constexpr bool operator==(nullopt_t, const optional<T>&) noexcept;
  template <class T> constexpr bool operator<(const optional<T>&, nullopt_t) noexcept;
  template <class T> constexpr bool operator<(nullopt_t, const optional<T>&) noexcept;

  // 20.5.10, Comparison with T
  template <class T> constexpr bool operator==(const optional<T>&, const T&);
  template <class T> constexpr bool operator==(const T&, const optional<T>&);
  template <class T> constexpr bool operator<(const optional<T>&, const T&);
```

```
  template <class T> constexpr bool operator<(const T&, const optional<T>&);

  // 20.5.11, Specialized algorithms
  template <class T> void swap(optional<T>&, optional<T>&) noexcept(see below);
  template <class T> constexpr optional<see below> make_optional(T&&);

  // 20.5.12, hash support
  template <class T> struct hash;
  template <class T> struct hash<optional<T>>;
} // namespace std
```

A program that necessitates the instantiation of template `optional` for a reference type, or for possibly cv-qualified types `in_place_t` or `nullopt_t` is ill-formed.

20.5.3 Definitions [optional.defs]

An instance of `optional<T>` is said to be *disengaged* if it has been default constructed, constructed with or assigned with a value of type `nullopt_t`, constructed with or assigned with a disengaged optional object of type `optional<T>`.

An instance of `optional<T>` is said to be *engaged* if it is not disengaged.

20.5.4 `optional` for object types [optional.object]

```
namespace std {

  template <class T>
  class optional
  {
  public:
    typedef T value_type;

    // 20.5.4.1, constructors
    constexpr optional() noexcept;
    constexpr optional(nullopt_t) noexcept;
    optional(const optional&);
    optional(optional&&) noexcept(see below);
    constexpr optional(const T&);
    constexpr optional(T&&);
    template <class... Args> constexpr explicit optional(in_place_t, Args&&...);
    template <class U, class... Args>
      constexpr explicit optional(in_place_t, initializer_list<U>, Args&&...);

    // 20.5.4.2, destructor
    ~optional();

    // 20.5.4.3, assignment
    optional& operator=(nullopt_t) noexcept;
    optional& operator=(const optional&);
    optional& operator=(optional&&) noexcept(see below);
    template <class U> optional& operator=(U&&);
    template <class... Args> void emplace(Args&&...);
    template <class U, class... Args>
      void emplace(initializer_list<U>, Args&&...);

    // 20.5.4.4, swap
    void swap(optional&) noexcept(see below);

    // 20.5.4.5, observers
    constexpr T const* operator ->() const;
    T* operator ->();
    constexpr T const& operator *() const;
    T& operator *();
    constexpr explicit operator bool() const noexcept;
    constexpr T const& value() const;
    T& value();
    template <class U> constexpr T value_or(U&&) const&;
```

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

  private:
    bool init; // exposition only
    T*   val;  // exposition only
  };

} // namespace std
```

Engaged instances of `optional<T>` where `T` is of object type shall contain a value of type `T` within its own storage. This value is referred to as the *contained value* of the optional object. Implementations are not permitted to use additional storage, such as dynamic memory, to allocate its contained value. The contained value shall be allocated in a region of the `optional<T>` storage suitably aligned for the type `T`.

Members *`init`* and *`val`* are provided for exposition only. Implementations need not provide those members. *`init`* indicates whether the `optional` object's contained value has been initialized (and not yet destroyed); *`val`* points to (a possibly uninitialized) contained value.

`T` shall be an object type and shall satisfy the requirements of `Destructible` (Table 24).

20.5.4.1 Constructors **[optional.object.ctor]**

```
constexpr optional<T>::optional() noexcept;
constexpr optional<T>::optional(nullopt_t) noexcept;
```

Postconditions: `*this` is disengaged.

Remarks: No `T` object referenced is initialized. For every object type `T` these constructors shall be `constexpr` constructors (7.1.5).

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

Requires: `is_copy_constructible<T>::value` is `true`.

