Document number: D0032
Date: 2015–09-13
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

On variant<Ts...>, any and optional<T> Coherency

This paper identifies some minor inconsistencies 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 (and thus the inconsistencies).

Contents

Introduction

This paper identifies some minor inconsistencies in the design of variant<Ts...>, any and optional, diagnoses them as owing to unnecessary asymmetry between those classes, and proposes wording to eliminate the asymmetry (and thus the inconsistencies).

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 `in_place` tag by a function. Add overloads for type and index.
- replacement of `emplace_type<T>`/`emplace_index<I>` by `in_place<T>`/`in_place<I>`
- addition of emplace factories for `any` and `optional` classes.
- replacement of the proposed variant `get` interface and the `any any_cast`, by `sum_cast.`
- replacement of `bad_optional_access` by `bad_optional_cast`
- replacement of `bad_variant_access` by `bad_optional_cast`
- make `bad_optional_cast` and `bad_variant_cast` inherit from `bad_cast`

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 been nullable, 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 `variant optional<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.
- `any` provides `any::clear()` to unset the value while `optional` uses assignment from a `nullopt_t`.
- `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, `variant` uses a tuple like interface, while `any` uses a cast like interface. As all these classes are in someway sum types, the first two limited and know at compile time, the last unlimited, it seems natural that both provide the same kind of interface. In addition it seems natural that the exception thrown when the access/cast fails inherits from a common exception `bad_cast`.

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` by an overloaded function (see [eggs-variant]).
- In class `optional<T>`
  - Add a `reset` member function.
- Add an `optional_cast` factory.
- Replace `bad_optional_access` by `bad_optional_cast` and make it inherit from `bad_cast`.
- Add an `emplace_optional` factory.
- In class `any`
  - make the default constructor constexpr,
  - add in place forward constructors,
  - add emplace forward member functions,
  - rename the `empty` function with an `explicit bool` conversion,
  - rename the `clear` member function to `reset`,
- Add a `none` constexpr variable of type `any`.
- Add an `emplace_any` factory.
- In class `variant<T>`
  - Replace the uses of `emplace_type_t<T>`/`emplace_index_t<I>` by `in_place_t (&)(unspecified<T>)`/`in_place_t (&)(unspecified<I>)`
  - Replace the uses of `emplace_type<T>`/`emplace_index<I>` by `in_place<T>`/`in_place<I>`
  - If `variant<Ts...>` proposal ends been possibly empty,
    - Add a `reset` member function.
    - Add an `explicit bool` conversion
  - Replace the `get<T>(variant<Ts...>)` by `variant_cast<T>(variant<Ts...>)`.
  - Replace the `get<I>(variant<Ts...>)` by `variant_cast<I>(variant<Ts...>)`
  - Replace the `get<T>(variant<Ts...>*)` by `variant_cast<T>(variant<Ts...>*)`.
  - Replace the `get<I>(variant<Ts...>*)` by `variant_cast<I>(variant<Ts...>*)`
- Replace `bad_variant_access` by `bad_variant_cast` and make it inherit from `bad_cast`.

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 can not have an implicit type T. We need a way to state explicitly which T must be constructed in place. The function in_place_t(&)(unspecified<T>) is used to convey the type T participating in overload resolution.

```
template <class T, class ...Args>
any(in_place_t(&)(unspecified<T>), , Args&& ...);
```

This can be used as

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

where

```
template <class T>

in_place_t in_place(unspecified<T>) { return {} };
```

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

```
in_place_t in_place(unspecified) { return {} };
```

and

```
template <class... Args>
constexpr explicit optional<T>::optional(

      in_place_t (&)(unspecified), Args&&... args);
```

Fortunately using function references would work for any unary function taken the unspecified type and returning in_place_t 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.

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

```
template <int N>

in_place_t in_place(unspecified<N>) { return {} };
```

Given

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

Before:

```
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(in_place<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');
```

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 to containers. We don't see neither `any`, `variant` 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 . If `variant` end modeling a probably valued type both should provide the `explicit operator bool`.

Given

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

Before:

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

After:

```
if (pf) ...
if (of) ...
if (vf) ...
if (af) ...
```

An alternative to `explicit operator bool()` is to use a member function `has_value`.

After:

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

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

`clear` is more associated to containers. We don't see neither `any`, `variant` 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=...;
optionals<int, Foo> vf=...
any af=...;
```

Before:

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

After:

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

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

`nullptr`, `nullopt` represent not-a-value for pointer-like types and to `optional` respectively.

`any` default destructor, as is the case for `optional` and smart pointers default constructor results in an `any` that doesn't contain any value, not-a-value

```
any a = 1;
```

```
a = any{};
```

However, the authors think that using a specific `none` constant to mean not-a-value for any is much more explicit

```
any a = 1;
a = none;
```

The advantage of having a specific type to mean not-a-value for any is that the construction and assignment of any from this type can be optimized by the compiler.

Given

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

Before:

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

After:

```
pf = nullptr;
of = nullopt;
af = none;
```

About a not-a-value optionals

It is too soon to add a not-a-variant value for variants, but if we end with variant that could be seen as a nullable type (a probably valued type), we believe that we should have an explicit not-a-value constant. We don't have yet a good proposal, let me call it nav.

