# Beyond struct: Meta-programming a struct Replacement in C++20

John R. Bandela, MD

# Disclaimer

This is not an official Google library

These opinions are my own

# Struct

```cpp
struct person {
  int id = 1;
  std::string name;
  int score = 0;
};

int main() {
  person p{.id = 1, .name = "John"};
  p.id = 2;
  p.name = "JRB";
  std::cout << p.id << " " << p.name;
}
```

# Limitations of struct

- No static reflection
- No static generation
- No required members
- Designated initializers required to be in order (in C++ not C)

# No static reflection

- Although there is a way to get the types of the struct (see magic_get) there is no way to get the names
- Cannot automatically generate a json serializer/deserializer given a struct.

# No generation

- Cannot create a new struct with names
- Example – given a compile time json string no way to turn that into a struct.

# No required members

- We cannot designate that a member must be specified on construction
- We could use some type of wrapper – but then we run into:
  - There are no transparent wrappers in C++

# Let’s Go Beyond Struct

- C++20
- No Macros

JURASSIK
JURASSIC PARK

# Defining and accessing members

```
using Person = meta_struct<      //
    member<"id", int>,           //
    member<"name", std::string>  //
    >;

Person p;
get<"id">(p) = 1;
get<"name">(p) = "John";

std::cout << get<"id">(p) << " " << get<"name">(p) << "\n";
```

# Fixed string

```
template <std::size_t N>
struct fixed_string {
  constexpr fixed_string(const char (&foo)[N + 1]) {
    std::copy_n(foo, N + 1, data);
  }
  auto operator<=>(const fixed_string&) const = default;
  char data[N + 1] = {};
};
template <std::size_t N>
fixed_string(const char (&str)[N]) -> fixed_string<N - 1>;
```

# Meta Struct

```
template <fixed_string Tag, typename T>
struct member {
  constexpr static auto tag() { return Tag; }
  using element_type = T;
  T value;
};

template <typename... Members>
struct meta_struct : Members... {};
```

# Get

```
template<fixed_string tag, typename T>
decltype(auto) get_impl(member<tag, T>& m) {
  return (m.value);
}

template<fixed_string tag, typename MetaStruct>
decltype(auto) get(MetaStruct&& s) {
  return get_impl<tag>(std::forward<MetaStruct>(s));
}
```

# Defining and accessing members

```
using Person = meta_struct<      //
   member<"id", int>,            //
   member<"name", std::string>  //
   >;

Person p;
get<"id">(p) = 1;
get<"name">(p) = "John";

std::cout << get<"id">(p) << " " << get<"name">(p) << "\n";
```

# Construction

```
using Person = meta_struct<      //
    member<"id", int>,          //
    member<"name", std::string>  //
    >;

 Person p{arg<"id"> = 1, arg<"name"> = "John"};

 std::cout << get<"id">(p) << " " << get<"name">(p) << "\n";
 p = Person{arg<"name"> = "John", arg<"id"> = 1};
 std::cout << get<"id">(p) << " " << get<"name">(p) << "\n";
```

# Arg

```
template <fixed_string Tag, typename T>
struct tag_and_value {
  T value;
};

template <fixed_string Tag>
struct arg_type {
  template <typename T>
  constexpr auto operator=(T t) const {
    return tag_and_value<Tag, T>{std::move(t)};
  }
};

template <fixed_string Tag>
inline constexpr auto arg = arg_type<Tag>{};
```

# Meta Struct Changes

```
template <typename... Members>
struct meta_struct : meta_struct_impl<Members...> {
    using super = meta_struct_impl<Members...>;
  template <typename... TagsAndValues>
  constexpr meta_struct(TagsAndValues... tags_and_values)
      : super(parms(std::move(tags_and_values)...)) {}

};
```

# Parms and member

```
template <typename... TagsAndValues>
struct parms : TagsAndValues... {};

template <typename... Members>
struct meta_struct_impl : Members... {
  template <typename Parms>
  constexpr meta_struct_impl(Parms p)
      : Members(std::move(p))... {}

};

template <fixed_string Tag, typename T>
struct member {
template <typename OtherT>
  constexpr member(tag_and_value<Tag, OtherT> tv)
          : value(std::move(tv.value)) {}
};
```

# Construction

```
using Person = meta_struct<      //
    member<"id", int>,          //
    member<"name", std::string>  //
    >;

 Person p{arg<"id"> = 1, arg<"name"> = "John"};

 std::cout << get<"id">(p) << " " << get<"name">(p) << "\n";
 p = Person{arg<"name"> = "John", arg<"id"> = 1};
 std::cout << get<"id">(p) << " " << get<"name">(p) << "\n";
```

