`std::optional` from Scratch

https://wg21.link/n4606

http://en.cppreference.com/w/cpp/utility/optional

http://codereview.stackexchange.com/questions/127498/an-optionalt-implementation

What is `std::optional`?

- An *optional object for object types* is an object that contains the storage for another object and manages the lifetime of this *contained object*.
- 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.

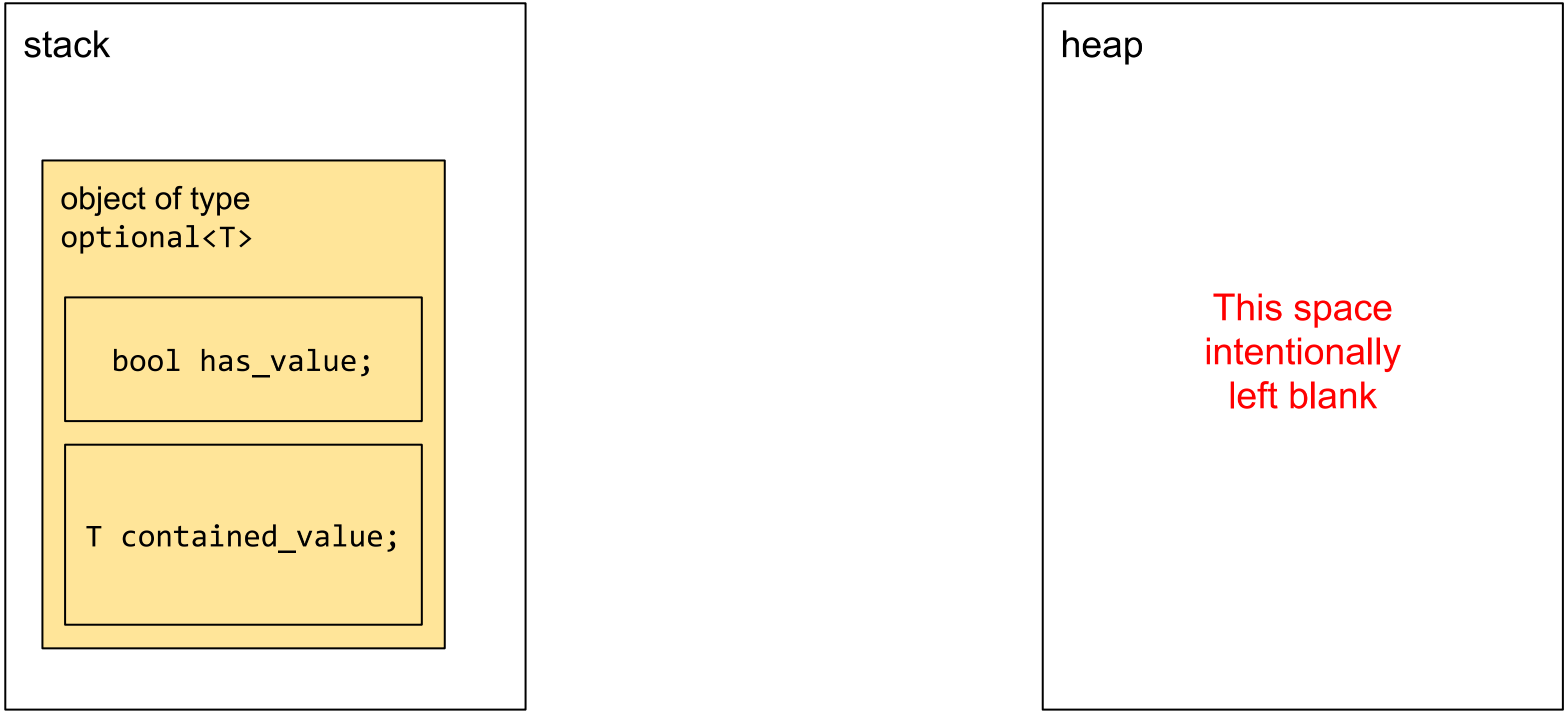
N4606 §20.6.1 [optional.general] /1

What is `std::optional`?

