Homogeneous interface for `variant`, `any` and `optional` (Revision 3)

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.

History

Revision 3

Fixes some issues found during the LWG review of the wording.

- Calling `in_place` function results in undefined behavior.

- `any` is not a literal type so except his default constructor no other function can be `constexpr`.

Revision 2

The 2nd revision of [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 .reset(), remove any.clear(), leave optional=nullopt?

- Add .has_value() to any and optional (in addition to optional's bool conversion?)

- Add .has_value() to smart pointers, including unique_ptr and shared_ptr; equivalent to operator bool?

- Make any::any() (the default constructor) constexpr? (Alisdair raises warnings)

(If implementations have significant problems, please tell us.)

- Change make_optional to be like make_unique?

- Add make_any

Yes.

- Want to change everything to in_place?

Yes. (Send us the error messages, plz)

- Add anyemplace?

- Send the changes approved above to LWG for C++17?

Unanimous, with mention that in_place might be instantiated into every object file.
Also check that \texttt{any(in\_place<	extit{Foo}>) stores \	extit{Foo}{}}, not \texttt{in\_place<	extit{Foo}>}.

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.
- Added reference to \texttt{Core issue 2510}.

\section*{Revision 1}

The 1st revision of \texttt{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 \texttt{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.

\begin{itemize}
  \item Leave \texttt{optional} different from \texttt{variant} and \texttt{any} 6
  \item Member function is \texttt{emplace}; tag type is \texttt{in\_place} 13
  \item Both are \texttt{emplace} 6
\end{itemize}

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

\begin{tabular}{lll}
\hline
SF & FN & SA  \\
13800 &  \\
\hline
\end{tabular}

- Do we want in place constructor for \texttt{any}? Unanimous Yes.

- Do we want the clear and reset changes? Yes

How to empty an \texttt{any} or \texttt{optional}?

\begin{itemize}
  \item \texttt{.reset()} 12
  \item \texttt{.clear()} 7
  \item \texttt{=none (different paper)} 7
  \item \texttt{=\{} 5
  \item \texttt{.drain()} 1
\end{itemize}
• 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

<table>
<thead>
<tr>
<th>Provide operator bool for both</th>
<th>Y: 6  N: 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provide <code>something()</code></td>
<td>Y: 17  N: 0</td>
</tr>
<tr>
<td>Provide <code>=={}</code></td>
<td>Y: 0   N: 5</td>
</tr>
<tr>
<td>Provide <code>==std::none</code></td>
<td>Y: 5   N: 2</td>
</tr>
<tr>
<td><code>something(any/optional)</code></td>
<td>Y: 3   N: 8</td>
</tr>
</tbody>
</table>

• 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`.

• Do we want `none_t` to be a separate paper?

SF F N A SA
11 1 3 0 0

• Do we want the `make_any` factory? Yes

SF F N A SA
19 7 2 0

• 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 contain 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<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`. 
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.

```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` 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(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:

```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<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 \texttt{in\_place\_t} and the old one

Cost of function reference versus tags

The proposed function reference for \texttt{in\_place\_t(&)(unspecified)} takes the size of an address while the previous \texttt{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.

<table>
<thead>
<tr>
<th>Conf</th>
<th>WITHOUT proposal</th>
<th>WITH proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 gcc 5.3.0</td>
<td>pushq $0 call g1(in_place_t)</td>
<td>movl in_place(in_place_unspecified), \texttt{edi} call g2(in_place_tag (&amp;)(in_place_unspecified))</td>
</tr>
<tr>
<td>x86 cmang 3.7.1</td>
<td>pushq %rax callq g1(in_place_t)</td>
<td>movl in_place(in_place_unspecified), \texttt{edi} callq g2(in_place_tag (&amp;)(in_place_unspecified))</td>
</tr>
</tbody>
</table>

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 `optional<T>` to the new `in_place_t` type we found that we cannot 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 `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;
t = in_place;
```

It needs to be noted that this is in line with the behavior of `nullopt_t` as the following compile fails.

```cpp
nullopt_t t = {}; // compile fails
```

However `nullptr_t{}` seems to support it.

```cpp
nullptr_t t = {}; // compile pass
```

To re-enforce this design, there is an pending issue 2510-Tag types should not be `DefaultConstructible` Core issue 2510.

**emplace forward member function**

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

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

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

```cpp
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
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:

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:

```cpp
#include <optional>
#include <any>

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)
```

