Document number:	P0032R3
Date:	2016–05-24
Project:	ISO/IEC JTC1 SC22 WG21 Programming Language C++
Audience:	Library Evolution Working Group
Reply-to:	Vicente J. Botet EscribEscribá <vicente.botet@nokia.com>

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`?

```
SF F N A SA
6  9 1 0 0
```

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

```
SF F N A SA
3  7 3 3 0
```

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

```
SF F N A SA
0  3 2 7 3
```

- Make `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 `make_optional` to be like `make_unique`?

```
SF F N A SA
0  1 7 5 2
```

- Add `make_any`

```
SF F N A SA
2  4 5 4 0
```

Yes.

- Want to change everything to `in_place`?

```
SF F N A SA
4  6 3 2 2
```

Yes. (Send us the error messages, plz)

- Add `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 `in_place` might be instantiated into every object file.

Also check that `any(in_place<Foo>)` stores `Foo{}`, not `in_place<Foo>`.
This revision then mainly moves the wording from `std::experimental` to `std` and

- Add a mention that `in_place` might be instantiated into every object file.
- Take in account the changes of variant after Kona and add the wording for `variant`.
- Added some examples of the code generated for `in_place` without the proposal and with.

- Added reference to Core issue 2510.

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 `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 doesn'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 ==std::nullopt          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`.
- 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

 1 9 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 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`.

```
    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.

```
      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 needs 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.

Conf	WITHOUT proposal	WITH proposal
x86 gcc 5.3.0	`pushq $0` `call g1(in_place_t)`	`movl in_place(in_place_unspecified),` `call g2(in_place_tag (&)(in_place_uns`
x86 cmang 3.7.1	`pushq %rax` `callq g1(in_place_t)`	`movl in_place(in_place_unspecified),` `callq g2(in_place_tag (&)(in_place_uns`

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`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 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.

```
      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.

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

However `nullptr_t` seems to be support it.

```
      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`.

```
      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.

```
      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:

```
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:

```
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

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

Before:

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

After:

```
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:

```
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:

```
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

```
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

```
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

```
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

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

Add inside class `any`

```
  // Constructors
  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`

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

by

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

Update in [any/cons]

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

Add in [any/cons]

```
  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`.

```
  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]

```
  ~any();
```

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

Add in [any/modifiers]

```
    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]

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

Replace in [any/modifiers]

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

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

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

by

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

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

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

Replace in [any/observers]

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

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

by

```
    bool has_value() const noexcept;
```

Returns: `true` if `*this` contains an object, otherwise `false`.

Add in [any.nonmembers]

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

Effect: Equivalent to: `return any(in_place<T>, std::forward<Args>(args)...).`

```
  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 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

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0032r1.pdf

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

  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0088r1.pdf

- 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

WITHOUT proposal	WITH proposal
in_place, in_place_type, in_place_index	**in_place**

`struct Foo { Foo(int, double, char); };`
 	
`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'});`
 	
NOTE: thus any currently does not support non move/copy-able
	`struct Foo { Foo(int, double, char); };`
	`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`);`
	
	Also, now any supports non move/copy-able

any.emplace()	

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

any does not currently emplace	Now any supports non move/copy-able
reset()	

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

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

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

	NOTE: smart-ptrs as well variant? – No. intentiona “corrupted_by_exception”
make_...() factories	

`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’);`

	NOTE: EWG has mandated RVO so non move/cop
constexpr any ctor	
`any a;`	`any a; // (at namespace scope) constant initializ`