- Any instance of `optional<T>` at any given time either contains a value or does not contain a value.
- When an instance of `optional<T>` *contains a value*, it means that an object of type `T`, referred to as the optional object's *contained value*, is allocated within the storage of the optional object.
- Implementations are not permitted to use additional storage, such as dynamic memory, to allocate its contained value.

N4606 §20.6.3 [optional.object] /1

Conceptually...



We’ve probably all done this by hand

```
void frobnicate(const char *input)
{
    char *buffer = nullptr;
    bool has_buffer = false;
    if (...) {
        buffer = strdup(input);
        has_buffer = true;
    }
    // ...
    if (has_buffer) {
        free(buffer);
    }
}
```


Not a great example: conflates memory deallocation with destruction, and don’t need `has_buffer`

We probably haven't done *this*

```
void frobnicate(const char *input)
{
    aligned_storage_t<sizeof(string), alignof(string)> buffer;
    bool has_buffer = false;
    if (...) {
        new (&buffer) string(input);
        has_buffer = true;
    }
    // ...
    if (has_buffer) {
        reinterpret_cast<string&>(buffer).~string();
    }
}
```

We might start doing this instead

```
void frobnicate(const char *input)
{
    optional<string> buffer;
    if (...) {
        buffer.emplace(input);
    }
    // ...
    if (buffer.has_value()) {  // Actually, this is just what the
        buffer.reset();        // destructor does on our behalf;
    }                          // we wouldn’t actually write this.
}
```

We might start doing this instead

```
void frobnicate(const char *input)
{
    optional<string> buffer;
    if (...) {
        buffer.emplace(input);
    }
    // ...



}
```

Naïve approach

```
template<class T>
struct optional {
    T val_;          // a.k.a. contained_value
    bool engaged_;   // a.k.a. has_value

    // ...
};
```

This won't work as written, because the whole point is that we don't *want* `optional<T>`'s default constructor to construct a `T` object! How can we get an appropriately sized and aligned block of memory for a `T`, without actually constructing that `T`?

std::aligned_storage_t

`std::aligned_storage` has been around since C++11; the `_t` alias template was added in C++14. We could implement `optional` in terms of it:

```
#include <type_traits>

template<class T>
struct optional {
    std::aligned_storage_t<sizeof(T), alignof(T)> buf_;
    bool engaged_;
};
```


N4606 §20.15.7.6 [meta.trans.other]

std::aligned_storage_t

```
#include <new>
#include <type_traits>

template<class T> struct optional {
    std::aligned_storage_t<sizeof(T), alignof(T)> buf_;
    bool engaged_;

    optional() : engaged_(false) {}

    optional(const T& t): engaged_(true) { ::new ((void *)&buf_) T(t); }

    ~optional() {
        if (engaged_) reinterpret_cast<T&>(buf_).~T();
    }
};
```

But this gets messy quickly.

N4606 §20.15.7.6 [meta.trans.other]

Anonymous unions to the rescue!

We could use `std::aligned_storage_t`, but it turns out that the code gets much cleaner if we use core language features instead.

```
template<class T>
struct optional {
    union {
        char dummy_;
        T val_;          // a.k.a. contained_value
    };
    bool engaged_;  // a.k.a. has_value

    // ...
};
```


N4606 §9.3.1 [class.union.anon]

Anonymous unions to the rescue!

```
template<class T> struct optional {
    union { char dummy_; T val_; };
    bool engaged_;

    optional() : dummy_(0), engaged_(false) {}

    optional(const T& t) : val_(t), engaged_(true) {}

    ~optional() {
        if (engaged_) val_.~T();
    }
};
```


N4606 §9.3.1 [class.union.anon]

`~optional` is *conditionally* trivial

The standard requires the following test case to compile:

```
template<typename T> constexpr bool trivial =
    std::is_trivially_destructible<T>::value;

static_assert(trivial<int>);
static_assert(trivial<std::optional<int>>);

// ... and in general,
static_assert(trivial<std::optional<X>> == trivial<X>);
```


