Yandex Taxi

Better C++14 Reflections

Without Macro, Markup nor external Tooling

Полухин Антон

Antony Polukhin

Yandex Taxi

[Boost.]PFR

https://github.com/apolukhin/magic_get

Some structure

```
struct complicated_struct {
    int i;
    short s;
    double d;
    unsigned u;
};
```

Something that should not work...

```
#include <iostream>
#include <boost/pfr/precise.hpp>

struct complicated_struct { /*...*/ };

int main() {
    using namespace boost::pfr::ops;

    complicated_struct s {1, 2, 3.0, 4};
    std::cout << "s == " << s << std::endl;      // Compile time error?
}
```

But it works!..

```
antoshkka@home:~$ ./test

S == {1, 2, 3.0, 4}
```

What's in the header?

```
#include <iostream>
#include <boost/pfr/precise.hpp>

struct complicated_struct { /*...*/ };

int main() {
    using namespace boost::pfr::ops;

    complicated_struct s {1, 2, 3.0, 4};
    std::cout << "s == " << s << std::endl;      // Compile time error?
}
```

We need to go deeper

```
template <class Char, class Traits, class T>
detail::enable_not_ostreamable_t<std::basic_ostream<Char, Traits>, T>
    operator<<(std::basic_ostream<Char, Traits>& out, const T& value)
{
    boost::pfr::write(out, value);
    return out;
}
```

We need to go deeper

```
template <class Char, class Traits, class T>
void write(std::basic_ostream<Char, Traits>& out, const T& val) {
    out << '{';
    detail::print_impl<0, boost::pfr::tuple_size_v<T> >::print(out, val);
    out << '}';
}
```

That's suspicious....

```
template <class Char, class Traits, class T>
void write(std::basic_ostream<Char, Traits>& out, const T& val) {
    out << '{';
    detail::print_impl<0, boost::pfr::tuple_size_v<T> >::print(out, val);
    out << '}';
}
```

O_O

```
template <std::size_t FieldIndex, std::size_t FieldsCount>
struct print_impl {

    template <class Stream, class T>
    static void print (Stream& out, const T& value) {
        if (!!FieldIndex) out << ", ";
        out << boost::pfr::get<FieldIndex>(value);    // std::get<FieldIndex>(value)
        print_impl<FieldIndex + 1, FieldsCount>::print(out, value);
    }

};
```

O_O

```
template <std::size_t FieldIndex, std::size_t FieldsCount>
struct print_impl {

    template <class Stream, class T>
    static void print (Stream& out, const T& value) {
        if (!!FieldIndex) out << ", ";
        out << boost::pfr::get<FieldIndex>(value);    // std::get<FieldIndex>(value)
        print_impl<FieldIndex + 1, FieldsCount>::print(out, value);
    }

};
```

O_O

```cpp
template <std::size_t FieldIndex, std::size_t FieldsCount>
struct print_impl {

    template <class Stream, class T>
    static void print (Stream& out, const T& value) {
        if (!!FieldIndex) out << ", ";
        out << boost::pfr::get<FieldIndex>(value);    // std::get<FieldIndex>(value)
        print_impl<FieldIndex + 1, FieldsCount>::print(out, value);
    }

};
```

O_O

```
template <std::size_t FieldIndex, std::size_t FieldsCount>
struct print_impl {

    template <class Stream, class T>
    static void print (Stream& out, const T& value) {
        if (!!FieldIndex) out << ", ";
        out << boost::pfr::get<FieldIndex>(value);    // std::get<FieldIndex>(value)
        print_impl<FieldIndex + 1, FieldsCount>::print(out, value);
    }

};
```

What's going on here?

```
/// Returns reference or const reference to a field with index `I` in aggregate T
/// Requires: <skipped>
template <std::size_t I, class T>
decltype(auto) get(const T& val) noexcept;



/// `tuple_size_v` is a template variable that contains fields count in a T
/// Requires: C++14
template <class T>
constexpr  std::size_t tuple_size_v = /*...*/;
```

How to count fields

(the basics of PFR)

Basic idea for counting fields

```
static_assert(std::is_pod<T>::value, "")
```

Basic idea for counting fields

```
static_assert(std::is_pod<T>::value, "")
```

```
T { args... }
```

Basic idea for counting fields

```
static_assert(std::is_pod<T>::value, "")
```

```
T { args... }
```

```
sizeof...(args) <= fields count
```

Basic idea for counting fields

```
static_assert(std::is_pod<T>::value, "")
```

```
T { args... }
```

```
sizeof...(args) <= fields count
```

```
typeid(args)... == typeid(fields)...
```

Basic idea for counting fields

```
static_assert(std::is_pod<T>::value, "")
```

```
T { args... }
```

```
sizeof...(args) <= fields count
```

```
typeid(args)... == typeid(fields)...
```

```
sizeof(char) == 1
sizeof...(args) <= sizeof(T)
```

Basic idea for counting fields

```
static_assert(std::is_pod<T>::value, "")
```

```
T { args... }
```

```
sizeof...(args) <= fields count
```

```
typeid(args)... == typeid(fields)...
```

```
sizeof(char) == 1

sizeof...(args) <= sizeof(T) * CHAR_BITS
```