Effects: If *`rhs`* is engaged initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `*`*`rhs`*.

Postconditions: `bool(`*`rhs`*`) == bool(*this)`.

Throws: Any exception thrown by the selected constructor of `T`.

```
optional<T>::optional(optional<T> && rhs) noexcept(see below);
```

Requires: `is_move_constructible<T>::value` is `true`.

Effects: If *`rhs`* is engaged initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::move(*`*`rhs`*`)`. `bool(`*`rhs`*`)` is unchanged.

Postconditions: `bool(`*`rhs`*`) == bool(*this)`.

Throws: Any exception thrown by the selected constructor of `T`.

Remarks: The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_constructible<T>::value
```

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

Requires: `is_copy_constructible<T>::value` is `true`.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression *`v`*.

Postconditions: `*this` is engaged.

Throws: Any exception thrown by the selected constructor of `T`.

Remarks: If `T`'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

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

Requires: `is_move_constructible<T>::value` is `true`.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::move(v)`.

Postconditions: `*this` is engaged.

Throws: Any exception thrown by the selected constructor of `T`.

Remarks: If `T`'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

```
template <class... Args> constexpr explicit optional(in_place_t, Args&&... args);
```

Requires: `is_constructible<T, Args&&...>::value` is `true`.

Effects: Initializes the contained value as if constructing an object of type `T` with the arguments `std::forward<Args>(args)...`.

Postconditions: `*this` is engaged.

Throws: Any exception thrown by the selected constructor of `T`.

Remarks: If `T`'s constructor selected for the initialization is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

```
template <class U, class... Args>
explicit optional(in_place_t, initializer_list<U> il, Args&&... args);
```

Requires: `is_constructible<T, initializer_list<U>&, Args&&...>::value` is `true`.

Effects: Initializes the contained value as if constructing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.

Postconditions: `*this` is engaged.

Throws: Any exception thrown by the selected constructor of `T`.

Remarks: The function shall not participate in overload resolution unless `is_constructible<T, initializer_list<U>&, Args&&...>::value` is `true`.

Remarks: If `T`'s constructor selected for the initialization is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

20.5.4.2 Destructor [optional.object.dtor]

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

Effects: If `is_trivially_destructible<T>::value != true` and `*this` is engaged, calls `val->T::~T()`.

Remarks: If `is_trivially_destructible<T>::value == true` then this destructor shall be a trivial destructor.

20.5.4.3 Assignment [optional.object.assign]

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

Effects: If `*this` is engaged calls `val->T::~T()` to destroy the *contained value*; otherwise no effect.

Returns: `*this`.

Postconditions: `*this` is disengaged.

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

Requires: `is_copy_constructible<T>::value` is `true` and `is_copy_assignable<T>::value` is `true`.

Effects:

- If `*this` is disengaged and *rhs* is disengaged, no effect, otherwise
- if `*this` is engaged and *rhs* is disengaged, destroys the contained value by calling `val->T::~T()`, otherwise
- if `*this` is disengaged and *rhs* is engaged, initializes the contained value as if direct-non-list-initializing an object of type `T` with `*rhs`, otherwise
- (if both `*this` and *rhs* are engaged) assigns `*rhs` to the contained value.

Returns:

`*this`.

Postconditions: `bool(rhs) == bool(*this)`.

Exception Safety: If any exception is thrown, values of *init* and *rhs.init* remain 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 constructor.

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

Requires: `is_move_constructible<T>::value` is `true` and `is_move_assignable<T>::value` is `true`.

Effects:

- If `*this` is disengaged and *rhs* is disengaged, no effect, otherwise
- if `*this` is engaged and *rhs* is disengaged, destroys the contained value by calling `val->T::~T()`, otherwise
- if `*this` is disengaged and *rhs* is engaged, initializes the contained value as if direct-non-list-initializing an object of type `T` with `std::move(*rhs)`, otherwise
- (if both `*this` and *rhs* are engaged) 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_assignable<T>::value && is_nothrow_move_constructible<T>::value
```

Exception Safety: If any exception is thrown, values of *init* and *rhs.init* remain unchanged. If an exception is thrown during the call to `T`'s move constructor, the state of `*rhs.val` is determined by exception safety guarantee of `T`'s move constructor. If an exception is thrown during the call to `T`'s move assignment, the state of `*val` and `*rhs.val` is determined by exception safety guarantee of `T`'s move assignment.