N4606 §20.6.3.2 [optional.object.dtor]

~optional is *conditionally* trivial

```
template<class T> struct optional {
    union { char dummy_; T val_; };
    bool engaged_;

    optional() : dummy_(0), engaged_(false) {}

    optional(const T& t) : val_(t), engaged_(true) {}

    ~optional() {  // This is not a trivial destructor!
        if (engaged_) val_.~T();
    }
};
```


N4606 §20.6.3.2 [optional.object.dtor]

~optional is *conditionally* trivial

```
template<class T> struct optional {
    union { char dummy_; T val_; };
    bool engaged_;

    optional() : dummy_(0), engaged_(false) {}

    optional(const T& t) : val_(t), engaged_(true) {}

    ~optional() {  // This also is not a trivial destructor!
        if constexpr (!std::is_trivially_destructible<T>{}) {
            if (engaged_) val_.~T();
        }
    }
};
```


N4606 §20.6.3.2 [optional.object.dtor]

We must *factor out* a maybe-trivial dtor.

```
template<class T, class E = void> struct optional_storage {
    union { char dummy_; T val_; };
    bool engaged_;

    ~optional_storage() {
        if (engaged_) val_.~T();  // this destructor is not trivial
    }
};

template<class T>  // partial specialization
struct optional_storage<T, enable_if_t<is_trivially_destructible_v<T>>> {
    union { char dummy_; T val_; };
    bool engaged_;

    ~optional_storage() = default;  // this destructor is trivial
};

template<class T> struct optional {
    optional_storage<T> storage;
    ~optional() = default;  // this destructor is sometimes trivial
};
```

The rest of the `optional` interface

```
template<class T> struct optional {

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

    T& value();
    template<class U> T value_or(U&& u) const;

    bool has_value() const;
};
```


I’m handwaving the exact signatures in some cases.

The rest of the `optional` interface

```
template<class T> struct optional {

    optional& operator= (const T&);  // somewhat of a lie
    optional& operator= (std::nullopt_t) noexcept;

    T& operator*() noexcept;  // and const versions of all these
    T& operator->() noexcept;

    explicit operator bool() const noexcept;
};
```


I’m handwaving the exact signatures in some cases.

Why have one syntax when you could have two?

- Two different syntaxes to ask “Is the object engaged?”
  - operator bool
  - has_value()
- Two different syntaxes to fetch the value of an engaged object
  - operator*, operator->
  - value()
- Three or four different syntaxes to construct a contained object
  - emplace()
  - = some-T-object
- Three or four different syntaxes to disengage the object
  - reset()
  - = nullopt
  - = some-optional-object

Why have one syntax when you could have two?

```
#include <optional>
struct Data { int i; Data(): i(42) {} };

int main()
{
    std::optional<Data> o;

    // TO QUERY

    if (o) puts("engaged");
    if (o.has_value()) puts("engaged");
}
```

Why have one syntax when you could have two?

```
#include <optional>
struct Data { int i; Data(): i(42) {} };

int main()
{
    std::optional<Data> o;

    // TO FETCH

    Data& a = *o;           // like vec[i]: does not throw
    Data& b = o.value();  // like vec.at(i): may throw bad_optional_access
}
```

Digression: Value categories

```
struct Data {
    Data() noexcept { puts("default ctor"); }
    Data(const Data&) noexcept { puts("copy ctor"); }
    Data(Data&&) noexcept { puts("move ctor"); }
    Data& operator=(Data&&) noexcept { puts("move assign"); return *this; }
    Data& operator=(const Data&) noexcept { puts("copy assign"); return *this; }
    ~Data() { puts("dtor"); }
};

auto v() { std::vector<Data> r; r.emplace_back(); return r; }

int main() {
    Data d;
    d = v()[0];  // What kind of assignment is this?
}
```


N4606 §23.3.11.1 [vector.overview]