Basic idea for counting fields

```
static_assert(std::is_pod<T>::value, "")

T { args... }

sizeof...(args) <= fields count                typeid(args)... == typeid(fields)...

sizeof(char) == 1                                                ???

sizeof...(args) <= sizeof(T) * CHAR_BITS
```

Basic idea for counting fields

```
static_assert(std::is_pod<T>::value, "")
```

```
T { args... }
```

```
sizeof...(args) <= fields count
```

```
typeid(args)... == typeid(fields)...
```

```
sizeof(char) == 1
```

???

```
sizeof...(args) <= sizeof(T) * CHAR_BITS
```

Ubiq

```
struct ubiq {
    template <class Type>
    constexpr operator Type&() const;
};
```

Ubiq

```
struct ubiq {
    template <class Type>
    constexpr operator Type&() const;
};



int i = ubiq{};
double d = ubiq{};
char c = ubiq{};
```

Done

```
static_assert(std::is_pod<T>::value, "")

T { args... }

sizeof...(args) <= fields count          typeid(args)... == typeid(fields)...

sizeof(char) == 1                                     ubiq{}

sizeof...(args) <= sizeof(T) * CHAR_BITS
```

Putting all together

```
struct ubiq_constructor {
    std::size_t ignore;

    template <class Type>
    constexpr operator Type&() const noexcept; // Undefined
};
```

Putting all together. Simplified version]:->

```
// #1
template <class T, std::size_t I0, std::size_t... I>
constexpr auto detect_fields_count(std::size_t& out, std::index_sequence<I0, I...>)
    -> decltype( T{ ubiq_constructor{I0}, ubiq_constructor{I}... } )
{ out = sizeof...(I) + 1;      /*...*/ }


// #2
template <class T, std::size_t... I>
constexpr void detect_fields_count(std::size_t& out, std::index_sequence<I...>) {
    detect_fields_count<T>(out, std::make_index_sequence<sizeof...(I) - 1>{});
}
```

Putting all together. Simplified version]:->

```
// #1
template <class T, std::size_t I0, std::size_t... I>
constexpr auto detect_fields_count(std::size_t& out, std::index_sequence<I0, I...>)
    -> decltype( T{ ubiq_constructor{I0}, ubiq_constructor{I}... } )
{ out = sizeof...(I) + 1;      /*...*/ }


// #2
template <class T, std::size_t... I>
constexpr void detect_fields_count(std::size_t& out, std::index_sequence<I...>) {
    detect_fields_count<T>(out, std::make_index_sequence<sizeof...(I) - 1>{});
}
```

Putting all together. Simplified version]:->

```
// #1
template <class T, std::size_t I0, std::size_t... I>
constexpr auto detect_fields_count(std::size_t& out, std::index_sequence<I0, I...>)
    -> decltype( T{ ubiq_constructor{I0}, ubiq_constructor{I}... } )
{ out = sizeof...(I) + 1;      /*...*/ }

// #2
template <class T, std::size_t... I>
constexpr void detect_fields_count(std::size_t& out, std::index_sequence<I...>) {
    detect_fields_count<T>(out, std::make_index_sequence<sizeof...(I) - 1>{});
}
```

Putting all together. Simplified version]:->

```
// #1
template <class T, std::size_t I0, std::size_t... I>
constexpr auto detect_fields_count(std::size_t& out, std::index_sequence<I0, I...>)
    -> decltype( T{ ubiq_constructor{I0}, ubiq_constructor{I}... } )
{ out = sizeof...(I) + 1;      /*...*/ }


// #2
template <class T, std::size_t... I>
constexpr void detect_fields_count(std::size_t& out, std::index_sequence<I...>) {
    detect_fields_count<T>(out, std::make_index_sequence<sizeof...(I) - 1>{});
}
```

Putting all together. Simplified version]:->

```
// #1
template <class T, std::size_t I0, std::size_t... I>
constexpr auto detect_fields_count(std::size_t& out, std::index_sequence<I0, I...>)
    -> decltype( T{ ubiq_constructor{I0}, ubiq_constructor{I}... } )
{ out = sizeof...(I) + 1;      /*...*/ }


// #2
template <class T, std::size_t... I>
constexpr void detect_fields_count(std::size_t& out, std::index_sequence<I...>) {
    detect_fields_count<T>(out, std::make_index_sequence<sizeof...(I) - 1>{});
}
```

Putting all together. Simplified version]:->

```
// #1
template <class T, std::size_t I0, std::size_t... I>
constexpr auto detect_fields_count(std::size_t& out, std::index_sequence<I0, I...>)
    -> decltype( T{ ubiq_constructor{I0}, ubiq_constructor{I}... } )
{ out = sizeof...(I) + 1;      /*...*/ }

// #2
template <class T, std::size_t... I>
constexpr void detect_fields_count(std::size_t& out, std::index_sequence<I...>) {
    detect_fields_count<T>(out, std::make_index_sequence<sizeof...(I) - 1>{});
}
```

It works!..

but...

Bad news!

