Convert `std::any` to `std::variant`

Asked 2 years, 10 months ago Modified 2 years, 10 months ago Viewed 542 times

1

There's not much to be done with a `std::any` (except store it) until its contained type is known or suspected. Then it can be queried (`type()`) or cast (`any_cast`). But what about when instead of one type there is a need to deal with multiple types? In this case a solution can be to convert it to `std::variant` .

E.g. An API provides `std::any` objects, but only a finite set of types is required and the objects need to be stored in a container (vector, tree etc.).

How can `std::any` be converted to `std::variant` ?

Disclaimer: `std::any` is mainly intended to be used in library code where its purpose is to replace `void *` in some clever templates and type erasures. As with any new thing `std::any` can be overused and misused. Please think twice if `std::any` is the right solution for your code.

c++ c++17 variant

Share Improve this question Follow

asked Apr 13, 2020 at 7:02
bolov
70.6k 15 139 217

1 Answer

Sorted by: Highest score (default)

4

This code takes a `std::any` object along with a list of types and converts the object to `std::variant` or throws `std::bad_any_cast` if the stored type is not one of the given types.

```
#include <any>
#include <variant>
#include <optional>
#include <typeinfo>
```

```
        bool found = ((a.type() == typeid(Args) && (v = std::any_cast<Args>(std::move(a))),
true)) || ...);

    if (!found)
        throw std::bad_any_cast{};

    return std::move(*v);
}
```

Example usage:

```
auto test(const std::any& a)
{
    auto v = any_to_variant_cast<int, std::string>(a);

    std::visit([](auto val) { std::cout << val << std::endl; }, v);
}
```

Code on godbolt

Some explanations:

`std::optional<std::variant<Args...>>` is used because `std::variant<Args...>` default constructor constructs the variant holding the value-initialized value of the first alternative and requires the first alternative to be default constructible.

```
    ((a.type() == typeid(Args) && (v = std::any_cast<Args>(std::move(a)), true)) || ...)
//    ------------------------     -------------------------------------
//          type_check                              any_cast
```

This is a fold expression. I've renamed some of the subexpression to be easier to explain. With the renaming the expression becomes:

```
// ((type_check && (any_cast, true)) || ...)
```

- if `type_check` is `false` then:
  - `(any_cast, true)` is not evaluated due to the short circuit of `&&`
  - `(type_check && (any_cast, true))` evaluates to `false`
  - the next op in the fold expression is evaluated

        - `(any_cast, true)` evaluates to `true`
    - `(type_check && (any_cast, true))` evaluates to `true`
    - the rest of the fold is not evaluated due to the short circuit of `||`
    - the whole expression (the fold) evaluates to `true`
- if no `type_check` evaluates to `true` then the whole expression (the fold) evaluates to `false`

Share Improve this answer Follow edited Apr 13, 2020 at 8:32 answered Apr 13, 2020 at 7:02

bolov
70.6k 15 139 217