# Construction

```
using Person = meta_struct<      //
    member<"id", int>,          //
    member<"name", std::string>  //
    >;

 Person p{arg<"id"> = 1, arg<"name"> = "John"};

 std::cout << get<"id">(p) << " " << get<"name">(p) << "\n";
 p = Person{arg<"name"> = "John", arg<"id"> = 1};
 std::cout << get<"id">(p) << " " << get<"name">(p) << "\n";
```

# Default Initialization

- We want to provide support for specifying the default value of a member item
- However, we cannot just add it as a template parameter, not all types are compatible with template parameters
- Instead we use a lambda

# Default Initialization With Constant

```
using Person = meta_struct<                              //
    member<"id", int>,                               //
    member<"name", std::string, [] { return "John"; }>  //
    >;

 Person p;

 std::cout << get<"id">(p) << " " << get<"name">(p) << "\n";
```

# Member

```
template <typename T>
struct default_init {
constexpr auto operator()() const {
    if constexpr (std::is_default_constructible_v<T>) {
      return T{};
    }
  }
};

template <fixed_string Tag, typename T, auto Init = default_init<T>()>
struct member {
  constexpr member() : value(Init()) {}
};
```

# Default Initialization With Constant

```
using Person = meta_struct<                         //
    member<"id", int>,                              //
    member<"name", std::string, [] { return "John"; }>  //
    >;

Person p;

std::cout << get<"id">(p) << " " << get<"name">(p) << "\n";
```

# Default Initialization with Expression

- Sometimes we need the default initialization to depend on another member
- We use a lambda that takes a self parameter

# Default Init with Expression

```
int main() {
  using Person = meta_struct<                                          //
    member<"id", int>,                                              //
    member<"score", int, [](auto& self) { return get<"id">(self) + 1; }>,  //
    member<"name", std::string, [] { return "John"; }>                //
    >;

  Person p;

  std::cout << get<"id">(p) << " " << get<"name">(p) << " " << get<"score">(p)
          << "\n";
}
```

# Member

```
template <fixed_string Tag, typename T, auto Init = default_init<T>()>
struct member {
 constexpr static auto tag() { return Tag; }
 constexpr static auto init() { return Init; }
 using element_type = T;
 T value;
 template <typename OtherT>
 constexpr member(tag_and_value<Tag, OtherT> tv)
   : value(std::move(tv.value)) {}

 template <typename Self>
 constexpr member(Self& self) : value(call_init<T>(self, Init)) {}
};
```

# Call Init

```
template <typename T, typename Self, typename F>
auto call_init(Self&, F& f) requires(requires {
  { f() } -> std::convertible_to<T>;
}) {
  return f();
}

template <typename T, typename Self, typename F>
auto call_init(Self& self, F& f) requires(requires {
  { f(self) } -> std::convertible_to<T>;
}) {
  return f(self);
}
```

# Meta Struct Impl

```
template <typename... Members>
struct meta_struct_impl : Members... {
  template <typename Parms>
  constexpr meta_struct_impl(Parms p) : Members(std::move(p))... {}

  constexpr meta_struct_impl() : Members(*this)... {}
};
```

# Default Init with Expression

```
int main() {
 using Person = meta_struct<                                          //
   member<"id", int>,                                             //
   member<"score", int, [](auto& self) { return get<"id">(self) + 1; }>,  //
   member<"name", std::string, [] { return "John"; }>                //
   >;

 Person p;

 std::cout << get<"id">(p) << " " << get<"name">(p) << " " << get<"score">(p)
      << "\n";
}
```

# Optional Arguments

```
using Person = meta_struct<                                              //
    member<"id", int>,                                               //
    member<"score", int, [](auto& self) { return get<"id">(self) + 1; }>,  //
    member<"name", std::string, [] { return "John"; }>                 //
    >;

 Person p{arg<"id"> = 2};

 Person p2{arg<"id"> = 2, arg<"score"> = std::optional<int>()};

 Person p3{arg<"id"> = 2, arg<"score"> = std::optional<int>(500)};
```

# Member

```
template <fixed_string Tag, typename T, auto Init = default_init<T>()>
struct member {
template <typename Self, typename OtherT>
 constexpr member(Self&, tag_and_value<Tag, OtherT> tv)
   : value(std::move(tv.value)) {}

 template <typename Self>
 constexpr member(Self& self) : value(call_init<T>(self, Init)) {}
 template <typename Self>
 constexpr member(Self& self, no_conversion)
   : value(call_init<T>(self, Init)) {}
 template <typename Self>
 constexpr member(Self& self, tag_and_value<Tag, std::optional<T>> tv_or)
   : value(tv_or.value.has_value() ? std::move(*tv_or.value)
                   : call_init<T>(self, Init)) {}

};
```

# Parms

```
struct no_conversion {};

template <typename... TagsAndValues>
struct parms : TagsAndValues... {
  constexpr operator no_conversion() const { return no_conversion{}; }
};
```