It compiles for eternity or reaches the compiler instantiation depth limit

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields

```
T{}
```

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields

```
T{}
T{ubiq{}}
```

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields

```
T{}
T{ubiq{}}
T{ubiq{}, ubiq{}}
```

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields

```
T{}
T{ubiq{}}
T{ubiq{}, ubiq{}}
T{ubiq{}, ubiq{}, ubiq{}}
```

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields

T{}

T{ubiq{}}

T{ubiq{}, ubiq{}}

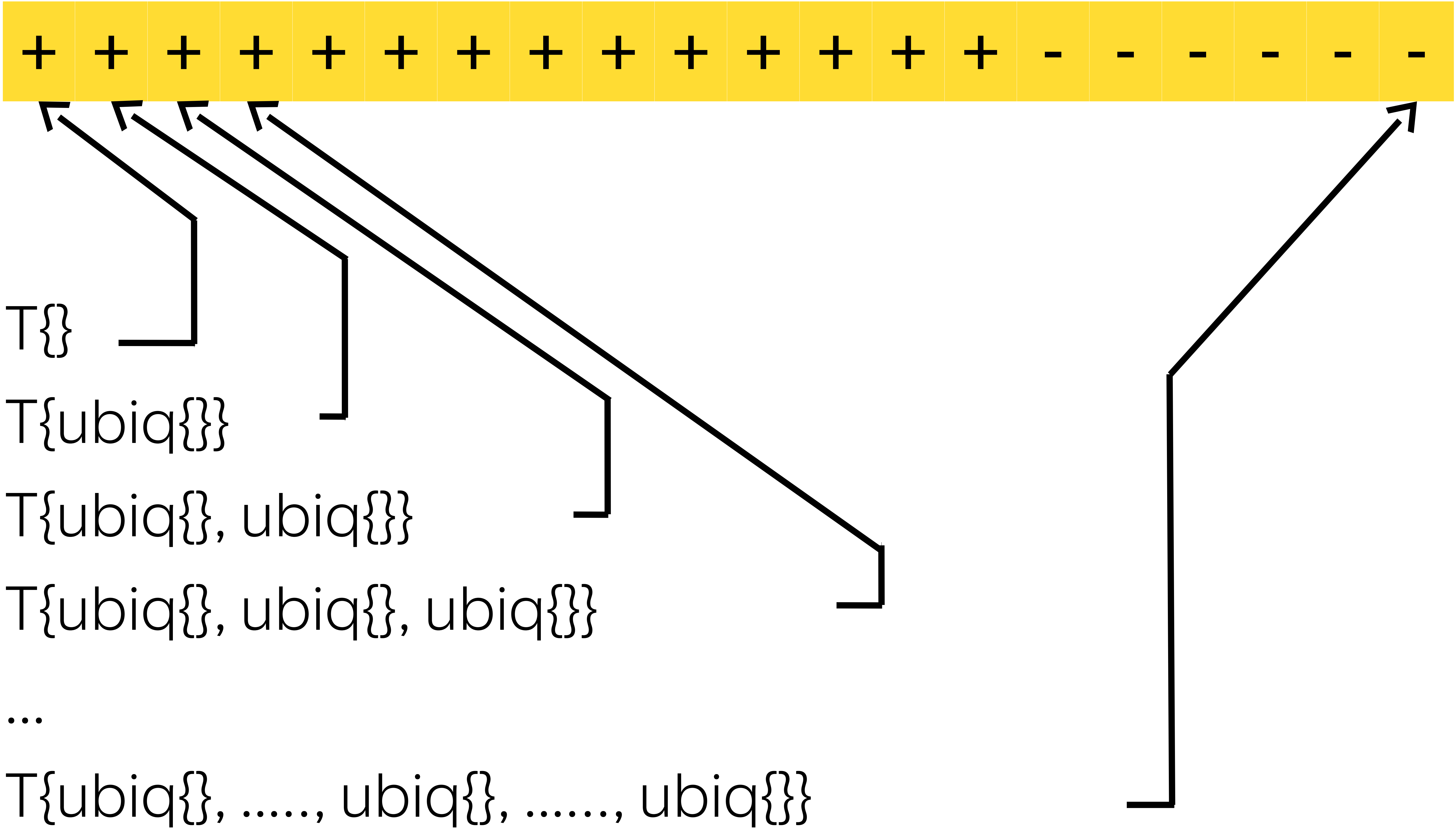
T{ubiq{}, ubiq{}, ubiq{}}

...

