Homogeneous interface for \texttt{variant}, \texttt{any} and \texttt{optional} (Revision 2)

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

History

Revision 2

The 2nd revision of \texttt{P0032R1} fixes some typos and takes in account the feedback from Jacksonville meeting. Next follows the direction of the committee: Adopt it for C++17 with the following strapools:

- Accept \texttt{.reset()}, \texttt{remove any.clear().leave optional=\_nullopt} ? SF F N A SA 6 9 1 0 0
- Add \texttt{.has\_value()} to \texttt{any} and \texttt{optional} (in addition to \texttt{optional}'s bool conversion)? SF F N A SA 3 7 3 3 0
- Add \texttt{.has\_value()} to smart pointers, including \texttt{unique\_ptr} and \texttt{shared\_ptr}; equivalent to operator bool? SF F N A SA 0 3 2 7 3

- Make \texttt{any::\_any()} (the default constructor) constexpr? (Alisdair raises warnings) SF F N A SA 2 4 9 0 1 (If implementations have significant problems, please tell us.)
- Change \texttt{make\_optional} to be like \texttt{make\_unique}? SF F N A SA 0 1 7 5 2
- Add \texttt{make\_any} SF F N A SA 2 4 5 4 0 Yes.
- Want to change everything to \texttt{in\_place}? SF F N A SA 4 6 3 2 2 Yes. (Send us the error messages, plz)
- Add \texttt{any\_emplace}? SF F N A SA 5 7 3 0 0
- Send the changes approved above to LWG for C++17?

Unanimous, with mention that \texttt{in\_place} might be instantiated into every object file.

Also check that \texttt{any\langle in\_place\langle Foo\rangle\rangle} stores \texttt{Foo\{}\texttt{, not in\_place\langle Foo\rangle}}.

This revision then mainly moves the wording from \texttt{std::experimental} to \texttt{std} and

- Add a mention that \texttt{in\_place} might be instantiated into every object file.
- Take in account the changes of variant after Kona and add the wording for \texttt{variant}.
- Added some examples of the code generated for \texttt{in\_place} without the proposal and with.
Revision 1

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

- Do we want to adopt the new \texttt{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 \texttt{in\_place} function reference. Nevertheless, the author doesn't know how to have the \texttt{in\_place} both for \texttt{optional}, \texttt{any} and \texttt{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 \texttt{in\_place} | 13 |
| Both are emplace | 6 |

- Do we want to adopt the new \texttt{in\_place} definition?

SF F N A SA
1 3 8 0 0

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

Do we want emptiness checking to be consistent between \texttt{any} / \texttt{optional}? Unanimous yes

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

- Do we want the \texttt{not-a-value \texttt{none}}? No, too much unit types. The committee wants a separated paper for a generic \texttt{none\_t} / \texttt{none}?

- Do we want \texttt{none\_t} to be a separate paper?

SF F N A SA
11 1 3 0 0

- Do we want the \texttt{make\_any} factory? Yes
Do we want to have a follow up for a concept based on the functions `holds` and `storage_address_of`? Not in this paper.

Do we want to have a follow up for `select<T> / select<I>`? Not in this paper. Considered as invention.

Do we want to have a follow up for the observers `reference_of`, `value_of` and `address_of`? Not in this paper.

**Other modifications**

- Added a section in the design rationale describing the differences between the new and current `in_place`.
- Improved the wording and in particular added some missing overloads using `initializer_list`.
- Added `constexpr` for `has_value`.
- Added a comparative table on the appendix also.

**Introduction**

This paper identifies some differences in the design of `variant<Ts...>`, `any` and `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 [N4562] and the variant proposal [P0088R1] and concerns mainly:

- coherency of functions that behave the same but that are named differently,
- replace the `in_place` tag by a function with overloads for type and index,
- replacement of `in_place_type<T> / in_place_index<I>` by `in_place<T> / in_place<I>`.
- addition of emplace factories for `any` and `optional` classes.

**Motivation and Scope**

Both `optional` and `any` are classes that can store possibly some underlying type. In the case of `optional` the underlying type is know at compile time, for `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 `Ts` or a not-a-value type, know at run-time. Let me refer to this possible variant of `nullable_variant <Ts...>` The following inconsistencies have been identified:

- `variant<Ts...>` and 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>`
  - Remove the definition of `in_place_type_t<T> / in_place_index_t<I>`.
  - Replace the uses (if any) of `in_place_type<T> / in_place_index_t<I>` by `in_place<T> / in_place<I>`, respectively.

