Document Number: D????
Date: 2014-02-??
Revises: N3848
Editor: Jeffrey Yasskin
Google, Inc.
jyasskin@google.com

Working Draft, Technical Specification on C++ Extensions for Library Fundamentals

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

1 General [general]

1.1 Scope [general.scope]

1 This technical specification describes extensions to the C++ Standard Library (1.2). These extensions are classes and functions that are likely to be used widely within a program and/or on the interface boundaries between libraries written by different organizations.

2 This technical specification is non-normative. Some of the library components in this technical specification may be considered for standardization in a future version of C++, but they are not currently part of any C++ standard. Some of the components in this technical specification may never be standardized, and others may be standardized in a substantially changed form.

3 The goal of this technical specification it to build more widespread existing practice for an expanded C++ standard library. It gives advice on extensions to those vendors who wish to provide them.

1.2 Normative references [general.references]

1 The following referenced document is indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 14882:2011, *Programming Languages — C++*

2 ISO/IEC 14882:2011 is herein called the *C++ Standard*. References to clauses within the C++ Standard are written as "C++11 §3.2". The library described in ISO/IEC 14882:2011 clauses 17–30 is herein called the *C++ Standard Library*.

3 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++11 §17) is included into this Technical Specification by reference.

1.3 Namespaces and headers [general.namespaces]

1 Since the extensions described in this technical specification are experimental and not part of the C++ standard library, they should not be declared directly within namespace `std`. Unless otherwise specifed, all components described in this technical specification are declared in namespace `std::experimental::fundamentals_v1` or a subnamespace thereof. Each header described in this technical specification shall import the contents of `std::experimental::fundamentals_v1` into `std::experimental` as if by

> **Editor's note:** This section reflects the consensus between the LWG and LEWG at the Chicago 2013 meeting.

```
namespace std {
  namespace experimental {
    inline namespace fundamentals_v1 {}
  }
}
```

2 Unless otherwise specified, references to other entities described in this technical specification are assumed to be qualified with `std::experimental::fundamentals_v1::`, and references to entities described in the standard are assumed to be qualified with `std::`.

3 Extensions that are expected to eventually be added to an existing header `<meow>` are provided inside the `<experimental/meow>` header, which shall include the standard contents of `<meow>` as if by

```
#include <meow>
```

4 [*Note:* This implies that this technical specification cannot remove an existing overload of a function. — *end note*]

5 New headers are also provided in the `<experimental/>` directory, but without such an `#include`.

Table 1 —
C++ library headers

`<experimental/optional>`

1.4 Terms and definitions [general.defns]

1 For the purposes of this document, the terms and definitions given in the C++ Standard and the folowing apply.

1.4.1 [general.defns.direct-non-list-init]

direct-non-list-initialization
A direct-initialization that is not list-initialization.

1.5 Future plans (Informative) [general.plans]

1 This section describes tentative plans for future versions of this technical specification and plans for moving content into future versions of the C++ Standard.

2 The C++ committee intends to release a new version of this technical specification approximately every year, containing the library extensions we hope to add to a near-future version of the C++ Standard. Future versions will define their contents in `std::experimental::fundamentals_v2`, `std::experimental::fundamentals_v3`, etc., with the most recent implemented version inlined into `std::experimental`.

3 When an extension defined in this or a future version of this technical specification represents enough existing practice, it will be moved into the next version of the C++ Standard by removing the `experimental::fundamentals_v`*N* segment of its namespace and by removing the `experimental/` prefix from its header's path.

2 Optional objects [optional]

2.1 In general [optional.general]

[1] 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.

2.2 Header `<experimental/optional>` synopsis [optional.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v1 {
  // 2.4, optional for object types
  template <class T> class optional;

  // 2.5, In-place construction
  struct in_place_t{};
  constexpr in_place_t in_place{};

  // 2.6, Disengaged state indicator
  struct nullopt_t{see below};
  constexpr nullopt_t nullopt(unspecified);

  // 2.7, Class bad_optional_access
  class bad_optional_access;

  // 2.8, 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>&);

  // 2.9, 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;

  // 2.10, 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>&);

  // 2.11, Specialized algorithms
  template <class T> void swap(optional<T>&, optional<T>&) noexcept(see below);
  template <class T> constexpr optional<see below> make_optional(T&&);

} // namespace fundamentals_v1
} // namespace experimental

  // 2.12, Hash support
  template <class T> struct hash;
  template <class T> struct hash<experimental::optional<T>>;
} // namespace std
```

1 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.

2.3 Definitions [optional.defs]

1 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>`.

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

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

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

  // 2.4.1, 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&&...);

  // 2.4.2, Destructor
  ~optional();

  // 2.4.3, 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&&...);

  // 2.4.4, Swap
  void swap(optional&) noexcept(see below);

  // 2.4.5, 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
};
```

1 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`.