Digression: Value categories

```
struct Data {
    Data() noexcept { puts("default ctor"); }
    Data(const Data&) noexcept { puts("copy ctor"); }
    Data(Data&&) noexcept { puts("move ctor"); }
    Data& operator=(Data&&) noexcept { puts("move assign"); return *this; }
    Data& operator=(const Data&) noexcept { puts("copy assign"); return *this; }
    ~Data() { puts("dtor"); }
};

auto v() { std::vector<Data> r; r.emplace_back(); return r; }

int main() {
    Data d;
    d = v()[0];  // What kind of assignment is this? It’s a copy assignment.
}
```

N4606 §23.3.11.1 [vector.overview]

Digression: Value categories

```
struct Data {
    Data() noexcept { puts("default ctor"); }
    Data(const Data&) noexcept { puts("copy ctor"); }
    Data(Data&&) noexcept { puts("move ctor"); }
    Data& operator=(Data&&) noexcept { puts("move assign"); return *this; }
    Data& operator=(const Data&) noexcept { puts("copy assign"); return *this; }
    ~Data() { puts("dtor"); }
};

auto o() { std::optional<Data> r; r.emplace(); return r; }

int main() {
    Data d;
    d = *o();  // What kind of assignment is this?
}
```


N4606 §20.6.3 [optional.object]

Digression: Value categories

```
struct Data {
    Data() noexcept { puts("default ctor"); }
    Data(const Data&) noexcept { puts("copy ctor"); }
    Data(Data&&) noexcept { puts("move ctor"); }
    Data& operator=(Data&&) noexcept { puts("move assign"); return *this; }
    Data& operator=(const Data&) noexcept { puts("copy assign"); return *this; }
    ~Data() { puts("dtor"); }
};

auto o() { std::optional<Data> r; r.emplace(); return r; }

int main() {
    Data d;
    d = *o();  // What kind of assignment is this? It's a move assignment.
}
```


N4606 §20.6.3 [optional.object]

Digression: Value categories

Ever since C++98, we’ve had the ability to overload member functions based on the cv-qualifiers of the implicit object parameter — that is, its degree of “constness” and/or its degree of “volatileness”.

In C++11, with the addition of rvalue references, we gained the ability to overload member functions based on the ref-qualifier of the implicit object parameter — that is, its degree of “rvalueness”.

```
void foo();  // used for non-const lvalues
void foo() const;  // used for const lvalues
void foo() &&;  // used for non-const rvalues
```


https://akrzemi1.wordpress.com/2014/06/02/ref-qualifiers/

Digression: Value categories

Rvalue-ref-qualified member functions can help enable move semantics.
Some standard library types implement rvalue-ref-qualified member functions, and some do not.

```
auto o() { return std::make_optional<Data>(); }
auto u() { return std::make_unique<Data>(); }
auto v() { return std::vector<Data>(1, Data()); }

int main() {
    Data d;
    d = *o();  // move assignment
    d = *u();  // copy assignment
    d = v()[0];  // copy assignment
}
```

https://akrzemi1.wordpress.com/2014/06/02/ref-qualifiers/
http://melpon.org/wandbox/permlink/fC8vUsJAwGO8SjNc

Implement operator*() for real

```
// ...
constexpr T& operator*() & {
    return val_;
}
constexpr const T& operator*() const & {
    return val_;
}
constexpr T&& operator*() && {
    return std::move(val_);
}
constexpr const T&& operator*() const && {
    return std::move(val_);
}
// ...
```


These signatures are now 100% exact.

Implement `value()` for real

```
// ...
T& value() & {
    return engaged_ ? val_ : throw bad_optional_access();
}
const T& value() const & {
    return engaged_ ? val_ : throw bad_optional_access();
}
T&& value() && {
    return engaged_ ? std::move(val_) : throw bad_optional_access();
}
const T&& value() const && {
    return engaged_ ? std::move(val_) : throw bad_optional_access();
}
// ...
```


These signatures are now 100% exact.

