Homogeneous interface for \texttt{variant}, \texttt{any} and \texttt{optional} (Revision 1)

This paper is the 1st revision of [P0032R0] taking in account the feedback from Kona meeting.

This paper identifies some differences in the design of \texttt{variant<Ts...>}, \texttt{any} and \texttt{optional<T>}, diagnoses them as owing to unnecessary asymmetry between those classes, and proposes wording to eliminate the asymmetry.

Contents

History.................................................................................................................................................. 2
  Revision 1........................................................................................................................................ 2
Introduction......................................................................................................................................... 3
Motivation and Scope........................................................................................................................... 3
Proposal.............................................................................................................................................. 4
Design rationale.................................................................................................................................... 5
  \texttt{in\_place} constructor.................................................................................................................. 5
  Differences between the new \texttt{in\_place\_t} and the old one............................................................ 6
  \texttt{emplace} forward member function.......................................................................................... 7
  About \texttt{empty()}\slash\texttt{explicit\ operator bool()} member functions .............................................. 7
  About \texttt{clear()}\slash\texttt{reset()} member functions......................................................................... 8
  About a not-a-value \texttt{any\ :\ none}................................................................................................. 8
  Do we need an explicit \texttt{make\_any} factory?............................................................................... 9
  About \texttt{emplace} factories............................................................................................................ 9
  Which file for \texttt{in\_place\_t} and \texttt{in\_place}?.................................................................................. 10
  Access interface............................................................................................................................. 10
Open points....................................................................................................................................... 10
Technical Specification.......................................................................................................................... 10
  General utilities library.................................................................................................................... 10
  Optional objects............................................................................................................................... 11
  Class \texttt{any} ................................................................................................................................. 11
Acknowledgements.............................................................................................................................. 14
References.......................................................................................................................................... 14
Appendix.......................................................................................................................................... 14
History

Revision 1

The 1st revision of [P0032R0] takes in account the feedback from Kona meeting.
Next follow the direction of the committee: globally keep the consensual part and extract the conflicting and less polished part.

- Do we want to adopt the new in_place definition?

It is clear that we want a different name for the emplace function and the tag, however it is not clear the committee wants the in_place function reference. Nevertheless, the author don't know how to have the in_place both for optional, any and variant without using function references, so this paper preserve this design.

| Leave optional different from variant and any | 6 |
| Member function is emplace; tag type is in_place | 13 |
| Both are emplace | 6 |

Do we want to adopt the new in_place definition?

SF F N A SA
1 3 8 0 0

- Do we want in place constructor for any?  Unanimous Yes.

- Do we want the clear and reset changes? Yes

How to empty an any or optional?

- .reset() 12
- .clear() 7
- =none (different paper) 7
- ={} 5
- .drain() 1

- Do we want the operator bool changes? No, instead a .something() member function (e.g. has_value) is preferred for the 3 classes. This doesn't mean yet that we replace the existing explicit operator bool in optional.

Do we want emptiness checking to be consistent between any/optional?  
Unanimous yes

| Provide operator bool for both | Y: 6 N: 5 |
| Provide .something() | Y: 17 N: 0 |
| Provide ==() | Y: 0 N: 5 |
| Provide ==std::none | Y: 5 N: 2 |
| something(any/optional) | Y: 3 N: 8 |

- Do we want the not-a-value none? No, too much unit types. The committee wants a separated paper for a generic none_t/none, none_tc_t<TC>/none<TC>.

Do we want none_t to be a separate paper?

SF F N A SA
11 1 3 0 0
Introduction

This paper identifies some differences in the design of \texttt{variant<Ts...>}, \texttt{any} and \texttt{optional<T>}, diagnoses them as owing to unnecessary asymmetry between those classes, and proposes wording to eliminate the asymmetry.

The identified issues are related to the last Fundamental TS proposal [N4480] and the variant proposal [N4542] and concerns mainly:

- coherency of functions that behave the same but that are named differently,
- replace the \texttt{in\_place} tag by a function with overloads for type and index,
- replacement of \texttt{emplace\_type<T>}/\texttt{emplace\_index<I>} by \texttt{in\_place<T>}/\texttt{in\_place<I>}
- addition of emplace factories for \texttt{any} and \texttt{optional} classes.