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

Requires: `is_constructible<T, U>::value` is `true` and `is_assignable<U, T>::value` is true.

Effects: If `*this` is engaged assigns `std::forward<U>(v)` to the contained value; otherwise initializes the contained value as if direct-non-list-initializing object of type `T` with `std::forward<U>(v)`.

Returns: `*this`.

Postconditions: `*this` is engaged.

Exception Safety: If any exception is thrown, value of *`init`* remains unchanged. If an exception is thrown during the call to `T`'s constructor, the state of *v* is determined by exception safety guarantee of `T`'s constructor. If an exception is thrown during the call to `T`'s assignment, the state of `*val` and *v* is determined by exception safety guarantee of `T`'s assignment.

Remarks: The function shall not participate in overload resolution unless `is_same<typename remove_reference<U>::type, T>::value` is `true`.

[*Note:* The reason to provide such generic assignment and then constraining it so that effectively `T == U` is to guarantee that assignment of the form `o = {}` is unambiguous. —*end note*]

`template <class... Args> void optional<T>::emplace(Args&&... args);`

Requires: `is_constructible<T, Args&&...>::value` is `true`.

Effects: Calls `*this = nullopt`. Then initializes the contained value as if constructing an object of type `T` with the arguments `std::forward<Args>(args)...`.

Postconditions: `*this` is engaged.

Throws: Any exception thrown by the selected constructor of `T`.

Exception Safety: If an exception is thrown during the call to `T`'s constructor, `*this` is disengaged, and the previous `*val` (if any) has been destroyed.

`template <class U, class... Args> void optional<T>::emplace(initializer_list<U> il, Args&&... args);`

Requires: `is_constructible<T, initializer_list<U>&, Args&&...>::value` is `true`.

Effects: Calls `*this = nullopt`. Then initializes the contained value as if constructing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.

Postconditions: `*this` is engaged.

Throws: Any exception thrown by the selected constructor of `T`.

Exception Safety: If an exception is thrown during the call to `T`'s constructor, `*this` is disengaged, and the previous `*val` (if any) has been destroyed.

Remarks: The function shall not participate in overload resolution unless `is_constructible<T, initializer_list<U>&, Args&&...>::value` is `true`.

20.5.4.4 Swap [optional.object.swap]

`void optional<T>::swap(optional<T>& rhs) noexcept(see below);`

Requires: LValues of type T shall be swappable and `is_move_constructible<T>::value` is `true`.

Effects:

- If `*this` is disengaged and *rhs* is disengaged, no effect, otherwise
- if `*this` is engaged and *rhs* is disengaged, initializes the contained value of *rhs* by direct-initialization with `std::move(*(*this))`, followed by `val->T::~T()`, `swap(init, rhs.init)`, otherwise
- if `*this` is disengaged and *rhs* is engaged, initializes the contained value of `*this` by direct-initialization with `std::move(*rhs)`, followed by `rhs.val->T::~T()`, `swap(init, rhs.init)`, otherwise
- (if both `*this` and *rhs* are engaged) calls `swap(*(*this), *rhs)`.

Throws:
Any exceptions that the expressions in the Effects clause throw.