T{ubiq{}, ..., ubiq{}, ..., ubiq{}}

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields



Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling:

+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling:

+	+	+	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

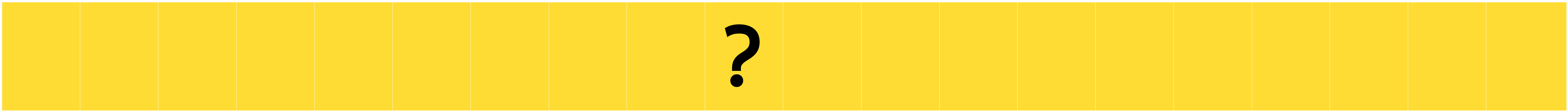
- log(N) instantiations depth
- log(N) instantiations

Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

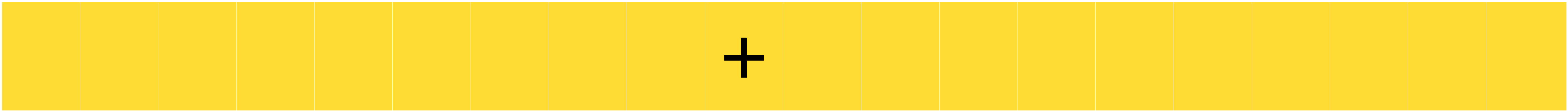


Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

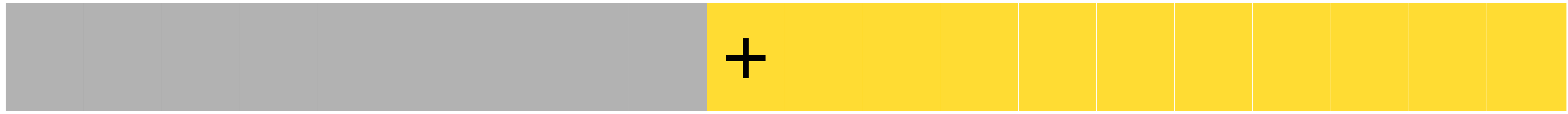


Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

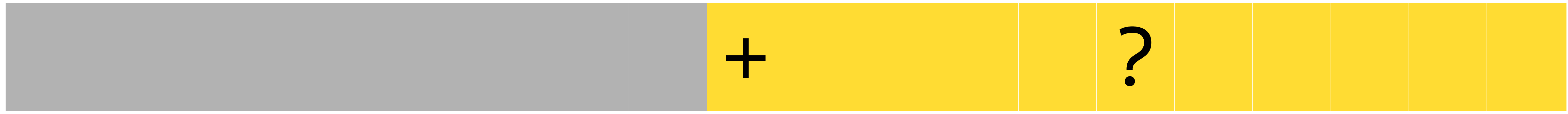


Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

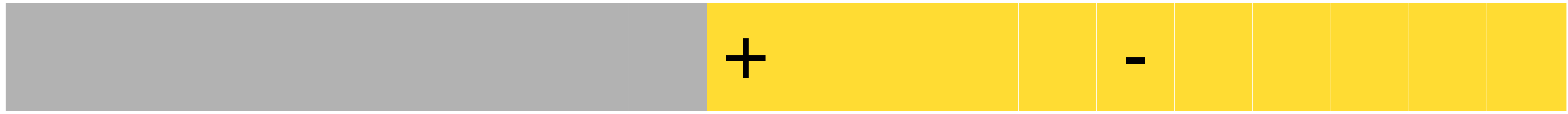


Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

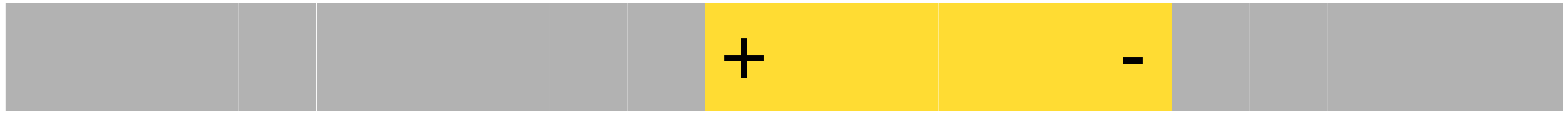


Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

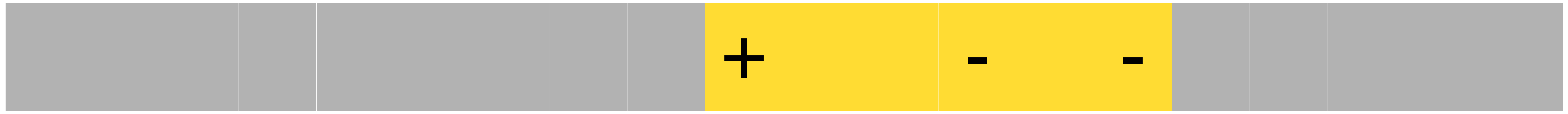


Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

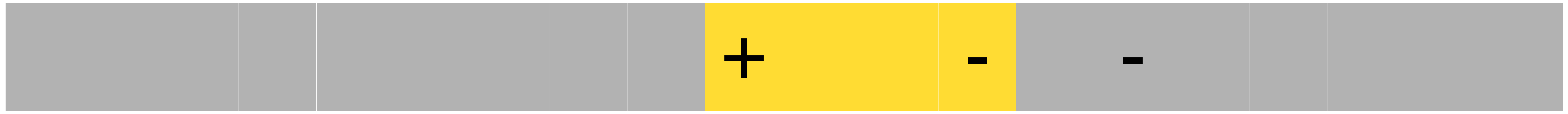


Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

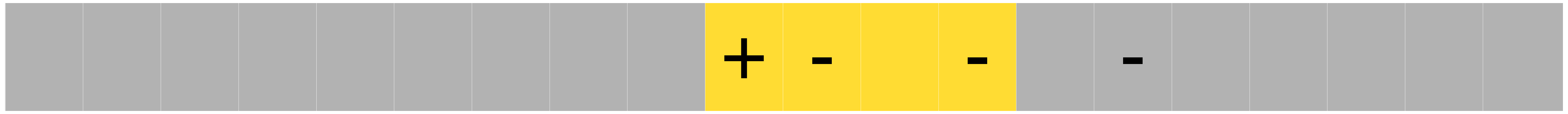


Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations

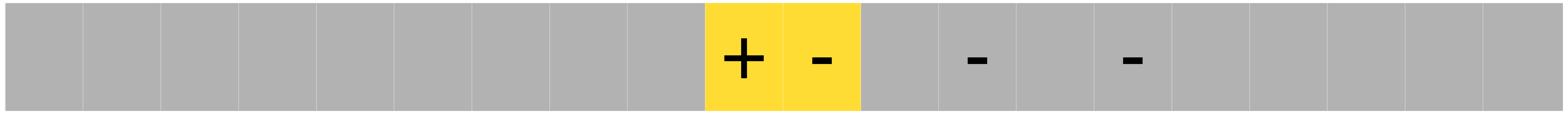


Let's fix it!

Aggregates **without** deleted constructors and **with** default constructible fields have a single point after which increasing ubiq counts stop compiling.

Oh! Binary search fits perfectly:

- log(N) instantiations depth
- log(N) instantiations



What about other aggregates?

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

-	-	-	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

-	-	-	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-

We need an **eager version** of search that does:

- log(N) instantiations depth
- N instantiations

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

-	-	-	+	+	+	+	+	+	+	+	+	+	+	-	-	-	-	-	-

We need a **bad version** Binary search:

- log(N) instantiations depth
- N instantiations

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

We need a **bad version** Binary search:

- log(N) instantiations depth
- N instantiations

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

We need a **bad version** Binary search:

- log(N) instantiations depth
- N instantiations

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

We need a **bad version** Binary search:

- log(N) instantiations depth
- N instantiations

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

We need a **bad version** Binary search:

- log(N) instantiations depth
- N instantiations

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

We need a **bad version** Binary search:

- log(N) instantiations depth
- N instantiations

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