Motivation and Scope

Both \texttt{optional} and \texttt{any} are classes that can store possibly some underlying type. In the case of \texttt{optional} the underlying type is know at compile time, for \texttt{any} the underlying type is any and know at run-time.

If the variant proposal ends by having nullable variant, the stored type would be any of the \texttt{T} or a \texttt{not-a-value} type, know at run-time. Let me refer to this possible variant of \texttt{nullable\_variant <Ts...>}

The following inconsistencies have been identified:

- \texttt{variant<Ts...>} and \texttt{optional} provides in place construction with different syntax
while any requires a specific instance.

- `variant<Ts...>` and `optional` provides emplace assignment while any requires a specific instance to be assigned.

- The in place tags for `variant<Ts...>` and `optional` are different. However the name should be the same. Any doesn't provides in place construction and assignment yet.

- `any` provides `any::clear()` to unset the value while `optional` uses assignment from a `nullopt_t` or from `{}`. This paper doesn't contains any proposal to improve this situation. A separated paper would include a generic `none_t/none` proposal.

- `optional` provides a explicit `bool` conversion while any provides an `any::empty` member function.

- `optional<T>, variant<Ts...>` and any provides different interfaces to get the stored value. `optional` uses a value member function and pointer-like functions, `variant` uses a tuple like interface, while any uses a cast like interface. As all these classes are in someway classes that can possibly store a specific type, the first two limited and know at compile time, the last unlimited, it seems natural that all provide the same kind of interface. This paper doesn't contains any proposal to improve this situation. A separated paper would include a generic `none_t/none` proposal.

The C++ standard should be coherent for features that behave the same way on different types. Instead of creating specific issues, we have preferred to write a specific paper so that we can discuss of the whole view.

**Proposal**

We propose to:

- Replace `in_place_t/in_place` by an overloaded function (see [eggs-variant]).

- In class `optional<T>`
  - Add a `reset` member function.
  - Add a `has_value` member function.

- Add an additional overload for `make_optional` factory to emplace construct.

- In class `any`
  - make the default constructor `constexpr`,
  - add `in_place` forward constructors,
  - add `emplace` forward member functions,
  - rename the `empty` function with `has_value` and make it `constexpr`,
  - rename the `clear` member function to `reset`,

- Add a `make_any` factory to emplace construct.

- In class `variant<T>`
  - Replace the uses of `emplace_type_t<T>/emplace_index_t<T>` by
Homogeneous interface for variant, any and optional(R1)

- Replace the uses of `emplace_type<T>/emplace_index<I>` by `in_place<T>/in_place<I>`.

**Design rationale**

**in_place constructor**

`optional<T>` in place constructor constructs implicitly a `T`.

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

In place construct for `any` can not have an implicit type `T`. We need a way to state explicitly which `T` must be constructed in place.

```cpp
struct in_place_tag {};
template <class T>
using in_place_type_t = in_place_tag(&)(unspecified<T>);
template <class T>
in_place_tag in_place(unspecified<T>) { return {} ; }
```

The function `in_place_tag(&)(unspecified<T>)` is used to transport the type `T` participating in overload resolution.

```cpp
template <class T, class ...Args>
any(in_place_type_t<T>, , Args&& ...);
```

This can be used as

```cpp
any(in_place<X>, v1, ..., vn);
```

Adopting this template class to `optional` would needs to change the definition of `in_place_t/in_place` to

```cpp
using in_place_t = in_place_tag(&)(unspecified);
in_place_tag in_place(unspecified) { return {} ; }
```

The same applies to `variant`. We need an additional overload for `in_place`

```cpp
template <int I>
using in_place_index_t = in_place_tag(&)(unspecified<I>);
template <int I>
in_place_tag in_place(unspecified<I>) { return {} ; }
```

Given

```cpp
struct Foo { Foo(int, double, char); };
```

Before:

```cpp
optional<Foo> of(in_place, 0, 1.5, 'c');
variant<int, Foo> vf(emplace_type<Foo>, 0, 1.5, 'c');
variant<int, Foo> vf(emplace_index<1>, 0, 1.5, 'c');
any af(Foo(0, 1.5, 'c')); // (*)
```
After:

```cpp
optional<Foo> of(in_place, 0, 1.5, 'c');
variant<int, Foo> vf(in_place<Foo>, 0, 1.5, 'c');
variant<int, Foo> vf(in_place<int>, 0, 1.5, 'c');
any af(in_place<Foo>, 0, 1.5, 'c');
```

Note that before any didn't support non-copyable-non-moveable objects like `std::mutex`. With `in_place` we are able to store a `mutex` in.