```
optionals<int, string> a = 1;
a = nav;
```

Which type for `none`?

Two possibilities: using a constexpr as it is the case of `nullopt`

```
struct none_t {};
constexpr none_t none;
```

or using a function reference like the proposed `in_place` tag

```
struct none_tag_t {};
none_tag_t (&none_t)(unspecified);
none_t none(unspecified) { return none_t{}; }
```

About a nav_t type implicitly convertible to any, variant<Ts..> and optional

An alternative to the reset member function would be to be able to assign a nav to an optional, a variant or an any. We could consider that a default instance of any contains an instance of a nav_t type, as optional contains a nullopt.

The problem is that then nav can be seen as an optional or an any and could result in possible ambiguity.

We think that this implicit conversions go against the raison d'être of nullptr and that we need explicit factories/constants.

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 is way this paper doesn't propose a make_any factory.

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 an emplace_xxx factory that in place constructs a T.

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

```
optional<T> opt(in_place_t(&)(unspecified), v1, vn);
f(optional<T>(in_place, v1, vn));

any a(in_place_t(&)(unspecified<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 emplace_xxx factory function would make the code even more uniform

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

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

The implementation of these emplace factories could be:

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

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

Given

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

Before:

```
      auto o = optional<Foo>(in_place, v1, ..., vn)
      auto a = any(Foo{v1, ..., vn})
```

After:

```
      auto o = emplace_optional<Foo>(v1, ..., vn)
      auto a = emplace_any<Foo>(v1, ..., vn)
```

Getters versus cast

The generic `get<T>(t)` and `get<I>(t)` are convenient for product types as we know that the product type will contain an instance of any one of its parts. However, `any`, `optional` and `variant` can be seen as sum types, and so we are not sure the sum type stores the request type. `any` uses `any_cast`, `variant` uses `get` and `optional` uses value.

We propose that a single name for all the sum types (even for `optional`), e.g. `sum_cast`. This will be a customization point and the user should be able to overload these functions.

Given

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

Before:

```
      auto& xo = of.value();
      auto& xv = get<1>(vf);
      auto& xv = get<Foo>(vf);
      auto& xa = any_cast<Foo>(af);
```

After:

```
      auto& xo2 = sum_cast<0>(of);
      auto& xo2 = sum_cast<Foo>(of);
      auto& xv = sum_cast<1>(vf);
      auto& xv = sum_cast<Foo>(vf);
      auto& xa = sum_cast<Foo>(af);
```

Moving to a cast like interface goes together with changing of `bad_xxx_access` to `bad_xxx_cast` both for optional and variant. It seems natural that all these `bad_xxx_cast`

inherits from bad_cast.

Bike-shedding: We can also use get instead for product and sum types. However the product version can not throw while the sum version can throw. In addition we should add the pointer overload for product types,

There is a new proposal [P0042] that has the same concern for ErasedClass types. The overlap between Sum types and ErasedClass types at a high level is surprising, as we can consider ErasedClass types as the Sum of all types satisfying the type we want to erase, and that Sum types can be seen as a ErasedClass type where the types are given explicitly.

However when we enter in the details the concepts are quite different. P0042 proposal defines an operation recover on ErasedClass types, while the cast operation is part of the Sum type definition.

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

Open points

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

- Do we want to adopt the new in_place definition?
- Do we want in place constructor for any?
- Do we want the clear and reset changes?
- Do we want the operator bool changes?
- Do we want the not-a-value none?
- Do we want the emplace_xxx factories?
- Do we want a single access interface from? get or sum_cast?

Technical Specification

The wording is relative to [N4480].

The present wording doesn't contain any modification to the variant proposal, as it is not yet on the TS.

Move in_place_t from [optional/synop] and [optional/inplace] to the synopsis, replace in_place by`

```
  struct in_place_t {};
  constexpr in_place_t in_place(unspecified);
  template <class ...T>;
    constexpr in_place_t in_place(unspecified<T...>);
  template <size N>;
  constexpr in_place_t in_place(unspecified<N>);
```

Update [optional.synopsis] adding after `make_optional`

```
  template <class T, class ...Args>
    optional<T> emplace_optional(Args&& ...args);
```

Update [optional.object] updating `in_place_t` by `in_place_t (&)(unspecified)` and add

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

Add in [optional.specalg]

```
  template <class T, class ...Args>
    optional<T> emplace_optional(Args&& ...args);
```

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

Update [any.synopsis] adding

```
Comparison with none
    template <class T> constexpr bool operator==(const any&, none_t) noexcept;
    template <class T> constexpr bool operator==(none_t, const any&) noexcept;
    template <class T> constexpr bool operator!=(const any&, none_t) noexcept;
    template <class T> constexpr bool operator!=(none_t, const any&) noexcept;
```

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

Add inside class `any`

```
// Constructors
```

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

```
  template <class T, class ...Args>
    any(in_place_t (&)(unspecified<T>), Args&& ...);
  template <class T, class U, class... Args>
    explicit any(in_place_t (&)(unspecified<T>), initializer_list<U>,
Args&&...);

// any assignment
  any& operator=(none_t) noexcept;
```

```
  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;
  explicit operator bool() const noexcept;
```

and replace any use of `empty()` by `bool(*this)`

Add in [any/cons]

```
  any(none_t);

  template <class T, class ...Args>
    any(in_place_t(&)(unspecified<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_t (&)(unspecified<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]

```
    any& operator=(none_t) 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.

```
    template <class T, class U, class ...Args>
    void emplace(initializer_list<U> il, 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 `sil, 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 `explicit operator bool`.

Add in [any.comparison]

```
template <class T> constexpr bool operator==(const any& x, none_t) noexcept;

template <class T> constexpr bool operator==(none_t, const any& x) noexcept;
```

Returns:

`!x`.

```
template <class T> constexpr bool operator!=(const any& x, none_t) noexcept;

template <class T> constexpr bool operator!=(none_t, const any& x) noexcept;
```

Returns:

`bool(x)`.

Add in [any.nonmembers]

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

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

Acknowledgements

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

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

References

[N4480] N4480 - Working Draft, C++ Extensions for Library Fundamentals http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4480.html

[N4542] N4542 - Variant: a type-safe union (v4) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4542.pdf

[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

[P0042] DXXXX – std::recover: undoing type erasure

509