Implement `value_or()` for real

Since we are returning a result by value, the only “special” case is when `*this` is a non-const rvalue; then it is safe for us to steal its guts. Every other case *must* copy.

```
// ...
template<class U> T value_or(U&& u) && {  // non-const rvalue
    return engaged_ ? std::move(val_) :
                      static_cast<T>(std::forward<U>(u));
}

template<class U> T value_or(U&& u) const {  // every other case
    return engaged_ ? val_ :
                      static_cast<T>(std::forward<U>(u));
}
// ...
```


These signatures are now 100% exact.

Why have one syntax when you could have two?

```
#include <optional>
struct Data { int i; Data(): i(42) {} };

int main()
{
    std::optional<Data> o;

    // TO ENGAGE                              // TO DISENGAGE

    o = std::optional<Data>{Data{}};   o = std::optional<Data>{};
    o = Data{};                        o = {};
    o = std::make_optional<Data>();    o = std::nullopt;
    o.emplace();                       o.reset();
}
```

N4606 §20.6.4 [optional.nullopt]

http://stackoverflow.com/questions/37872533/stdnullopt-t-constructor-rationale

Let’s talk about constructors.

Q: How many different ways are there to construct an `optional<T>`?

A: All the ways there are to construct a `T`, plus one!

“Empty” versus “disengaged”

```
optional<const char *> o1;                assert(!o1);
optional<const char *> o2 = "hello";      assert(o2 && *o2);
optional<const char *> o3 = nullptr;      assert(o3 && !*o3);
optional<const char *> o4 {};             assert(!o4);
optional<const char *> o5 {nullptr};      assert(o5 && !*o5);
optional<const char *> o6 {"hello"};      assert(o6 && *o6);
optional<const char *> o7 {{}};           assert(o7 && !*o7);
optional<optional<int>> o8 {};            assert(!o8);
optional<optional<int>> o9 {{}};          assert(o9 && !*o9);
optional<optional<int>> oA {{0}};         assert(oA && *oA && !**oA);
```

`std::nullopt` means “disengaged”

```
optional<const char *> o1 = nullopt;  assert(!o1);
optional<optional<int>> o2 = nullopt; assert(!o2);
optional<optional<int>> o3 {{{{nullopt}}}}; assert(o3 && !*o3);
optional<optional<int>> o4 {{0}};     assert(o4 && *o4 && !**o4);

// One of these things is not like the others...
```

`optional<optional<int>> o3 {nullopt};` will prefer to call `optional<optional<int>>(nullopt_t)` instead of the perfect-forwarding template constructor.

`std::in_place` helps with `nullopt`

```
optional<const char *> o1(nullopt);  assert(!o1);
optional<optional<int>> o2(nullopt); assert(!o2);
optional<optional<int>> o3(in_place, nullopt);
    assert(o3 && !*o3);
optional<optional<int>> o4(in_place, in_place, 42);
    assert(o4 && *o4 && **o4 == 42);
```

`in_place` means “I explicitly want to emplace a contained value using these arguments. Don’t try any other constructors; just strip off the first `in_place` and construct a `T` with the rest.”

So our final list of constructors is...

```
// Construct disengaged.
constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;

// Construct by move or copy from an existing optional<T> object.
optional(const optional&);
optional(optional&&) noexcept(is_nothrow_move_constructible_v<T>);

// Construct engaged, emplacement-style (possibly from a T object).
template<class U> constexpr optional(U&&);

// Construct engaged, emplacement-style.
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&&...);
```

So our final list of constructors is...

```
// Construct disengaged.
constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;

// Construct by move or copy from an existing optional<T> object.
optional(const optional&);
optional(optional&&) noexcept(is_nothrow_move_constructible_v<T>);

// Construct engaged, emplacement-style (possibly from
template<class U> constexpr optional(U&&);

// Construct engaged, emplacement-style.
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&&...);
```

What's this one for?

Perfectly forwarding `initializer_list`

```
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&&...);

optional<std::vector<int>> o1(in_place, 10, 42);
  assert(o1 && (*o1).size() == 10);

optional<std::vector<int>> o2(in_place, {10, 42});
  assert(o2 && (*o2).size() == 2);
```

Recall that *braced-initializer-list* expressions have no type, so they don't contribute to template type deduction. So o2 does not compile unless we uncomment the extra constructor above.

