Document number: P0032
Date: 2015–09-24
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

Homogeneous interface for `variant<Ts...>`, `any` and `optional<T>`

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.

Contents

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 [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 with 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.

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 yet.
- `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 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.

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 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 an explicit bool conversion,
    - rename the clear member function to reset,
- Add a none_t type.
- Add a none constexpr variable of type none_t.
- Add a make_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>.

This paper doesn't propose yet an homogeneous interface to access these possibly valued types, even if a possible direction is suggested.

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');
```

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 was empty and so its size is 1. We don't think this would reduce significantly the performances, however some measure need to be done if there is an interest.

`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` 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=...;
any af=...;
```

Before:

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

After:

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

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 (vf.has_value()) ...
```

```
if (af.has_value()) ...
```

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();
```

About a *not-a-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;
```

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{}; }
```

Do we need an explicit `make_any` factory?

`any` is not a generic type but a type erased type. `any` play the same role than a possible `make_any`.

This paper however propose a `make_any` factory for the emplace case, see below.

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

About emplace factories

However, we could consider a `make_xxx` factory that in place constructs a `T`.

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

```
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 `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 be:

```
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)
      auto a = any(Foo{v1, ..., vn})
```

After:

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

Which file for `in_place_t` and `in_place`?

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

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

Access interface

The generic `get<T>(t)` is convenient for product types as we know that the product type will contain an instance of any one of its parts. `any`, `optional<T>` and `variant<..., T, ...>` can only possibly store an instance of type T. We could also use `get` for product and sum types. However the product version can not throw while the sum version can throw.

[P0042] contains a complete description of the asymmetries on the design of the interface access to these classes.

The best example of possibly storing an instance of type `T` is in our opinion `optional<T>`. The interface to the value is familiar to most of the C++ developers as it uses the pointer like interface.

- Explicit bool conversion (`operator bool()`) to check if there is a value,
- dereferencing (`operator*()`) to get a reference to the stored instance,

- `get()` to get the address of the stored instance and
- the indirection operator (`operator ->()`) to access to one of the members of the stored instance.

All the access operations have as pre-condition that the type contains an instance of `T`. In addition, `optional<T>` has a `value()` safe function member that throws a `bad_optional_access` if the type doesn't contains an instance of `T`. It have been argued that value is not a good name as the parameter can be `T&` and that this is not a problem for `optional`, because the standard support `optional<T&>`. A name more appropriated would be preferred.

`any` and `variant` could have a similar interface (and why not any sum type or type erased class, as e.g. `std::function`). The problem is that classes as `any`, `variant<Ts, ...>` haven't a differentiated type `T`. However once we fix a specific type `T`, we can see these types as possibly storing an instance of type `T`. The role of the following wrapper is exactly that: wrap any of these types by selecting just a possibly type `T` for which we want to have access to.

```
template <class T, class Possibly>
class type_selector;

template <class T, class P>
type_selector<T,P> select(P&& p)
{
  return type_selector<T, P>(p);
}
```

[P0042] proposed `try_recover` which is similar to `select`, however the result type of `try_recover` doesn't provide the `operator->()` and the `value()` member functions.

This selection should behave as `optional<T>` and so we could define the typical `operator bool()`, `operator*()`, `get()` and `operator->()` on this class. With this interface we could use it as in

```
any a;
// ...
if (select<int>(a)) …
// …
int& i = select<int>(a).value(); // can throw
//
auto& api = select<int>(a);
if (api) return *api;

int * ptr = api.get();

auto& apt = select<T>(a)
apt->f(); // for some function member T::f()
```

This interface is more in line with the smart pointer interface, once we have fixed one of the alternative types.

An alternative design is to have a function that transforms any of these types in an `optional<T&>`. The main problem is that we don't have yet optional of references.

We can also provide non-member functions,

`holds<T>(s)` (the equivalent to `select<T>(s)::operator bool()`), `storage_address_of<T>(s)` (the storage address of a possibly `T`)

With these functions we can define

`reference_of<T>(s)` (the equivalent of `*select<T>(s)`)

`address_of<T>(s)` (the equivalent of `select<T>(s)::operator->()`)

`value_of<T>()` (the equivalent of `select<T>(s)::value()`).

These functions can be used as

```
any a;
// ...
if (holds<int>(a)) return reference_of<int>(a);

auto& ref = value_of<int>(a);

int* ptr = address_of<int>(a);
```

Even if the standard provides a default definition for these function, these should be customization points and the user should be able to overload theme.

Given

```
struct Foo { Foo(int, double, char); };
optional<Foo> of=...;
const optional<Foo> cof=...;
optional<Foo> fof();
variant<int, Foo> vf=...;
const variant<int, Foo> cvf=...;
variant<int, Foo> fvf();
any af=...;
const any caf=...;
any faf();
```

Before:

```
auto& xo = *of;
auto const& cxo = *cof;
auto&& rxo = *fof();
auto& xo = of.value();
auto& xv = get<1>(vf);
auto& xv = get<Foo>(vf);

auto& xa = any_cast<Foo&>(af);
auto const& xa = any_cast<Foo const&>(caf);
auto && xa = any_cast<Foo const&>(faf());
auto* pa = any_cast<Foo>(&af);
auto const* cpa = any_cast<Foo>(&caf);
```

After:

```
auto& xo1 = referece_of<1>(of);
auto& xo2 = referece_of<Foo>(of);
auto const& cxo1 = referece_of<1>(cof);
```

```
auto const& cxo2 = referece_of<Foo>(cof);
auto && fxo1 = referece_of<1>(fof());
auto && fxo2 = referece_of<Foo>(fof());

auto& xo1 = value_of<1>(of);
auto& xo2 = value_of<Foo>(of);
auto& xv = value_of<1>(vf);
auto& xv = value_of<Foo>(vf);
auto& xa = value_of<Foo>(af);
```

An open point is what should `holds` return when the selected type is `nullopt_t` on an optional

```
if (holds<nullopt_t>(opt)) ...
```

or the equivalent

```
if (select<nullopt_t>(opt)) ...
```

We are checking here if the optional value opt is disengaged.

Moving to a access like interface goes together with changing of `bad_any_cast` to `bad_any_access`. It seems natural that all these `bad_xxx_access` inherits from `bad_access`.

[P0050] proposes a high level alternative way to inspect the stored value of sum types, through a `match` function.

We have not yet a implemented yet a concrete proposal respect to this access issue and a separated paper will be needed if there is interest, maybe a follow up of [P0042].

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 `make_xxx` factories?
- Do we want to have a follow up for aconcept based on the functions `holds` and `storage_address_of`
- Do we want to have a follow up for `select<T>/select<I>`?
- Do we want to have a follow up for the observers `reference_of`, `value_of` and `address_of`?

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, nor the `select`, `holds`, `storage_address_of`, `reference_of`, `value_of`, `address_of` functions as we have not yet a prototype .

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> make_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> make_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 make_any(Args&& ...args);
```

Add inside class `any`

```
// Constructors

  constexpr any() noexcept;
  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]

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

  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 make_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.

David Krauss for its proposal [P0042] which inspired me to reduce the minimal interface for possibly valued types to `holds/storage_address_of`.

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] P0042 – std::recover: undoing type erasure

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0042r0.pdf

[P0050] P0050 – C++ generic match function

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0050r0.pdf