### Differences between the new `in_place_t` and the old one

#### Cost of function reference versus tags

The prosed function reference for `in_place_t(&)(unspecified)` takes the size of an address while the previous `in_place_t` struct tag was empty and so its size is 1. We don't think this would reduce significantly the performances an believe that it can even perform better, however some measure would be done if there is an interest.

#### Possible malicious attacks

Unfortunately using function references would work for any unary function taken the unspecified type and returning `in_place_tag` in addition to `in_place`. Of course defining such a function would imply to hack the unspecified type. This can be seen as a hole on this proposal, but the author think that it is better to have a uniform interface than protecting from malicious attacks from a hacker.

#### No default constructible

While adapting `optional<T>` to the new `in_place_t` type we found that we can not anymore use `in_place_t{}`. The authors don't consider this a big limitation as the user can use `in_place` instead.

It needs to be noted that this is in line with the behavior of `nullopt_t` as `nullopt_t{}` fails as no default constructible. However `nullptr_t{}` seems to be well formed.

#### Not assignable from `{}`

After a deeper analysis we found also that the old `in_place_t` supported

```cpp
in_place_t t = {};
```

The authors don't consider this a big limitation as we don't expect that a lot of users could use this and the user can use `in_place` instead.

```cpp
in_place_t t = in_place;
```

It needs to be noted that this is in line with the behavior of `nullopt_t` as the following compile fails.
nullopt_t t = {}; // compile fails

However nullptr_t seems to be support it.

nullptr_t t = {}; // compile pass

emplace forward member function

optional<T> emplace member function emplaces implicitly a T.

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

emplace for any can not have an implicit type T. We need a way to state explicitly which T must be emplaced.

    template <class T, class ...Args>
    any::emplace(Args&& ...);

and used as follows

    any af;
    optional<Foo> of;
    variant<int, Foo> vf;
    af.emplace<Foo>(v1, ..., vn);
    of.emplace<Foo>(v1, ..., vn);
    vf.emplace<Foo>(v1, ..., vn);

About empty()/explicit operator bool() member functions

empty is more associated with containers. We don't see neither any nor optional as container classes. For probably valued types (as are the smart pointers and optional) the standard uses explicit operator bool conversion instead.
We consider any as a probably valued type.

Given

    struct Foo { Foo(int, double, char); }; 
    unique_ptr<Foo> pf=... 
    optional<Foo> of=...; 
    any af=...;

Before:

    if (pf) ...
    if (of) ...
    if ( ! af.empty()) ...

After:

    if (pf) ...
    if (of) ...
    if (af) ...
A lot of people consider that the `explicit operator bool` conversion is not explicit enough. An alternative to `explicit operator bool()` is to use a member function `has_value` (or `holds`).

After:

```cpp
if (pf.has_value()) ... 
if (of.has_value()) ... 
if (af.has_value()) ... 
```

The `has_value` member function is retained as more explicit and easy to read.

As this proposal is not about any change in pointe-like classes we lost uniform syntax respect to pointe-like classes. For `optional` we propose to have both.

After:

```cpp
if (pf) ... 
if (of) ... 
if (of.has_value()) ... 
if (af.has_value()) ... 
```

Having a uniform interface for pointe-like, type-erased and sum type classes should be the subject of another proposal. This is because there are other function for which the interfaces are not uniform.

### About `clear()` / `reset()` member functions

`clear()` is more associated to containers. We don't see neither `any` nor `optional` as container classes. For probably valued types (as are the smart pointers) the standard uses `reset` instead.

Given

```cpp
struct Foo { Foo(int, double, char); }; 
unique_ptr<Foo> pf=...; 
optional<Foo> of=...; 
any af=...; 
```

Before:

```cpp
pf.reset(); 
of = nullopt; 
af.clear(); 
```

After:

```cpp
pf.reset(); 
of.reset(); 
af.reset(); 
```

### About a `not-a-value` any: none

The original proposal contained a `none_t/none` for `any`. It has been considered that we have too much unit types and that another paper should take care of a more generic `none` separately.
Do we need an explicit make\_any factory?

any is not a generic type but a type erased type. any play the same role than a possible make\_any.