Perfectly forwarding `initializer_list`*

```
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&&...);
```

```
struct S {
    S(std::initializer_list<int>, int) {}
    S(int, std::initializer_list<int>, int) {}
};
```

```
S a ({1,2,3}, 4);  // OK
S b (1, {2,3,4}, 5);  // OK
optional<S> a (in_place, {1,2,3}, 4);  // OK
optional<S> b (in_place, 1, {2,3,4}, 5);  // error
```

Implement `nullopt_t` for real

`nullopt_t` is a tag type, that is, its sole purpose is to be a unique identifier within the type system. So normally we'd just make it an empty struct type.

```
struct nullopt_t {};
constexpr nullopt_t nullopt;
```

But there's a complication here! Consider:

```
struct S {};
std::optional<S> o;
o = {};
```

Does this mean `o = nullopt_t{}` or `o = S{}` or `o = optional<S>{}`?

http://melpon.org/wandbox/permlink/HnFXRjLyqi4s4ikv

Implement `nullopt_t` for real

```
// Disengage.
optional& operator=(nullopt_t) noexcept;

// Assign by move or copy from an existing optional<T> object.
optional& operator=(const optional&);
optional& operator=(optional&&);

// Assign engaged, from an existing T object.
optional& operator=(const T&);
optional& operator=(T&&);
```

```
struct S {};
std::optional<S> o;
o = {};
```

Does this mean `o = nullopt_t{}` or `o = S{}` or `o = optional<S>{}`?

http://melpon.org/wandbox/permlink/HnFXRjLyqi4s4ikv

Implement `nullopt_t` for real

```
// Disengage.
optional& operator=(nullopt_t) noexcept;

// Assign by move or copy from an existing optional<T> object.
optional& operator=(const optional&);
optional& operator=(optional&&);

// Assign engaged, from an existing T object.
template<class U, class = enable_if_t<is_same_v<decay_t<U>, T>>>
optional& operator=(U&&);
```

```
struct S {};
std::optional<S> o;
o = {};
```

Does this mean `o = nullopt_t{}` or **`o = S{}`** or `o = optional<S>{}`?

http://melpon.org/wandbox/permlink/HnFXRjLyqi4s4ikv

Implement `nullopt_t` for real

```
// Disengage.
optional& operator=(nullopt_t) noexcept;

// Assign by move or copy from an existing optional<T> object.
optional& operator=(const optional&);
optional& operator=(optional&&);

// Assign engaged, from an existing T object.
template<class U, class = enable_if_t<is_same_v<decay_t<U>,
T>>>
optional& operator=(U&&);
```

```
struct S {};
std::optional<S> o;
o = {};
```

Now `o = S{}` is right out. We just need to eliminate `o = nullopt_t{}` as a possibility...

http://melpon.org/wandbox/permlink/HnFXRjLyqi4s4ikv

Implement `nullopt_t` for real

`nullopt_t` is a tag type, that is, its sole purpose is to be a unique identifier within the type system. So normally we'd just make it an empty struct type... but we have to make sure that there's no implicit conversion from a pair of empty braces to `nullopt_t`.

```
struct nullopt_t {
    constexpr explicit nullopt_t(int) {}
};
constexpr nullopt_t nullopt{42};
```

There. Now `{}` can't possibly be mistaken for `nullopt_t{}`... which means that `o = {}` is unambiguously interpreted as `o = optional<S>{}`, i.e., "disengage o."

20.6.4 [optional.nullopt] /2
http://melpon.org/wandbox/permlink/HnFXRjLyqi4s4ikv

make_optional

Recall that we say `make_unique<T>(v)` for a smart pointer, but simply `make_tuple(v)` for a 1-ary tuple. `optional` is schizophrenic about whether it's pointer-like or container-like, so it's equally schizophrenic about how you `make` one of it.

```
std::optional<int> o1 = make_optional(42);
std::optional<long> o2 = make_optional<long>(42);
auto o3 = make_optional<std::vector<int, A>>({1,2,3}, A());
```

Notice that `make_optional(nullopt)` means `make_optional<nullopt_t>(nullopt)`, which is ill-formed. You aren't allowed to express `optional<nullopt_t>`.