**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` cannot 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.
template <class T, class ... Args>
    any(in_place_type_t<T>, Args&& ...);

This can be used as

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

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

```
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`

```
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

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

Before:

```
optional<Foo> of(in_place, 0, 1.5, 'c');
variant<int, Foo> vf(in_place_type<Foo>, 0, 1.5, 'c');
variant<int, Foo> vf(in_place_index<1>, 0, 1.5, 'c');
any af(Foo(0, 1.5, 'c')); // (*)
```

After:

```
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<1>, 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, however some measure are needed.

We have done some measures and when the functions having these tags are inlined, there is no difference as the compiler removes the call. However when the function is not inlined we see a difference without the proposal there is a push while with the proposal there is a move.

All the measure have been done -std=c++14 -O3.
It is up to the committee to decide if the difference is significant or not.

**Possible malicious attacks**

Unfortunately using function references would work for any unary function taken the unspecified type and returning \texttt{in\_place\_tag} in addition to \texttt{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 \texttt{optional<T>} to the new \texttt{in\_place\_t} type we found that we cannot anymore use \texttt{in\_place\_t{}}. The authors don't consider this a big limitation as the user can use \texttt{in\_place} instead. It needs to be noted that this is in line with the behavior of \texttt{nullopt\_t} as \texttt{nullopt\_t{}} fails as no default constructible. However \texttt{nullptr\_t{}} seems to be well formed.

**Not assignable from \{\}**

After a deeper analysis we found also that the old \texttt{in\_place\_t} supported \texttt{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 \texttt{in\_place} instead.

```c
in\_place\_t t;
t = in\_place;
```

It needs to be noted that this is in line with the behavior of \texttt{nullopt\_t} as the following compile fails.

```c
nullopt\_t t = {}; // compile fails
```

However \texttt{nullptr\_t} seems to be support it.

```c
nullptr\_t t = {}; // compile pass
```

To re-enforce this design, there is an pending issue 2510-Tag types should not be \texttt{DefaultConstructible} \cite{Core issue 2510}.

**emplace forward member function**

\texttt{emplace} member function emplaces implicitly a \texttt{T}.

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

\texttt{emplace} for \texttt{any} cannot have an implicit type \texttt{T}. We need a way to state explicitly which \texttt{T} must be emplaced.

```c
template <class T, class ...Args>
any::emplace(Args&& ...);
```
and used as follows

```cpp
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

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

Before:

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

After:

```cpp
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();
```

### Do we need an explicit `make_any` factory?

`any` is not a generic type but a type-erased type. `any` play the same role as a possible `make_any`. This paper however propose a `make_any` factory for the emplace case, see below. Note also that if P0091R0 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 as simple as:
template <class T, class ...Args>
optional<T> make_optional(Args&& ...args) {
    return optional<T>(in_place, std::forward<Args>(args)...);
}

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

Given c++
struct Foo { Foo(int, double, char););

Before:
auto up = make_unique<Foo>(v1, ..., vn)
auto sp = make_shared<Foo>(v1, ..., vn)
auto o = optional<Foo>(in_place, v1, ..., vn)

After:
auto a = any(Foo(v1, ..., vn))
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 <utility>.

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

Open points

None.

Proposed wording

The wording is relative to N4562.

General utilities library

Add in [utility/synop]
namespace std {

    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_tag in_place(unspecified);
    template <class T>
        constexpr in_place_tag in_place(unspecified<T>);
    template <size N>
        constexpr in_place_tag in_place(unspecified<N>);
}

in_place function might be instantiated into every object file.

Optional objects

Remove in_place_t / in_place from [optional/synop] and [optional/inplace]

Update [optional.synopis] adding after make_optional

namespace std {
    [...] 
    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);
    [...] 
}

Add in [optional.object]

void reset() noexcept;

Effects: If *this contains a value, calls val->T::~T() to destroy the contained value; otherwise no effect.

Returns: *this.

Postconditions: *this does not contain a value.

constexpr bool has_value() const noexcept;

Returns: true if and only if *this contains a value.

Remarks: This function shall be a constexpr function.

Add in [optional.specalg]

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

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

Returns: `optional<T>(in_place, il, std::forward(args)...)`.