After:

```cpp
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 {
        in_place_tag() = delete;
    };
    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>);
    
    in_place_tag in_place(unspecified);
    template <class T>
        in_place_tag in_place(unspecified<T>);
    template <size N>
        in_place_tag in_place(unspecified<N>);
    [...]
}

Add a section [utility/in_place]

20.2.x In-place construction [utility.inplace]

The in_place_t/in_place_type_t/in_place_index_t function types are used as unique types to disambiguate constructor and function overloading. Specifically, optional has a constructor with in_place_t as the first parameter followed by a parameter pack; this indicates that T should be constructed in-place (as if by a call to a placement new expression) with the forwarded pack expansion as arguments for the initialization of T.

*Remark:* Calling in_place functions results in undefined behavior. [Note: These functions might be instantiated into every object file. – end note]

Optional objects

Remove in_place_t/in_place from [optional/synop].
namespace std {
  // 20.6.3, optional for object types
  template <class T> optional;
  // 20.6.4, in-place construction
  struct in_place_t{}
  constexpr in_place_t in_place{};
  [...]

  Update [optional.synopsis] adding after make_optional.

  namespace std {
    [...]
    template <class T, class ...Args>
      constexpr optional<T> make_optional(Args&& ...args);
    template <class T, class U, class ...Args>
      constexpr optional<T> make_optional(initializer_list<U> il, Args&& ...args);
    [...]
  }

  Add a section in [optional.object.modifier]

  20.6.3.6 Modifiers
  void reset() noexcept;

  Effects: If *this contains a value, calls val->T::~T() to destroy the contained value;
           otherwise no effect.
  Postconditions: *this does not contain a value.
                   constexpr bool has_value() const noexcept;

  Returns: true if and only if *this contains a value.
  Remark: This function shall be a constexpr function.

  Remove section [optional/inplace].

  Add in [optional.specalg]
  template <class T, class ...Args>
    constexpr optional<T> make_optional(Args&& ...args);

  Effects: Equivalent to: return optional<T>(in_place, std::forward<Args>(args)...).
  template <class T, class U, class ...Args>
    constexpr optional<T> make_optional(initializer_list<U> il, Args&& ...args);
**Effects:** Equivalent to: `return optional<T>(in_place, il, std::forward<Args>(args))`.

**Class** any

Add a note.

[Note any is a not a literal type --end note]

Update

An object of class any stores an instance of any type that satisfies the constructor requirements or is empty, it has no value, and this is referred to as the state of the class any object. The stored instance is called the contained object. Two states are equivalent if they are either both empty or if both are not empty and if either they both have no value, or both have a value and the contained objects are equivalent.

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);

    [...]
}
```

Update constexpr on any default constructor

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

Add inside class any

```cpp
// Constructors
```

```cpp
    template <class T, class ...Args>
    explicit 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

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

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

**Update in [any/cons]**

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

**Add in [any/cons]**

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

```cpp
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.

**Update [any.cons]**

```cpp
~any();
```

**Effects**: As if `clear reset()`.

**Add in [any/modifiers]**

```cpp
template <class T, class ...Args>
void emplace(Args&& ... 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]`

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

**Requires**: is_constructible_v<T, initializer_list<U>&, 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 \(il, \text{std::forward}(\text{Args}) (\text{args})\)....

**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/modifiers]`

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

**Effect**: If not empty, destroys the contained object.

**Postcondition**: empty() is true.

by

```cpp
void reset() const noexcept;
```

**Effect**: If has a contained object, destroys the contained object.

**Postcondition**: has_value() is false.

Replace in `[any/observers]`

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

**Returns**: true if *this has no contained object, otherwise false.
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);
```

_Effect: Equivalent to:_ return 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);
```

_Effect: Equivalent to:_ return 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].

### 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.

Thanks to the LWG for its careful reading.

### References

- [eggs-variant](https://github.com/eggs-cpp/variant)

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)
  
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0091r0.html

- **Core issue 2510** Tag types should not be DefaultConstructible
  
  http://cplusplus.github.io/LWG/lwg-active.html#2510

## Appendix

<table>
<thead>
<tr>
<th>WITHOUT proposal</th>
<th>WITH proposal</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>in_place, in_place_type, in_place_index</strong></td>
<td><strong>in_place</strong></td>
</tr>
<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, 'c');</td>
<td>optional&lt;Foo&gt; of(in_place, 0, 1.5, 'c');</td>
</tr>
<tr>
<td>variant&lt;int, Foo&gt; vf(in_place_type&lt;Foo&gt;, 0, 1.5, 'c');</td>
<td>variant&lt;int, Foo&gt; vf(in_place&lt;Foo&gt;, 0, 1.5, 'c');</td>
</tr>
<tr>
<td>variant&lt;int, Foo&gt; vf(in_place_index&lt;1&gt;, 0, 1.5, 'c');</td>
<td>variant&lt;int, Foo&gt; vf(in_place&lt;1&gt;, 0, 1.5, 'c');</td>
</tr>
<tr>
<td>any af(Foo{0, 1.5, 'c'});</td>
<td>any af(in_place&lt;Foo&gt;, 0, 1.5, 'c');</td>
</tr>
<tr>
<td>NOTE: thus any currently does not support non move/copy-able</td>
<td>Also, now any supports non move/copy-able</td>
</tr>
<tr>
<td>any.emplace()</td>
<td></td>
</tr>
</tbody>
</table>

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

Any.emplace()
```cpp
of.emplace(0, 1.5, 'c');
vf.emplace<Foo>(0, 1.5, 'c');
vf.emplace<1>(0, 1.5, 'c');
af = Foo{0, 1.5, 'c'};
```

any does not currently emplace

Now any supports non move/copy-able

### reset()

```cpp
unique_ptr<Foo> uf = new Foo(0, 1.5, 'c');
uf.reset();
of = nullopt;
af.clear();
```

variant? No. Does not go empty. Could default-construct have has_value(). Don’t force false consistency.

### has_value()

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

NOTE: smart-ptrs as well variant? – No. intentionally “corrupted_by_exception”

### 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'});
```

auto uf = make_unique<Foo>(0, 1.5, 'c');
auto sf = make_shared<Foo>(0, 1.5, 'c');
auto of = make_optional<Foo>(0, 1.5, 'c');
auto af = make_any<Foo>(0, 1.5, 'c');
<table>
<thead>
<tr>
<th>constexpr any ctor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>any a;</td>
<td>any a; // (at namespace scope) constant initialization</td>
</tr>
</tbody>
</table>

NOTE: EWG has mandated RVO so non move/copy