Document number: D0032
Date: 2015–09-08
Project: Programming Language C++, Library Evolution Working Group
Reply-to: Vicente J. Botet Escriba <vicente.botet@wanadoo.fr>

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

This paper identifies some minor inconveniences 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 inconveniences).

Contents

Introduction

This paper identifies some minor inconveniences 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 inconveniences).

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`/`get_if` interface by the something like the `any_cast`, `variant_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 `variant<Ts...>` proposal ends by supporting possible empty variants, the stored type is any of the `Ts` and know at run-time.

The following incoherencies 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` provides `any::clear()` to unset the value while `optional` uses assignment from a `nullopt_t`. If `variant<Ts...>` proposal ends by supporting possibly empty variants, we expect that it will have a `reset()` member function.
- `optional` provides a `explicit bool` conversion while `any` provides an `any::empty` member function. If `variant<Ts...>` proposal ends been possibly empty, we expect that it will have a `explicit bool` conversion.
- `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_if<T>(variant<Ts...>*)` by `variant_cast<T>(variant<Ts...>*)`.
  - Replace the `get_if<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

any 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 {} };
```

any 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 a;
a.emplace<T>(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.

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.

About the constant `none`

Instead of an additional `none`, `any{}` plays well the role. However, the authors think that using `none` is much more explicit.

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

Do we need a specific type for `none`

Instead of an additional type `none_t` to declare the constant `none` as

```
constexpr none_t none{}
```

we can just declare it as

```
constexpr any none{}
```

which plays well its role.

The advantages of the `any` constant is that we don't need conversions. However, assignment from `none` could be less efficient. If performance is required the user should use the `reset` function.

Alternatively we can add `none_t` and define the corresponding conversions as done for

optional

About a none_t type implicitly convertible to any and optional

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

The problem is that then none 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 optional(1) would e.g.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 emplace_any could be:

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

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

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.

Getters versus cast

The generic `get<T>(t)` and `get<I>(t)` is convenient for product types as we know that the product type will contain an instance of any one of its parts. However, both `any` and `variant` are sum types, and so we are not sure the sum type stores the request type. This is why `any` propose the use of `any_cast`. We suggest that variant should use some kind of cast, e.g. `variant_cast`.

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`.

`Generic sum_cast`

In the same way we have `get` for product types, why not have the same generic name for sum types, e.g. `sum_cast`. This will be a customization point and the user should be able to overload these functions.

There is a new proposal [RC] that has the same concern for TypeEsared types. The overlap between Sum types and TypeErased types is incredible, as we can consider TypeErased types as the Sum of all types satisfying type we want to erase, and that the TypeErased recover proposal consider Sum types as a TypeErased type where the types are given explicitly.

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 in place constructor for `any`?
- Do we want to adopt the new `in_place` definition ?
- Do we want the `clear` and `reset` changes?
- Do we want the `operator bool` changes?
- Do we want the `emplace_xxx` factories?

- Do we want to move `variant` access interface from `get`/`get_if` to a cast like `variant_cast` interface?
- If yes, do we want a generic `sum_cast`?
- If yes, do we want a generic `sum_cast` overload for `optional`?

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(in_place, std::forward(args)...)`.

Update [any.synopsis] adding

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

Add inside class `any`

```
  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&&...);

  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;
```

Add after class `any`

```
  constexpr any none{};
```

Add in [any/cons] `any` construct/destruc after p14

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

Requires: is_constructible_v 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 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.

```
template <class T, class U, class ...Args>
void emplace(initializer_list<U> il, 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/nonmember]

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

Returns: `any(in_place, 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

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