This paper however propose a make\_any factory for the emplace case, see below.

Note also that if [N4471] is adopted we wouldn't need any more make\_optional, as e.g. optional(1) would be deduced as optional<int>.

About emplace factories

However, we could consider a make\_xxx factory that in place constructs a T.

optional<T> and any could be in place constructed as follows:

```cpp
optional<T> opt(in_place, v1, vn);
f(optional<T>(in_place, v1, vn));

any a(in_place<T>, v1, vn);
f(any(in_place<T>, v1, vn));
```

When we use auto things change a little bit

```cpp
auto opt = optional<T>(in_place, v1, vn);
auto a = any(in_place<T>, v1, vn);
```

This is almost uniform. However having an make\_xxx factory function would make the code even more uniform

```cpp
auto opt = make_optional<T>(v1, vn);
f(make_optional<T>(v1, vn));

auto a = make_any<T>(v1, vn);
f(make_any<T>(v1, vn));
```

The implementation of these emplace factories could be:

```cpp
template <class T, class ...Args>
optional<T> make_optional(Args&& ...args) {
    return optional(in_place, std::forward<Args>(args)...);
}

template <class T, class ...Args>
any make_any(Args&& ...args) {
    return any(in_place<T>, std::forward<Args>(args)...);
}
```

Given

```cpp
struct Foo { Foo(int, double, char); };
```

Before:

```cpp
auto up = make_unique<Foo>(v1, ..., vn)
auto sp = make_shared<Foo>(v1, ..., vn)
auto o = optional<Foo>(in_place, v1, ..., vn)
```
auto a = any(Foo(v1, ..., vn))

After:
auto up = make_unique<Foo>(v1, ..., vn)
auto sp = make_shared<Foo>(v1, ..., vn)
auto o = make_optional<Foo>(v1, ..., vn)
auto a = make_any<Foo>(v1, ..., vn)

Which file for `in_place_t` and `in_place`?

As `in_place_t` and `in_place` are used by `optional` and `any` we need to move its definition to another file. The preference of the authors will be to place them in `<experimental/utility>.

Note that `in_place` could also be used by `experimental::variant` and that in this case it could also take an index as template parameter.

Access interface

The original paper suggested a possible interface for sum types access. As the subject is quite contentious, another paper could take care of it separately.

Open points

The authors would like to have an answer to the following points if there is yet at all an interest in this proposal:

• Are the differences in behavior of the new `in_place_t` acceptable?
• Where to place `in_place_t/in_place`? `<experimental/utility>`?
• Do we prefer `has_value/holds`?

Technical Specification

The wording is relative to [N4480].

General utilities library

Add in `[utility/synop]`

```cpp
struct in_place_tag {};
using in_place_t = in_place_tag(&)(unspecified);
template <class T>
using in_place_type_t = in_place_tag(&)(unspecified<T>);
template <int N>
using in_place_index_t = in_place_tag(&)(unspecified<N>);
constexpr in_place_t in_place(unspecified);
template <class ...T>;
```
Optional objects

Remove \texttt{in\_place\_t/in\_place} from \texttt{[optional/synop]} and \texttt{[optional/inplace]}

Update \texttt{[optional.synopsis]} adding after \texttt{make\_optional}

\begin{verbatim}
    template <class T, class ...Args>
    optional<T> make_optional(Args&& ...args);

    template <class T, class U, class ...Args>
    optional<T> make_optional(initializer_list<U> il, Args&& ...args);
\end{verbatim}

Add in \texttt{[optional.object]}

\begin{verbatim}
    void reset() noexcept;
\end{verbatim}

\textit{Effects}: If \texttt{*this} contains a value, calls \texttt{val\rightarrow T::~T()} to destroy the contained value; otherwise no effect.

\textit{Returns}: \texttt{*this}.

\textit{Postconditions}: \texttt{*this} does not contain a value.

\begin{verbatim}
    constexpr bool has_value() const noexcept;
\end{verbatim}

\textit{Returns}: \texttt{true} if and only if \texttt{*this} contains a value.

