C++14 Reflections
Without Macros, Markup nor External Tooling

Metaprogramming Tricks for POD Types

Antony Polukhin

Boost libraries maintainer (DLL, LexicalCast, Any, TypeIndex, Conversion)
+ Boost.CircularBuffer, Boost.Variant

Structure (very complicated)

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

Something that must not work...

```
#include <iostream>
#include "magic_get.hpp"

struct complicated_struct { /* … */ };

int main() {
    using namespace pod_ops;

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

But how?..

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

s == {1, 2, 3.0, 4}

What's in the header?

```cpp
#include <iostream>
#include "magic_get.hpp"

struct complicated_struct { /* … */ };

int main() {
    using namespace pod_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>
std::basic_ostream<Char, Traits>&
    operator<<(std::basic_ostream<Char, Traits>& out, const T& value)
{
    flat_write(out, value);
    return out;
}
```

...

```cpp
template <class Char, class Traits, class T>
void flat_write(std::basic_ostream<Char, Traits>& out, const T& val) {
    out << '{';
    detail::flat_print_impl<0, flat_tuple_size<T>::value >::print(out, val);
    out << '}';
}
```

O_O

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

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

};
```

Wow!..

```
/// Returns const reference to a field with index `I`
/// Example usage: flat_get<0>(my_structure());
template <std::size_t I, class T>
decltype(auto) flat_get(const T& val) noexcept;



/// `flat_tuple_size` has a member `value` that constins fields count
/// Example usage: std::array<int, flat_tuple_size<my_structure>::value > a;
template <class T>
using flat_tuple_size;
```

How to count fields?

Idea for counting fields

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

Idea for counting fields

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

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

Idea for counting fields

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

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

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

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

Idea for counting fields

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

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

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

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

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)
```

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)
```

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
```

```
struct ubiq {}
```

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

Putting all together

```
template <std::size_t I>
struct ubiq_constructor {
    template <class Type>
    constexpr operator Type&() const noexcept; // Undefined
};
```

Putting all together

```
std::make_index_sequence<5>{}        ===>          std::index_sequence<0, 1, 2, 3, 4>{}
```

Putting all together

```
// #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>{});
}
```

How to get the field type?

Idea #2

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

Idea #2

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

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

Idea #2

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

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

```
Type
```

Idea #2

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

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

```
Type
```

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

Idea #2

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

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

```
Type
```

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

What is a POD (roughly)?

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

Idea #2.5

`fundamental (not a pointer) → int`

`int → output`

`output[I]... → Types...`

Putting together Idea #2

```
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{};
    }
};
```

Putting together Idea #2

```
#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;                                                 \
    }                                                               \
    /**/
```

Putting together Idea #2

```
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)
```

Putting together Idea #2

```
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;
}
```

Putting together Idea #2

```
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
        ))...
    >{};
}
```

What about pointers to const pointers to volatile pointers to <...> fundamental type?

Taking care of pointers

```
constexpr std::size_t type_to_id(identity<Type>)
```

Taking care of pointers

```
constexpr std::size_t type_to_id(identity<Type>)
```

```
sizeof(std::size_t) * 8 == 64/32 bits
```

Taking care of pointers

```
constexpr std::size_t type_to_id(identity<Type>)
```

```
sizeof(std::size_t) * 8 == 64/32 bits
```

```
fundamental types < 32
```

Taking care of pointers

```
constexpr std::size_t type_to_id(identity<Type>)

sizeof(std::size_t) * 8 == 64/32 bits
```

```
fundamental types < 32

fundamental types require 5 bits
```

Taking care of pointers

```
constexpr std::size_t type_to_id(identity<Type>)

sizeof(std::size_t) * 8 == 64/32 bits
```

```
fundamental types < 32

fundamental types require 5 bits
```

```
not a pointer | pointer | const pointer | volatile pointer | const volatile pointer
```

Taking care of pointers

```
constexpr std::size_t type_to_id(identity<Type>)
```

```
sizeof(std::size_t) * 8 == 64/32 bits
```

```
fundamental types < 32
```

```
fundamental types require 5 bits
```

```
not a pointer | pointer | const pointer | volatile pointer | const volatile pointer
```