Class `any`

Update [any synopsis] adding

```cpp
namespace std {

    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);

} // std
```

Add constexpr on `any` default constructor

```cpp
constexpr any() noexcept;
```

Add inside class `any`

```cpp
// 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<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`

```cpp
void clear() noexcept;
bool empty() const noexcept;
```

by

```cpp
void reset() noexcept;
constexpr bool has_value() const noexcept;
```

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

Add in [any/cons]

```cpp
constexpr any() noexcept;
    template <class T, class ...Args>
        any(in_place_type_t<T>, Args&& ...);
```
Requires: \texttt{is\_constructible\_v<T, Args&&...> is true}.

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

Postconditions: this contains a value of type \texttt{T}.

Throws: Any exception thrown by the selected constructor of \texttt{T}.

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

Requires: \texttt{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 \texttt{T} with the arguments 
\texttt{il, std::forward<Args>(args)....}

Postconditions: *this contains a value.

Throws: Any exception thrown by the selected constructor of \texttt{T}.

Remarks: The function shall not participate in overload resolution unless 
\texttt{is\_constructible\_v<T, initializer_list<U>&, Args&&...> is true}.

Add in \texttt{[any/modifiers]}

\begin{verbatim}
  template <class T, class ...Args>
  void emplace(Args&& ...);
\end{verbatim}

Requires: \texttt{is\_constructible\_v<T, Args&&> is true}.

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

Postconditions: *this contains a value.

 Throws: Any exception thrown by the selected constructor of \texttt{T}.

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

Add in \texttt{[any.assign]}

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

Requires: \texttt{is\_constructible\_v<T, initializer_list<U>&, Args&&...>}.

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

Postconditions: *this contains a value.

Throws: Any exception thrown by the selected constructor of \texttt{T}.

Remarks: If an exception is thrown during the call to \texttt{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 \texttt{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).

```cpp
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)...).

Class variant

Remove in_place_type_t / in_place_type / in_place_index_t / in_place_index from [variant/synop].

Replace [variant.emplaced]

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 and its valuable comments.

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

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

- N4562 Working Draft, C++ Extensions for Library Fundamentals
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4562.html

- P0032R0 Homogeneous interface for variant, any and optional
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0032r0.pdf

- P0032R1 Homogeneous interface for variant, any and optional

- [P0088R1] Variant: a type-safe union that is rarely invalid (v5)

- P0091R0 Template parameter deduction for constructors (Rev 3)
Core issue 2510: Tag types should not be DefaultConstructible

Appendix
### WITHOUT proposal

#### in_place, in_place_type, in_place_index

```cpp
def struct Foo { Foo(int, double, char); }
def optional<Foo> of(in_place, 0, 1.5, 'c');
def variant<int, Foo> vf(in_place_type<Foo>, 0, 1.5, 'c');
def variant<int, Foo> vf(in_place_index<1>, 0, 1.5, 'c');
def any af(Foo(0, 1.5, 'c'));
```

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

### WITH proposal

#### in_place

```cpp
def struct Foo { Foo(int, double, char); }
def optional<Foo> of(in_place, 0, 1.5, 'c');
def variant<int, Foo> vf(in_place_type<Foo>, 0, 1.5, 'c');
def variant<int, Foo> vf(in_place_index<1>, 0, 1.5, 'c');
def any af(in_place<Foo>, 0, 1.5, 'c');
```

Also, now any supports non move/copy-able

### any.emplace()

```cpp
def of.emplace(0, 1.5, 'c');
def vf.emplace<Foo>(0, 1.5, 'c');
def vf.emplace<1>( 0, 1.5, 'c');
def af.emplace<Foo>(0, 1.5, 'c');
```

Any does not currentlyemplace

### reset()

```cpp
def unique_ptr<Foo> uf = new Foo(0, 1.5, 'c');
deuf.reset();
deof = nullopt;
deaf.clear();
```

### has_value()

```cpp
if (uf) ...  
if (of) ...  
if (!af.empty()) ... 
```

### make...() factories

```cpp
auto uf = make_unique<Foo>(0, 1.5, 'c');
auto sf = make_shared<Foo>(0, 1.5, 'c');
auto of = make_optional<Foo>(Foo(0, 1.5, 'c'));
auto af = any(Foo(0, 1.5, 'c'));
```

### constexpr any ctor

```cpp
any a;
```

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