Some ctors are *conditionally* constexpr

```
constexpr optional(const T& v);
```

Requires: `is_copy_constructible_v<T>` is `true`.
Effects: Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `v`.
Postcondition: `*this` contains a value.
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.

N4606 §7.1.5 [dcl.constexpr] /5,6

Some ctors are *conditionally* constexpr

It turns out that we get this for free!
You can put the `constexpr` specifier indiscriminately onto any function you want, as long as it's vaguely templatey.
I don't see any Standard wording about this, so I think vendors are doing it just to make their library implementations a bit easier.

```
constexpr void foo() { puts("hello"); }  // OK on GCC, error on Clang/MSVC
template<int=0> constexpr void foo() { puts("hello"); }  // OK
template<int=0> struct S { constexpr void foo() { puts("hello"); } }; //
OK
```


N4606 §7.1.5 [dcl.constexpr] /5,6

Some ctors are *conditionally* explicit

The standard requires the following behavior:

```
struct A { A(int); };
std::optional<A> oa(42);  // OK
std::optional<A> oa{42};  // OK
std::optional<A> oa = 42;  // OK

struct B { explicit B(int); };
std::optional<B> ob(42);  // OK
std::optional<B> ob{42};  // OK
std::optional<B> ob = 42;  // error
```


N4618 §20.6.3.1 [optional.ctor] /22

Some ctors are *conditionally* explicit

```
template <class U = T>
  EXPLICIT constexpr optional(U&& v);
```

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

Postconditions: `*this` contains a value.

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

Remarks: [...] **The constructor is explicit if and only if `is_convertible_v<U&&, T>` is `false`.**

N4618 §20.6.3.1 [optional.ctor] /22

We must *repeat* a maybe-explicit ctor.

```
template<class T> struct optional {
    optional_storage<T> storage;

    template<class U>
    optional(U&& u) : storage(std::forward<U>(u)) {}
};
```


N4618 §20.6.3.1 [optional.ctor] /22

We must *repeat* a maybe-explicit ctor.

```
template<class T> struct optional {
    optional_storage<T> storage;

    template<class U, std::enable_if_t<X, int> = 0>
    optional(U&& u) : storage(std::forward<U>(u)) {}

    template<class U, std::enable_if_t<Y, int> = 0>
    explicit optional(U&& u) : storage(std::forward<U>(u)) {}
};

// For the record:
#define X std::is_constructible_v<T,U&&> && std::is_convertible_v<U&&,T>
#define Y std::is_constructible_v<T,U&&> &&
!std::is_convertible_v<U&&,T>
```


N4618 §20.6.3.1 [optional.ctor] /22

We must *repeat* a maybe-explicit ctor.

```
template<class T> struct optional {
    optional_storage<T> storage;

    template<class U, std::enable_if_t<X, int> = 0>
    optional(U&& u) : storage(std::forward<U>(u)) {}

    template<class U, std::enable_if_t<Y, int> = 0>
    explicit optional(U&& u) : storage(std::forward<U>(u)) {}
};
```

Now, the standard actually said one more unusual thing here:

```
    template <class U = T>
      EXPLICIT constexpr optional(U&& v);
```


http://stackoverflow.com/questions/34876969

Default a deducible template parameter?

```
template <class U = T>
  EXPLICIT constexpr optional(U&& v);
```

This looks redundant. Why specify that `U` defaults to `T`, when `U` can always(?) be deduced from the type of v in the first place?

It turns out that some constructs' types cannot be deduced. Braced-initializer-lists are one example, but not the relevant example in this case. The relevant example here is ***overloaded function names.***

```
void f(int);
void f(double);
optional<void(*)(int)> o = f;  // we mean f(int)
```

Without the unusual ***=T***, this code would fail to compile!

http://stackoverflow.com/questions/34876969