\textit{Remarks}: This function shall be a \texttt{constexpr} function.

Add in \texttt{[optional.specalg]}

\begin{verbatim}
    template <class T, class ...Args>
    optional<T> make_optional(Args&& ...args);
\end{verbatim}

\textit{Returns}: \texttt{optional<T>(in\_place, std::forward(args)...).}

\begin{verbatim}
    template <class T, class U, class ...Args>
    optional<T> make_optional(initializer_list<U> il, Args&& ...args);
\end{verbatim}

\textit{Returns}: \texttt{optional<T>(in\_place, il, std::forward(args)...).}

\section*{Class \texttt{any}}

Update \texttt{[any.synopsis]} adding
template <class T, class ...Args>
any make_any(Args&& ...args);
template <class U, class T, class ...Args>
any make_any(initializer_list<U>, Args&& ...args);

Add constexpr on any default constructor
constexpr any() noexcept;

Add inside class any

// Constructors

template <class T, class ...Args>
any(in_place_type_t<T>, Args&& ...);
template <class T, class U, class... Args>
explicit any(in_place_type<T>, initializer_list<U>, Args&&...);

template <class T, class ...Args>
void emplace(Args&& ...);
template <class T, class U, class... Args>
void emplace(initializer_list<U>, Args&&...);

Replace inside class any
void clear() noexcept;
bool empty() const noexcept;

by
void reset() noexcept;
constexpr bool has_value() const noexcept;

and replace any use of empty() by ! has_value()

Add in [any/cons]
constexpr any() noexcept;

constexpr any() noexcept;

template <class T, class ...Args>
any(in_place_type_t<T>, Args&& ...);

Requires: is_constructible_v<T, Args&&...> is true.

Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the arguments std::forward<Args>(args)....

Postconditions: this contains a value of type T.

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

template <class T, class U, class ...Args>
any(in_place_type_t<T>, initializer_list<U> il, Args&& ...args);

Requires: is_constructible_v<T, initializer_list<U>&, Args&&...> is true.
Effects: Initializes the contained value as if direct-non-list-initializing an object of type T with the arguments il, std::forward<Args>(args)....

Postconditions: *this contains a value.

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

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

Add in [any/modifiers]

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

Requires: is_constructible_v(<T, Args&&> is true.

Effects: Calls this.reset(). Then initializes the contained value as if direct-non-list-initializing an object of type T with the arguments std::forward<Args>(args)....

Postconditions: this contains a value.

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

Remarks: If an exception is thrown during the call to T's constructor, *this does not contain a value, and the previous (if any) has been destroyed.

Add in [any.assign]

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

Requires: is_constructible<T, initializer_list<U>&, Args&&...>

Effects: Calls this->reset(). Then initializes the contained value as if direct-non-list-initializing an object of type T with the argument s il, std::forward(args)....

Postconditions: this contains a value.

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

Remarks: If an exception is thrown during the call to T's constructor, *this does not contain a value, and the previous (if any) has been destroyed.

The function shall not participate in overload resolution unless is_constructible_v<T, initializer_list<U>&, Args&&...> is true.

Replace in [any/modifier], clear by reset.

Replace in [any/observers], empty by has_value (reversing the meaning).

    constexpr bool has_value() const noexcept;
Returns:

true if *this contains an object, otherwise false.

Add in [any.nonmembers]

```cpp
template <class T, class ...Args>
any make_any(Args&& ...args);
```

**Returns:** any(in_place<T>, std::forward<Args>(args)...).

```cpp
template <class T, class U, class ...Args>
any make_any(initializer_list<U> il, Args&& ...args);
```

**Returns:** any(in_place<T>, il, std::forward<Args>(args)...).

## Acknowledgements

Thanks to Jeffrey Yasskin to encourage me to report these as possible issues of the TS,

Many thanks to Agustin Bergé K-Balo for the function reference idea to represent in_place tags overloads.

Thanks to Tony Van Eerd for championing this proposal during the C++ standard committee meetings and helping me to improve globally the paper. The comparative table in the appendix comes from him.