```
3 bits
```

Taking care of pointers

```
unsigned char c0;                   // 0b00000000 00000000 00000000 00000001
```

Taking care of pointers

```
unsigned char c0;                  // 0b00000000 00000000 00000000 00000001
unsigned char* c1;                 // 0b00100000 00000000 00000000 00000001
```

Taking care of pointers

```
unsigned char c0;                // 0b00000000 00000000 00000000 00000001
unsigned char* c1;               // 0b00100000 00000000 00000000 00000001
const unsigned char* c2;         // 0b01000000 00000000 00000000 00000001
```

Taking care of pointers

```
unsigned char c0;                // 0b00000000 00000000 00000000 00000001
unsigned char* c1;               // 0b00100000 00000000 00000000 00000001
const unsigned char* c2;         // 0b01000000 00000000 00000000 00000001
const unsigned char** c3;        // 0b01000100 00000000 00000000 00000001
```

Taking care of pointers

```
unsigned char c0;                 // 0b00000000 00000000 00000000 00000001
unsigned char* c1;                // 0b00100000 00000000 00000000 00000001
const unsigned char* c2;          // 0b01000000 00000000 00000000 00000001
const unsigned char** c3;         // 0b01000100 00000000 00000000 00000001
const short** s0;                 // 0b01000100 00000000 00000000 00000111
```

Taking care of pointers

```
template <class Type>
constexpr std::size_t type_to_id(identity<Type*>);

template <class Type>
constexpr std::size_t type_to_id(identity<const Type*>);

template <class Type>
constexpr std::size_t type_to_id(identity<const volatile Type*>);

template <class Type>
constexpr std::size_t type_to_id(identity<volatile Type*>);
```

Taking care of pointers

```
template <std::size_t Index> constexpr auto id_to_type(size_t_<Index >,
if_extension<Index, native_const_ptr_type> = 0) noexcept;

template <std::size_t Index> constexpr auto id_to_type(size_t_<Index >,
if_extension<Index, native_ptr_type> = 0) noexcept;

template <std::size_t Index> constexpr auto id_to_type(size_t_<Index >,
if_extension<Index, native_const_volatile_ptr_type> = 0) noexcept;

template <std::size_t Index> constexpr auto id_to_type(size_t_<Index >,
if_extension<Index, native_volatile_ptr_type> = 0) noexcept;
```

Enums?

Enums

```
template <class Type>
constexpr std::size_t type_to_id(identity<Type>,
    typename std::enable_if<std::is_enum<Type>::value>::type*) noexcept
{
    return type_to_id(identity<
        typename std::underlying_type<Type>::type
    >{});
}
```

Nested structures and classes?

Nested structures and classes

```
template <class Type>
constexpr auto type_to_id(identity<Type>, typename std::enable_if<
    !std::is_enum<Type>::value && !std::is_empty<Type>::value>::type*) noexcept
{
    return array_of_type_ids<Type>(); // Returns array!
}
```

Nested structures and classes

```
// ... in struct ubiq_val
template <class Type>
constexpr operator Type() const noexcept {
    constexpr auto typeids = typeid_conversions::type_to_id(identity<Type>{});
    assign(typeids);
    return Type{};
}
```

Nested structures and classes

```
// ... in struct ubiq_val

constexpr void assign(std::size_t val) const noexcept {
    ref_[I] = val;
}

template <class T>
constexpr void assign(const T& typeids) const noexcept {
    for (std::size_t i = 0; i < T::size(); ++i)
        ref_[I + i] = typeids.data[i]; // ref_[I + I] must not overlap with next field
}
```

Space for storing type info

```
T { args... }

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

Space for storing type info

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

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

```
T needs up to sizeof(T) space for ids
```

Space for storing type info

```
T { args... }

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

```
T needs up to sizeof(T) space for ids

Field needs up to sizeof(Field) space for ids
```

Space for storing type info

```
T { args... }

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

```
T needs up to sizeof(T) space for ids

Field needs up to sizeof(Field) space for ids
```

```
I == sizeof(PrevFields) + …

I ~ offsetof(T, Field)
```