Remarks: The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_constructible<T>::value && noexcept(swap(declval<T&>(), declval<T&>(
```

Exception Safety: If any exception is thrown, values of *init* and *rhs.init* remain unchanged. If an exception is thrown during the call to function `swap` the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `swap` for lvalues of `T`. If an exception is thrown during the call to `T`'s move constructor, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `T`'s move constructor.

20.5.4.5 Observers [optional.object.observe]

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

Requires: `*this` is engaged.

Returns: *val*.

Throws: Nothing.

Remarks: Unless `T` is a user-defined type with overloaded unary `operator&`, the first function shall be a `constexpr` function.

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

Requires: `*this` is engaged.

Returns: `*val`.

Throws: Nothing.

Remarks: The first function shall be a `constexpr` function.

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

Returns: *init*.

Remarks: this function shall be a `constexpr` function.

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

Returns: `*val`, if `bool(*this)`.

Throws: `bad_optional_access` if `!*this`.

Remarks: The first function shall be a `constexpr` function.

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

Requires: `is_copy_constructible<T>::value` is `true` and `is_convertible<U&&, T>::value` is `true`.

Returns: `bool(*this) ? **this : static_cast<T>(std::forward<U>(v))`.

Throws: Any exception thrown by the selected constructor of `T`.

Exception Safety: If *`init`* `== true` and exception is thrown during the call to `T`'s constructor, the value of *`init`* and *v* remains unchanged and the state of *`*val`* is determined by the exception safety guarantee of the selected constructor of `T`. Otherwise, when exception is thrown during the call to `T`'s constructor, the value of *`*this`* remains unchanged and the state of *v* is determined by the exception safety guarantee of the selected constructor of `T`.

Remarks: If the selected constructor of `T` is a `constexpr` constructor, this function shall be a `constexpr` function.

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

Requires: `is_move_constructible<T>::value` is `true` and `is_convertible<U&&, T>::value` is `true`.

Returns: `bool(*this) ? std::move(**this) : static_cast<T>(std::forward<U>(v))`.

Throws: Any exception thrown by the selected constructor of `T`.

Exception Safety: If *`init`* `== true` and exception is thrown during the call to `T`'s constructor, the value of *`init`* and *v* remains unchanged and the state of *`*val`* is determined by the exception safety guarantee of the `T`'s constructor. Otherwise, when exception is thrown during the call to `T`'s constructor, the value of *`*this`* remains unchanged and the state of *v* is determined by the exception safety guarantee of the selected constructor of `T`.

20.5.5 In-place construction [optional.inplace]

```
struct in_place_t{};
constexpr in_place_t in_place{};
```

The struct `in_place_t` is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, `optional<T>` has a constructor with `in_place_t` as the first argument followed by an argument pack; this indicates that `T` should be constructed in-place (as if by a call to placement new expression) with the forwarded argument pack as parameters.

20.5.6 Disengaged state indicator [optional.nullopt]

```
struct nullopt_t{see below};
constexpr nullopt_t nullopt(unspecified);
```

The struct `nullopt_t` is an empty structure type used as a unique type to indicate a disengaged state for `optional` objects. In particular, `optional<T>` has a constructor with `nullopt_t` as single argument; this indicates that a disengaged optional object shall be constructed.

Type `nullopt_t` shall not have a default constructor. It shall be a literal type. Constant `nullopt` shall be initialized with an argument of literal type.

20.5.7 Class `bad_optional_access` [optional.bad_optional_access]