## References


http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4480.html

[N4542] N4542 - Variant: a type-safe union (v4)


[eggs-variant] eggs::variant

https://github.com/eggs-cpp/variant

[N4471] N4471 -Template parameter deduction for constructors (Rev 2)

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4471.html

[P0032R0] Homogeneous interface for variant, any and optional

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0032r0.pdf

## Appendix
### WITHOUT proposal

**in_place, emplace_type, emplace_index**

<table>
<thead>
<tr>
<th>WITHOUT proposal</th>
<th>WITH proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct Foo { Foo(int, double, char); };</td>
<td>struct Foo { Foo(int, double, char); };</td>
</tr>
<tr>
<td>optional&lt;Foo&gt; of(in_place, 0, 1.5);</td>
<td>optional&lt;Foo&gt; of(in_place, 0, 1.5);</td>
</tr>
<tr>
<td>variant&lt;int, Foo&gt; vf(emplace_type&lt;Foo&gt;, 0, 1.5);</td>
<td>variant&lt;int, Foo&gt; vf(in_place&lt;Foo&gt;, 0, 1.5);</td>
</tr>
<tr>
<td>any af[Foo{0, 1.5, 'c'}];</td>
<td>any af[Foo{0, 1.5, 'c'}];</td>
</tr>
</tbody>
</table>

**NOTE:** thus any currently does not support non move/copy-able

### WITH proposal

**any.emplace()**

<table>
<thead>
<tr>
<th>WITHOUT proposal</th>
<th>WITH proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>of.emplace(0, 1.5, 'c');</td>
<td>of.emplace(0, 1.5, 'c');</td>
</tr>
<tr>
<td>vf.emplace&lt;Foo&gt;(0, 1.5, 'c');</td>
<td>vf.emplace&lt;Foo&gt;(0, 1.5, 'c');</td>
</tr>
<tr>
<td>af = Foo{0, 1.5, 'c'};</td>
<td>af = Foo{0, 1.5, 'c'};</td>
</tr>
</tbody>
</table>

any does not currently emplace

### reset()

<table>
<thead>
<tr>
<th>WITHOUT proposal</th>
<th>WITH proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>unique_ptr&lt;Foo&gt; uf = new Foo{0, 1.5, 'c'};</td>
<td>unique_ptr&lt;Foo&gt; uf = new Foo{0, 1.5, 'c'};</td>
</tr>
<tr>
<td>uf.reset();</td>
<td>uf.reset();</td>
</tr>
<tr>
<td>of = nullopt;</td>
<td>of = nullopt;</td>
</tr>
<tr>
<td>af.clear();</td>
<td>af.clear();</td>
</tr>
</tbody>
</table>

### has_value()

<table>
<thead>
<tr>
<th>WITHOUT proposal</th>
<th>WITH proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (uf) ...</td>
<td>if (uf.has_value()) ...</td>
</tr>
<tr>
<td>if (of) ...</td>
<td>if (of.has_value()) ...</td>
</tr>
<tr>
<td>if (!af.empty()) ...</td>
<td>if (af.has_value()) ...</td>
</tr>
</tbody>
</table>

**NOTE:** smart-ptrs as well

### make...() factories

<table>
<thead>
<tr>
<th>WITHOUT proposal</th>
<th>WITH proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto uf = make_unique&lt;Foo&gt;{0, 1.5, 'c'};</td>
<td>auto uf = make_unique&lt;Foo&gt;{0, 1.5, 'c'};</td>
</tr>
<tr>
<td>auto sf = make_shared&lt;Foo&gt;{0, 1.5, 'c'};</td>
<td>auto sf = make_shared&lt;Foo&gt;{0, 1.5, 'c'};</td>
</tr>
<tr>
<td>auto of = make_optional&lt;Foo&gt;{Foo{0, 1.5, 'c'} };</td>
<td>auto of = make_optional&lt;Foo&gt;{Foo{0, 1.5, 'c'} };</td>
</tr>
<tr>
<td>auto af = any(Foo{0, 1.5, 'c'});</td>
<td>auto af = make_any&lt;Foo&gt;{0, 1.5, 'c'};</td>
</tr>
</tbody>
</table>

**NOTE:** EWG has mandated RVO so non move/copy-able also work

### constexpr any ctor

<table>
<thead>
<tr>
<th>WITHOUT proposal</th>
<th>WITH proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>any a;</td>
<td>constexpr any a;</td>
</tr>
</tbody>
</table>