We need a **bad version** Binary search:

- log(N) instantiations depth
- N instantiations

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

We need a **bad version** Binary search:

- log(N) instantiations depth
- N instantiations

Let's fix it (2)!

Aggregates **with** deleted constructor or with some non default constructible fields have gaps:

We need a **bad version** Binary search:

- log(N) instantiations depth
- N instantiations

-	-	+	+	+	+	+	+	+	-	-	-	-	-	-	-	-	-	-	-

How to get the field type?

Getting the field type

```
T{ ubiq_constructor<I>{}... }
```

Getting the field type

```
T{ ubiq_constructor<I>{}... }
```

```
ubiq_constructor<I>{}::operator Type&() const
```

Getting the field type

```
T{ ubiq_constructor<I>{}... }
```

```
ubiq_constructor<I>{}::operator Type&() const
```

```
Type
```

Getting the field type

```
T{ ubiq_constructor<I>{}... }
```

```
ubiq_constructor<I>{}::operator Type&() const
```

```
Type
```

```
ubiq_constructor<I>{ TypeOut& }
```

Getting the field type

```
T{ ubiq_constructor<I>{}... }
```

```
ubiq_constructor<I>{}::operator Type&() const
```

```
Type
```

~~`ubiq_constructor<I>{ TypeOut& }`~~

😭

Naïve solution

POD = { (public|private|protected) + (fundamental | POD)* };

Naïve solution

fundamental (not a pointer) → int

int → output

output[I]... → Types...

Naïve solution

```
template <std::size_t I>
struct ubiq_val {
    std::size_t* ref_;

    template <class Type>
    constexpr operator Type() const noexcept {
        ref_[I] = typeid_conversions::type_to_id(identity<Type>{});
        return Type{};
    }
};
```

Naïve solution

```
#define BOOST_MAGIC_GET_REGISTER_TYPE(Type, Index)              \
    constexpr std::size_t type_to_id(identity<Type>) noexcept { \
        return Index;                                           \
    }                                                           \
    constexpr Type id_to_type( size_t_<Index > ) noexcept {     \
        Type res{};                                             \
        return res;                                             \
    }                                                           \
    /**/
```

Naïve solution

```
BOOST_MAGIC_GET_REGISTER_TYPE(unsigned char         , 1)
BOOST_MAGIC_GET_REGISTER_TYPE(unsigned short        , 2)
BOOST_MAGIC_GET_REGISTER_TYPE(unsigned int          , 3)
BOOST_MAGIC_GET_REGISTER_TYPE(unsigned long         , 4)
BOOST_MAGIC_GET_REGISTER_TYPE(unsigned long long    , 5)
BOOST_MAGIC_GET_REGISTER_TYPE(signed char           , 6)
BOOST_MAGIC_GET_REGISTER_TYPE(short                 , 7)
BOOST_MAGIC_GET_REGISTER_TYPE(int                   , 8)
BOOST_MAGIC_GET_REGISTER_TYPE(long                  , 9)
BOOST_MAGIC_GET_REGISTER_TYPE(long long             , 10)
```