```
namespace std {
```

```
      class bad_optional_access : public logic_error {
      public:
        explicit bad_optional_access(const string& what_arg);
        explicit bad_optional_access(const char* what_arg);
      };
    }
```

The class `bad_optional_access` defines the type of objects thrown as exceptions to report the situation where an attempt is made to access the value of a disengaged optional object.

`bad_optional_access(const string& what_arg);`

Effects: Constructs an object of class `bad_optional_access`.

Postcondition: `strcmp(what(), what_arg.c_str()) == 0`.

`bad_optional_access(const char* what_arg);`

Effects: Constructs an object of class `bad_optional_access`.

Postcondition: `strcmp(what(), what_arg) == 0`.

20.5.8 Relational operators [optional.relops]

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

Requires: `T` shall meet the requirements of `EqualityComparable`.

Returns: If `bool(x) != bool(y)`, `false`; otherwise if `bool(x) == false`, `true`; otherwise `*x == *y`.

Remarks: Instantiations of this function template for which `*x == *y` is a core constant expression, shall be `constexpr` functions.

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

Requires: Expression `less<T>{}(*x, *y)` shall be well-formed.

Returns: If `(!y)`, `false`; otherwise, if `(!x)`, `true`; otherwise `less<T>{}(*x, *y)`.

Remarks: Instantiations of this function template for which `less<T>{}(*x, *y)` is a core constant expression, shall be `constexpr` functions.

20.5.9 Comparison with `nullopt` [optional.nullops]

`template <class T> constexpr bool operator==(const optional<T>& x, nullopt_t) noexcept;`
`template <class T> constexpr bool operator==(nullopt_t, const optional<T>& x) noexcept;`

Returns: `(!x)`.

`template <class T> constexpr bool operator<(const optional<T>& x, nullopt_t) noexcept;`

Returns: `false`.

`template <class T> constexpr bool operator<(nullopt_t, const optional<T>& x) noexcept;`

Returns: `bool(x)`.

20.5.10 Comparison with `T` [optional.comp_with_t]

```
template <class T> constexpr bool operator==(const optional<T>& x, const T& v);
```

Returns: `bool(x) ? *x == v : false`.

```
template <class T> constexpr bool operator==(const T& v, const optional<T>& x);
```

Returns: `bool(x) ? v == *x : false`.

```
template <class T> constexpr bool operator<(const optional<T>& x, const T& v);
```

Returns: `bool(x) ? less<T>{}(*x, v) : true`.

20.5.11 Specialized algorithms [optional.specalg]

```
template <class T> void swap(optional<T>& x, optional<T>& y)
noexcept(noexcept(x.swap(y)));
```

Effects: calls `x.swap(y)`.

```
template <class T>
  constexpr optional<typename decay<T>::type> make_optional(T&& v);
```

Returns: `optional<typename decay<T>::type>(std::forward<T>(v))`.

20.5.12 Hash support [optional.hash]

```
template <class T> struct hash<optional<T>>;
```

Requires: the template specilaization `hash<T>` shall meet the requirements of class template `hash` (20.9.12). The template specilaization `hash<optional<T>>` shall meet the requirements of class template `hash`. For an object `o` of type `optional<T>`, if `bool(o) == true`, `hash<optional<T>>()(o)` shall evaluate to the same value as `hash<T>()(*o)`.

Auxiliary proposal — optional references

We propose optional references as an auxiliary proposal. This is to give the Committee the freedom to make the decision to accept or not optional references independently of the decision to accept optional values.

Overview

Optional references are surprising to many people because they do not appear to add any more functionality than pointers do. There exist though a couple of arguments in favour optional references:

- `optional<T>` can be used in generic code, were `T` can be either a reference or an object.
- It is slightly easier to pass arguments to functions, beacause addressof operator is not required.
- Raw pointers historically add confusion in the sense that it is not clear whether we should delete the value they point to or not, as well as whether we should expect `nullptr` or not. With optional references the answer to these questions is obvious.

The interface for optional references is somewhat limited compared to that for optional values.

```
int i = 1;
int j = 2;
optional<int&> ora;                 // disengaged optional reference to int
optional<int&> orb = i;             // contained reference refers to object i

*orb = 3;                           // i becomes 3
ora = j;                            // ERROR: optional refs do not have assignment from T
```

```
ora = optional<int&>{j};           // contained reference refers to object j
ora = {j};                         // same as line above
orb = ora;                         // rebinds orb to refers to the same object as ora
*orb = 7;                          // j becomes 7
ora.emplace(j);                    // OK: contained reference refers to object j
ora.emplace(i);                    // OK: contained reference now refers to object i
ora = nullopt;                     // OK: ora becomes disengaged
```

In some aspects optional lvalue references act like raw pointers: they are rebindable, may become disengaged, and do not have special powers (such as extending the life-time of a temporary). In other aspects they act like C++ references: they do not provide pointer arithmetic, operations like comparisons, hashing are performed on referenced objects.