2 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.

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

2.4.1 Constructors [optional.object.ctor]

1
```
constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;
```

2 *Postconditions:* `*this` is disengaged.

3 *Remarks:* No `T` object referenced is initialized. For every object type `T` these constructors shall be `constexpr` constructors (C++11 §7.1.5).

4
```
optional(const optional<T>& rhs);
```

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

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

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

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

9
```
optional(optional<T>&& rhs) noexcept(see below);
```

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

11 *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.

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

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

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

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

15
```
optional(const T& v);
```

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

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

18 *Postconditions:* `*this` is engaged.

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

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

21 `optional(T&& v);`

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

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

24 *Postconditions:* `*this` is engaged.

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

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

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

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

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

30 *Postconditions:* `*this` is engaged.

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

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

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

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

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

36 *Postconditions:* `*this` is engaged.

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

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

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

2.4.2 Destructor [optional.object.dtor]

1 `~optional();`

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

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

2.4.3 Assignment [optional.object.assign]

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

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

3 *Returns:* `*this`.

4 *Postconditions:* `*this` is disengaged.

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

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

7 *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.

8 *Returns:* `*this`.

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

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

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

13 *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.

14 *Returns:* `*this`.

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

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

```
is_nothrow_move_assignable<T>::value && is_nothrow_move_constructible<T>::value
```

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

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

20 *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*`)`.

21 *Returns:* `*this`.

22 *Postconditions:* `*this` is engaged.

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

25 *Notes:* The reason for providing such generic assignment and then constraining it so that effectively `T` == `U` is to guarantee that assignment of the form `o = {}` is unambiguous.

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

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

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

29 *Postconditions:* `*this` is engaged.

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

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

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

34 *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)...`.

35 *Postconditions:* `*this` is engaged.

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

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

2.4.4 Swap [optional.object.swap]

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

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

3 *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)`.

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

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

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

2.4.5 Observers [optional.object.observe]

1
```
constexpr T const* operator->() const;
T* operator->();
```

2 *Requires:* `*this` is engaged.

3 *Returns:* `val`.

4 *Throws:* Nothing.

5

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

6
```
constexpr T const& operator*() const;
T& operator*();
```

7 *Requires:* `*this` is engaged.

8 *Returns:* `*val`.

9 *Throws:* Nothing.

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

11
```
constexpr explicit operator bool() noexcept;
```

12 *Returns:* `init`.

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

14
```
constexpr T const& value() const;
T& value();
```

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

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

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

18
```
template <class U> constexpr T value_or(U&& v) const&;
```

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

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

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

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

24
```
template <class U> T value_or(U&& v) &&;
```

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

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

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

2.5 In-place construction [optional.inplace]

1
```
struct in_place_t{};
constexpr in_place_t in_place{};
```

2 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.

2.6 Disengaged state indicator [optional.nullopt]

1
```
struct nullopt_t{see below};
constexpr nullopt_t nullopt(unspecified);
```

2 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.

3 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.

2.7 Class `bad_optional_access` [optional.bad_optional_access]

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

1 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.

2
```
bad_optional_access(const string& what_arg);
```

3 *Effects:* Constructs an object of class `bad_optional_access`.

4
```
bad_optional_access(const char* what_arg);
```

5 *Effects:* Constructs an object of class `bad_optional_access`.

2.8 Relational operators [optional.relops]

1
```
template <class T> constexpr bool operator==(const optional<T>& x, const optional<T>& y);
```

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

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

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

5
```
template <class T> constexpr bool operator<(const optional<T>& x, const optional<T>& y);
```

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

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

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

2.9 Comparison with `nullopt` [optional.nullops]

1
```
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;
```

2 *Returns:* `(!x)`.

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

4 *Returns:* `false`.

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

6 *Returns:* `bool(x)`.

2.10 Comparison with `T` [optional.comp_with_t]

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

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

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

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

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

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

2.11 Specialized algorithms [optional.specalg]

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

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

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

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

2.12 Hash support [optional.hash]

1 `template <class T> struct hash<experimental::optional<T>>;`

2 *Requires:* the template specialization `hash<T>` shall meet the requirements of class template `hash` (C++11 §20.8.12). The template specialization `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)`.