Naïve solution

```
template <class T, std::size_t N, std::size_t... I>
constexpr auto type_to_array_of_type_ids(std::size_t* types) noexcept
    -> decltype(T{ ubiq_constructor<I>{}... })
{
    T tmp{ ubiq_val< I >{types}... };
    return tmp;
}
```

Naïve solution

```
template <class T, std::size_t... I>
constexpr auto as_tuple_impl(std::index_sequence<I...>) noexcept {
    constexpr auto a = array_of_type_ids<T>();              // #0

    return std::tuple<                                      // #3
        decltype(typeid_conversions::id_to_type(            // #2
            size_t_<a[I]>{}                                 // #1
        ))...
    >{};
}
```

T to tuple

T to tuple

```
template <size_t I, class T>
auto get(T& v) noexcept {
    using tuple = decltype(as_tuple_impl<T>(...));

    return ???;
}
```

T to tuple

```
template <size_t I, class T>
auto get(T& v) noexcept {
    using tuple = decltype(as_tuple_impl<T>(...));


    return ???;
}
```

T to tuple

```
template <size_t I, class T>
auto get(T& v) noexcept {
    using tuple = decltype(as_tuple_impl<T>(...));

    return std::get<I>(
        reinterpret_cast<tuple&>(v)   // UB
    );
}
```

T to tuple

```
template <size_t I, class T>
auto get(T& v) noexcept {
    using tuple = decltype(as_tuple_impl<T>(...));

    return std::get<I>(
        reinterpret_cast<tuple&>(v)   // UB    :-(
    );
}
```

T to tuple

```
template <size_t I, class T>
auto get(T& v) noexcept {
    using tuple = decltype(as_tuple_impl<T>(...));

    return *reinterpret_cast<tuple_element_t<I, tuple>* >(
        reinterpret_cast<unsigned char*>(&v) + offset_for<tuple, I>()
    );
}
```

T to tuple

```
template <size_t I, class T>
auto get(T& v) noexcept {
    using tuple = decltype(as_tuple_impl<T>(...));

    return *reinterpret_cast<tuple_element_t<I, tuple>* >(
        reinterpret_cast<unsigned char*>(&v) + offset_for<tuple, I>()
    );
}
```

T to tuple

```
template <size_t I, class T>
auto get(T& v) noexcept {
    using tuple = decltype(as_tuple_impl<T>(...));

    return *reinterpret_cast<tuple_element_t<I, tuple>* >(
        reinterpret_cast<unsigned char*>(&v) + offset_for<tuple, I>()
    );
}
```

T to tuple

```
template <size_t I, class T>
auto get(T& v) noexcept {
    using tuple = decltype(as_tuple_impl<T>(...));

    return *reinterpret_cast<tuple_element_t<I, tuple>* >(
        reinterpret_cast<unsigned char*>(&v) + offset_for<tuple, I>()
    );
}
```

Pitfalls of naive reflection

Pitfalls

- Enums are represented as an underlying type

Pitfalls

- Enums are represented as an underlying type
- Does not work with nested structures

Pitfalls

- Enums are represented as an underlying type
- Does not work with nested structures
  - Or does the reflection recursively

Pitfalls

- Enums are represented as an underlying type
- Does not work with nested structures
  - Or does the reflection recursively (**flat reflection**)

Pitfalls

- Enums are represented as an underlying type
- Does not work with nested structures
  - Or does the reflection recursively (**flat reflection**)

```
struct struct0 {
    int i;
    short s;
};
struct struct1 {
    struct0 st;
    double d;
    unsigned u;
};
```

Pitfalls

- Enums are represented as an underlying type
- Does not work with nested structures
  - Or does the reflection recursively (**flat reflection**)

```
struct struct_flat {
    int i;
    short s;
    double d;
    unsigned u;
};
```

Pitfalls

- Enums are represented as an underlying type
- Does not work with nested structures
  - Or does the reflection recursively (flat reflection)
- A lot of computations to encode *cv* pointers as an integer

Pitfalls

- Enums are represented as an underlying type
- Does not work with nested structures
  - Or does the reflection recursively (flat reflection)
- A lot of computations to encode *cv* pointers as an integer
- Works only with PODs

That’s the best we can do!

That’s the best we can do!

…was I thinking

That’s NOT the best we can do!