```
hash<int> hi;
hash<optional<int&>> hoi;

int i = 0;
int j = 0;
optional<int&> ori = i;            // orj and ori refer to two different objects
optional<int&> orj = j;            // but with the same value

assert (hoi(ori) == hi(i));
assert (hoi(ori) == hoi(orj));     // because we are hashing the values
assert (ori == orj);               // because we are comparing the values
```

Because optional references are easily implementable with a raw pointer, we require that almost no operation on optional references (except `value` and `value_or`) throws exceptions.

Optional references do not popagate constness to the object they indirect to:

```
int i = 9;
const optional<int&> mi = i;
int& r = *mi;                      // OK: decltype(*mi) == int&

optional<const int&> ci = i;
int& q = *ci;                      // ERROR: decltype(*ci) == const int&
```

Note also the peculiar way in which function `make_optional` can create optional references:

```
int i = 1;
auto oi = make_optional(i);          // decltype(oi) == optional<int>
auto ri = make_optional(ref(i));     // decltype(ri) == optional<int&>
```

Practical use cases

While used significantly less often, optional references are useful in certain cases. For instance, consider that you need to find a possibly missing element in some sort of a collection, and if it is found, change it. The following is a possible implementation of such function that will help with the find.

```
optional<int&> find_biggest( vector<int>& vec )
{
  optional<int&> biggest;
  for (int & val : vec) {
    if (!biggest || *biggest < val) {
      biggest.emplace(val);
    }
  }
  return biggest;
}
```

This could be alternatively implemented using a raw pointer; however, optional reference makes it clear that the caller will not be owing the object. For another example, consider that we want a function to modify the passed object (`storeHere` in the example) as one of its responsibilities, but we sometimes cannot provide the object, but we do want the function to perform other responsibilities. The following is the possible implementation

```
template <typename T>
```

```
T getValue( optional<T> newVal = nullopt, optional<T&> storeHere = nullopt )
{
  if (newVal) {
    cached = *newVal;

    if (storeHere) {
      *storeHere = *newVal; // LEGAL: assigning T to T
    }
  }
  return cached;
}
```

Again, this can also be implemented with pointers; however, with optional references the ownership of the memory is clear. Additionally, we do not have to use the indirection operator:

```
static int global = 0;
const int newVal = 2;
return getValue(newVal, global);
```

Assignment for optional references

The semantics of optional references' copy assignment turned out to be very controversial. This is because whatever semantics for such assignment is chosen, it is confusing to many programmers. An optional reference can be seen as a reference with postponed initialization. In this case, assignment (to engaged optional reference) is expected to have deep copy semantics: it should assign value to the referred object. On the other hand, an optional reference can be seen as a pointer with different syntax. In this case the assignment (to engaged optional reference) should change the reference, so that it refers to the new object. Neither of these models appears more valid than the other. On the other hand, the majority of people insist that optional should be copy-assignable. We choose somewhat arbitralrily to provide a rebinding semantics for `std::optional`. This is to follow the practice adapted by `std::reference_wrapper` and `boost::optional`. In consequence, optional references are not value semantic types: `operator==` compares something else than copy assignment and constructor are copying. This should not be surprising, though, for a component that is called a "reference".

The other semantics can be implemented by using the following idiom:

```
void assign_norebind(optional<T&>& optref, T& obj)
{
  if (optref) *optref = obj;
  else        optref.emplace(obj);
}
```

No rvalue binding for optional references

Optional referencs cannot provide an essential feature of native references: extending the life-time of temporaries (rvalues). Temporaries can be bound to (1) rvalue references and to (2) lvalue references to const. In order to avoid dangling reference problems we need to prevent either type of binfing to optional references. In order to prevent the former, we disallow optional rvalue references altogether. We are not aware of any practical use case for such entities. Since optional lvalue references to const appear useful, we avoid the rvalue binding problem by requiring implementations to "poison" rvalue reference constructors for optional lvalue references to const. This may appear surprising as it is inconsistent with normal (non-optional) reference behavior:

```
const int& nr = int{1};           // ok
optional<const int&> or = int{1}; // error: int&& ctor deleted
```

Implementability

This proposal can be implemented as pure library extension, without any compiler magic support, in C++11. An almost full rerefence implementation of this proposal can be found at https://github.com/akrzemi1/Optional/. Below we demonstrate how one can implement `optional`'s `constexpr` constructors to engaged and disengaged state as well as `constexpr operator*` for `TriviallyDestructible` `T`'s.