Space for storing type info

```
struct foo1 { short s; unsigned char i; };        // { 7, 0, 1, 0};
```

Space for storing type info

```
struct foo1 { short s; unsigned char i; };          // { 7, 0, 1, 0};

struct foo2 { unsigned char i; foo1 f;};            // {1, 7, 0, 1, 0, 0};
```

Space for storing type info

```
struct foo1 { short s; unsigned char i; };          // { 7, 0, 1, 0};

struct foo2 { unsigned char i; foo1 f;};           // {1, 7, 0, 1, 0, 0};

struct foo3 { foo1 f0; foo1 f; };                   // {7, 0, 1, 0, 7, 0, 1, 0};
```

Space for storing type info

```
struct foo1 { short s; unsigned char i; };        // { 7, 0, 1, 0};

struct foo2 { unsigned char i; foo1 f;};          // {1, 7, 0, 1, 0, 0};

struct foo3 { foo1 f0; foo1 f; };                 // {7, 0, 1, 0, 7, 0, 1, 0};

struct foo4 { foo2 f0; foo1 f; };                 // {1, 7, 0, 1, 0, 0, 7, 0, 1, 0};
```

Nested structures and offsets

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

    return tmp;
}
```

Do we need it?

Advantages and features:

- Comparisons : <, <=, >, >=, !=, ==
- Heterogeneous comparators: flat_less<>, flat_equal<>
- IO stream operators: operator <<, operator>>
- Hashing: flat_hash<>
- 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

Compile times

$$\log_2(\text{sizeof}(T)) + \Sigma \log_2(\text{sizeof}(\text{nested\_structures}))$$

In practice:

- no noticeable slowdown on reasonable structures
- #includes consume more time than metaprogramming stuff

Examples

```
namespace foo {
    struct comparable_struct {
        int i; short s; char data[50]; bool bl; int a,b,c,d,e,f;
    };
} // namespace foo
```

```
std::set<foo::comparable_struct> s;
```

Examples

```
std::set<foo::comparable_struct> s = { /* ... */ };
std::ofstream ofs("dump.txt");

for (auto& a: s)
    ofs << a << '\n';
```

Examples

```
std::set<foo::comparable_struct> s;
std::ifstream ifs("dump.txt");

foo::comparable_struct cs;
while (ifs >> cs) {
    char ignore = {};
    ifs >> ignore;
    s.insert(cs);
}
```

My favourite

```
template <class T>
auto flat_tie(T& val) noexcept;



struct my_struct { int i, short s; };
my_struct s;
flat_tie(s) = std::tuple<int, short>{10, 11};
```

Thank you! Any questions?

https://github.com/apolukhin/magic_get

C++17 Bonus

C++14

```
template <class T>
constexpr auto as_tuple(T& val) noexcept {
  typedef size_t_<fields_count<T>()> fields_count_tag;
  return detail::as_tuple_impl(val, fields_count_tag{});
}
```

C++14

```
template <class T>
constexpr auto as_tuple(T& val) noexcept {
  typedef size_t_<fields_count<T>()> fields_count_tag;
  return detail::as_tuple_impl(val, fields_count_tag{});
}
```

Structured bindings for greater good

```
template <class T>
constexpr auto as_tuple_impl(T&& val, size_t_<1>) noexcept {
  auto& [a] = std::forward<T>(val);
  return detail::make_tuple_of_references(a);
}

template <class T>
constexpr auto as_tuple_impl(T&& val, size_t_<2>) noexcept {
  auto& [a,b] = std::forward<T>(val);
  return detail::make_tuple_of_references(a,b);
}
```

Structured bindings for greater good

```
template <class T>
constexpr auto as_tuple_impl(T&& val, size_t_<1>) noexcept {
  auto& [a] = std::forward<T>(val);
  return detail::make_tuple_of_references(a);
}

template <class T>
constexpr auto as_tuple_impl(T&& val, size_t_<2>) noexcept {
  auto& [a,b] = std::forward<T>(val);
  return detail::make_tuple_of_references(a,b);
}
```

Thank you! Any questions?

https://github.com/apolukhin/magic_get