Better field type detection

Version #1; creepy

Getting the field type

```
T{ ubiq_constructor<I>{}... }
```

```
ubiq_constructor<I>{}::operator Type&() const
```

```
Type
```

Getting the field type

```
T{ ubiq_constructor<I>{}... }
```

```
ubiq_constructor<I>{}::operator Type&() const
```

Type

Getting the field type

```
T{ ubiq_constructor<I>{}... }
```

```
ubiq_constructor<I>{}::operator Type&() const
```

Type

call `T{ ubiq_constructor<I>{}... }` **again from within the** `operator Type&()`

Getting the field type recursively

```
for_each_field_in_depth<T, I, Types...>(t, callback, ... );
```

Getting the field type recursively

```
for_each_field_in_depth<T, I, Types...>(t, callback, ... );
```

```
T{ ubiq_constructor_next<T, I>{}... }
```

Getting the field type recursively

```
for_each_field_in_depth<T, I, Types...>(t, callback, ... );
```

```
T{ ubiq_constructor_next<T, I>{}... }
```

```
ubiq_constructor_next<I>{}::operator Type&() const
```

Getting the field type recursively

```
for_each_field_in_depth<T, I, Types...>(t, callback, ... );
```

```
T{ ubiq_constructor_next<T, I>{}... }
```

```
ubiq_constructor_next<I>{}::operator Type&() const
{ for_each_field_in_depth<T, I+1, Types..., Type>(t, callback, ... ); }
```

Getting the field type recursively

```
for_each_field_in_depth<T, I, Types...>(t, callback, ... );
```

```
T{ ubiq_constructor_next<T, I>{}... }
```

```
ubiq_constructor_next<I>{}::operator Type&() const
{ for_each_field_in_depth<T, I+1, Types..., Type>(t, callback, ... ); }
```

Getting the field type recursively

```
for_each_field_in_depth<T, I, Types...>(t, callback, ... );

T{ ubiq_constructor_next<T, I>{}... }

ubiq_constructor_next<I>{}::operator Type&() const
{ for_each_field_in_depth<T, I+1, Types..., Type>(t, callback, ... ); }

...
```

Getting the field type recursively

```
for_each_field_in_depth<T, I, Types...>(t, callback, ... );

T{ ubiq_constructor_next<T, I>{}... }

ubiq_constructor_next<I>{}::operator Type&() const
{ for_each_field_in_depth<T, I+1, Types..., Type>(t, callback, ... ); }

...

callback<Types...>(t, make_tuple_of_references<Types...>(t))
```

Pitfalls

- Big compile time overhead

Pitfalls

- Big compile time overhead
- We rely on compiler cleverness for eliminating unnecessary copies

Pitfalls

- Big compile time overhead
- We rely on compiler cleverness for eliminating necessary copies
  - But no too much, so we assert that the compiler is clever enough:

    ```
    constexpr T tmp{ ubiq{I}... };
    ```

Pitfalls

- Big compile time overhead
- We rely on compiler cleverness for eliminating unnecessary copies
  - But no too much, so we assert that the compiler is clever enough:

    ```
    constexpr T tmp{ ubiq{I}... };
    ```
- Not always works

Even better field type detection

Version #2; very very creepy

The essence

The essence

- To register id ↔ type relations we need a way to save a global state in metafunctions

The essence

- To register id ↔ type relations we need a way to save a global state in metafunctions
- Statefull metaprogramming is not permitted by the standard

The essence

- To register id ↔ type relations we need a way to save a global state in metafunctions
- Statefull metaprogramming is not permitted by the standard
  - CWG is very serious about that

The essence

- To register id ↔ type relations we need a way to save a global state in metafunctions
- Statefull metaprogramming is not permitted by the standard
  - CWG is very serious about that
  - There's even a CWG 2118 issue to deal with some border cases of impure metafunctions

The essence

- To register id ↔ type relations we need a way to save a global state in metafunctions
- Statefull metaprogramming is not permitted by the standard
  - CWG is very serious about that
  - There's even a CWG 2118 **issue** to deal with some border cases of **impure metafunctions**

  Huh!

Type Loophole

```
template <class T, std::size_t N>
struct tag {
    // forward declaration of loophole(tag<T,N>) without a result type
    friend auto loophole(tag<T,N>);
};
```

Type Loophole

```
template <class T, std::size_t N>
struct tag {
    // forward declaration of loophole(tag<T,N>) without a result type
    friend auto loophole(tag<T,N>);
};

template <class T, class FieldType, std::size_t N, bool B>
struct fn_def {
    // definition of loophole(tag<T,N>) with a result type
    friend auto loophole(tag<T,N>) { return FieldType{}; }
};
```

Type Loophole

```
template <class T, std::size_t N>
struct loophole_ubiq {
    template<class U, std::size_t M> static std::size_t ins(...);
    template<class U, std::size_t M, std::size_t = sizeof(loophole(tag<T,M>{})) >
    static char ins(int);

    template<class U, std::size_t = sizeof(fn_def<
          T, U, N, sizeof(ins<U, N>(0)) == sizeof(char)
    >)>
    constexpr operator U&() const noexcept;
};
```

Type Loophole

```
template <class T, std::size_t N>
struct loophole_ubiq {
    template<class U, std::size_t M> static std::size_t ins(...);
    template<class U, std::size_t M, std::size_t = sizeof(loophole(tag<T,M>{})) >
    static char ins(int);

    template<class U, std::size_t = sizeof(fn_def<
          T, U, N, sizeof(ins<U, N>(0)) == sizeof(char)
    >)>
    constexpr operator U&() const noexcept;
};
```