```
namespace std {

#if defined NDEBUG
# define ASSERTED_EXPRESSION(CHECK, EXPR) (EXPR)
#else
# define ASSERTED_EXPRESSION(CHECK, EXPR) ((CHECK) ? (EXPR) : (fail(#CHECK, __FILE__, __LINE__)
  inline void fail(const char* expr, const char* file, unsigned line) { /*...*/ }
#endif

struct dummy_t{};

template <class T>
union optional_storage
{
  static_assert( is_trivially_destructible<T>::value, "" );

  dummy_t dummy_;
  T       value_;

  constexpr optional_storage()            // null-state ctor
    : dummy_{} {}

  constexpr optional_storage(T const& v)  // value ctor
    : value_{v} {}

  ~optional_storage() = default;          // trivial dtor
};


template <class T>
// requires: is_trivially_destructible<T>::value
class optional
{
  bool initialized_;
  optional_storage<T> storage_;

public:
  constexpr optional(nullopt_t) : initialized_{false}, storage_{} {}

  constexpr optional(T const& v) : initialized_{true}, storage_{v} {}

  constexpr T const& operator*()
  {
    return ASSERTED_EXPRESSION(bool(*this), storage_.value_);
  }

  constexpr T const& value()
  {
    return *this ? storage_.value_ : (throw bad_optional_access(""), storage_.value_);
  }

  // ...
};

} // namespace std
```


Acknowledgements

Many people from the Boost community, participated in the developement of the Boost.Optional library. Sebastian Redl suggested the usage of function `emplace`.

Daniel Krügler provided numerous helpful suggestions, corrections and comments on this paper; in particular he suggested the addition of and reference implementation for "perfect initialization" operations.

Tony Van Eerd offered many useful suggestions and corrections to the proposal.

People in discussion group "ISO C++ Standard - Future Proposals" provided numerous insightful suggestions: Vladimir Batov (who described and supported the perfect forwarding constructor), Nevin Liber, Ville

Voutilainen, Richard Smiths, Dave Abrahams, Chris Jefferson, Jeffrey Yasskin, Nikolay Ivchenkov, Matias Capeletto, Olaf van der Spek, Vincent Jacquet, Kazutoshi Satoda, Vicente J. Botet Escriba, Róbert Dávid, Vincent Jacquet, Luc Danton, Greg Marr, and many more.

Joe Gottman suggested the support for hashing some optional objects.

Nicol Bolas suggested to make `operator->` conditionally `constexpr` based on whether `T::operator&` is overloaded.

References

1. John J. Barton, Lee R. Nackman, "Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples".
2. Fernando Cacciola, Boost.Optional library (http://www.boost.org/doc/libs/1_49_0/libs/optional/doc/html/index.html)
3. MSDN Library, "Nullable Types (C# Programming Guide)", (http://msdn.microsoft.com/en-us/library/1t3y8s4s.aspx
4. Code Synthesis Tools, "C++ Object Persistence with ODB", (http://www.codesynthesis.com/products/odb/doc/manual.xhtml#7.3)
5. Jaakko Järvi, Boost Tuple Library (http://www.boost.org/doc/libs/1_49_0/libs/tuple/doc/tuple_users_guide.html)
6. Alisdair Meredith, John Lakos, "`noexcept` Prevents Library Validation" (N3248, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3248.pdf)
7. Walter E. Brown, "A Preliminary Proposal for a Deep-Copying Smart Pointer" (N3339, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3339.pdf)
8. Andrzej Krzemieński, Optional library implementation in C++11 (https://github.com/akrzemi1/Optional/)