Type Loophole

```
template <class T, std::size_t N>
struct loophole_ubiq {
    template<class U, std::size_t M> static std::size_t ins(...);
    template<class U, std::size_t M, std::size_t = sizeof(loophole(tag<T,M>{})) >
    static char ins(int);

    template<class U, std::size_t = sizeof(fn_def<
          T, U, N, sizeof(ins<U, N>(0)) == sizeof(char)
    >)>
    constexpr operator U&() const noexcept;
};
```

Type Loophole

```
template <class T, std::size_t N>
struct loophole_ubiq {
    template<class U, std::size_t M> static std::size_t ins(...);
    template<class U, std::size_t M, std::size_t = sizeof(loophole(tag<T,M>{})) >
    static char ins(int);

    template<class U, std::size_t = sizeof(fn_def<
            T, U, N, sizeof(ins<U, N>(0)) == sizeof(char)
    >)>
    constexpr operator U&() const noexcept;
};
```

Type Loophole

```
template <class T, std::size_t N>
struct loophole_ubiq {
    template<class U, std::size_t M> static std::size_t ins(...);
    template<class U, std::size_t M, std::size_t = sizeof(loophole(tag<T,M>{})) >
    static char ins(int);

    template<class U, std::size_t = sizeof(fn_def<
        T, U, N, sizeof(ins<U, N>(0)) == sizeof(char)
    >)>
    constexpr operator U&() const noexcept;
};
```

Type Loophole

```
int main() {
    struct test { char c; int i; };
    test t{loophole_ubiq<test, 0>{} };

    static_assert(
        std::is_same<
        char,
        decltype( loophole(tag<test, 0>{}) )
        >::value, ""
    );
}
```

Type Loophole

```
int main() {
    struct test { char c; int i; };
    test t{loophole_ubiq<test, 0>{} };

    static_assert(
        std::is_same<
        char,
        decltype( loophole(tag<test, 0>{}) )
        >::value, ""
    );
}
```

Perfect field type detection

Version #3; C++17 required

C++17

```
template <class T>
constexpr auto as_tuple_impl(T&& val) noexcept {
  constexpr auto count = fields_count<T>();
  if constexpr (count == 1) {
    auto& [a] = std::forward<T>(val);
    return detail::make_tuple_of_references(a);
  } else if constexpr (count == 2) {
    auto& [a,b] = std::forward<T>(val);
    return detail::make_tuple_of_references(a,b);
  } // ...
}
```

C++17

```
template <class T>
constexpr auto as_tuple_impl(T&& val) noexcept {
  constexpr auto count = fields_count<T>();
  if constexpr (count == 1) {
    auto& [a] = std::forward<T>(val);
    return detail::make_tuple_of_references(a);
  } else if constexpr (count == 2) {
    auto& [a,b] = std::forward<T>(val);
    return detail::make_tuple_of_references(a,b);
  } // ...
}
```

C++17

```
template <class T>
constexpr auto as_tuple_impl(T&& val) noexcept {
  constexpr auto count = fields_count<T>();
  if constexpr (count == 1) {
    auto& [a] = std::forward<T>(val);
    return detail::make_tuple_of_references(a);
  } else if constexpr (count == 2) {
    auto& [a,b] = std::forward<T>(val);
    return detail::make_tuple_of_references(a,b);
  } // ...
}
```

C++17

```
template <class T>
constexpr auto as_tuple_impl(T&& val) noexcept {
  constexpr auto count = fields_count<T>();
  if constexpr (count == 1) {
    auto& [a] = std::forward<T>(val);
    return detail::make_tuple_of_references(a);
  } else if constexpr (count == 2) {
    auto& [a,b] = std::forward<T>(val);
    return detail::make_tuple_of_references(a,b);
  } // ...
}
```

Is it useful?

or what features are available in [Boost.]PFR

Features

- Implemented (flat and precise):
  - Comparisons : <, <=, >, >=, !=, ==
  - Heterogeneous comparators: less<>, flat_equal<>
  - IO stream operators: operator <<, operator>>
  - Hashing: flat_hash, hash
  - Tie to/from structure

Features

- Implemented (flat and precise):
  - Comparisons : <, <=, >, >=, !=, ==
  - Heterogeneous comparators: less<>, flat_equal<>
  - IO stream operators: operator <<, operator>>
  - Hashing: flat_hash, hash
  - Tie to/from structure

- Do it on your own:
  - User defined serializers
  - Basic reflections
  - New type_traits: is_continuous_layout<T>, is_padded<T>, has_unique_object_representation<T>
  - New features for containers: punch_hole<T, Index>
  - More generic algorithms: vector_mult, parse to struct

Acknowledgements

people who helped and invented stuff

Acks

- Alexandr Poltavsky — for the Loophole
- Bruno Dutra — for the recursive reflection
- Chris Beck — for removing UBs from reinterpret_casts
- Abutcher-gh — for adding some tie functions

- Lisa Lippincott and Alexey Moiseytsev — for finding issues with alignments
- Nikita Kniazev, Anton Bikineev and others – for testing reporting and finding issues.

Спасибо!

Спасибо!

Thanks for listening!

Antony Polukhin

Developer in Yandex.Taxi

antoshkka@gmail.com

antoshkka@yandex-team.ru

https://github.com/apolukhin

РГ21 C++ РОССИЯ

https://stdcpp.ru/