# Optional Arguments

```
using Person = meta_struct<                                    //
    member<"id", int>,                                       //
    member<"score", int, [](auto& self) { return get<"id">(self) + 1; }>,  //
    member<"name", std::string, [] { return "John"; }>                  //
    >;

 Person p{arg<"id"> = 2};

 Person p2{arg<"id"> = 2, arg<"score"> = std::optional<int>()};

 Person p3{arg<"id"> = 2, arg<"score"> = std::optional<int>(500)};
```

# Required members

```
using Person = meta_struct<                                    //
    member<"id", int, required>,
    member<"score", int, [](auto& self) { return get<"id">(self) + 1; }>,  //
    member<"name", std::string, required>                      //
    >;

 Person p{arg<"id"> = 2, arg<"name"> = “John"};
```

# Required

```
inline constexpr auto required = [] {};

template <typename T, typename Self, typename F>
auto call_init(Self& self, F& f) requires(requires {
  { f() } -> std::same_as<void>;
}) {
  static_assert(!std::is_same_v<decltype(f()), void>,
                "Required argument not specified");
}
```

# Required members

```
using Person = meta_struct<                                    //
    member<"id", int, required>,
    member<"score", int, [](auto& self) { return get<"id">(self) + 1; }>,  //
    member<"name", std::string, required>                      //
    >;

 Person p{arg<"id"> = 2, arg<"name"> = “John"};
```

# Reflection

```
using Person = meta_struct<                              //
    member<"id", int>,                               //
    member<"score", int, [](auto& self) { return get<"id">(self) + 1; }>,  //
    member<"name", std::string, [] { return "John"; }>          //
    >;
 meta_struct_apply<Person>([]<typename... M>(M * ...) {
  std::cout << "The tags are: ";
  ((std::cout << M::tag().sv() << " "), ...);
  std::cout << "\n";
 });
 Person p;
 meta_struct_apply(
    [&](const auto&... m) {
      ((std::cout << m.tag().sv() << ":" << m.value << "\n"), ...);
    },
    p);
```

# Fixed String

```
template <std::size_t N>
struct fixed_string {
 constexpr fixed_string(const char (&foo)[N + 1]) {
  std::copy_n(foo, N + 1, data);
 }
 constexpr std::string_view sv() const { return std::string_view(data); }

 auto operator<=>(const fixed_string&) const = default;
 char data[N + 1] = {};
};
```

# Member

```
template <fixed_string Tag, typename T, auto Init = default_init<T>()>
struct member {
  constexpr static auto tag() { return Tag; }
  constexpr static auto init() { return Init; }
  using element_type = T;
  T value;
  template <typename OtherT>
  constexpr member(tag_and_value<Tag, OtherT> tv)
      : value(std::move(tv.value)) {}

  template <typename Self>
  constexpr member(Self& self) : value(call_init<T>(self, Init)) {}
};
```

# Meta Struct Apply Object version

```
template <typename F, typename... Members>
constexpr decltype(auto) meta_struct_apply(
    F&& f, meta_struct_impl<Members...>& m) {
  return std::forward<F>(f)(static_cast<Members&>(m)...);
}
```

# Meta Struct Apply Type Version

```
template <typename MetaStructImpl>
struct apply_static_impl;

template <typename... Members>
struct apply_static_impl<meta_struct_impl<Members...>> {
  template <typename F>
  constexpr static decltype(auto) apply(F&& f) {
    return f(static_cast<Members*>(nullptr)...);
  }
};

template <typename MetaStruct, typename F>
auto meta_struct_apply(F&& f) {
  return apply_static_impl<typename MetaStruct::super>::apply(
      std::forward<F>(f));
}
```

# Reflection

```
using Person = meta_struct<                                    //
    member<"id", int>,                                         //
    member<"score", int, [](auto& self) { return get<"id">(self) + 1; }>,  //
    member<"name", std::string, [] { return "John"; }>               //
    >;
 meta_struct_apply<Person>([]<typename... M>(M * ...) {
  std::cout << "The tags are: ";
  ((std::cout << M::tag().sv() << " "), ...);
  std::cout << "\n";
 });
 Person p;
 meta_struct_apply(
    [&](const auto&... m) {
      ((std::cout << m.tag().sv() << ":" << m.value << "\n"), ...);
    },
    p);
```

# Other Features

# Subset conversions

```
int main() {
 using Person = meta_struct<                                    //
    member<"id", int, required>,                                //
    member<"name", std::string, required>,                           //
    member<"score", int, [](auto& self) { return get<"id">(self) + 1; }>  //
    >;


 Person p{arg<"id"> = 2, arg<"name"> = "John"};

 using NameAndId = meta_struct<       //
    member<"name", std::string>,      //
    member<"id", int>                 //
    >;

 NameAndId n = p;
}
```

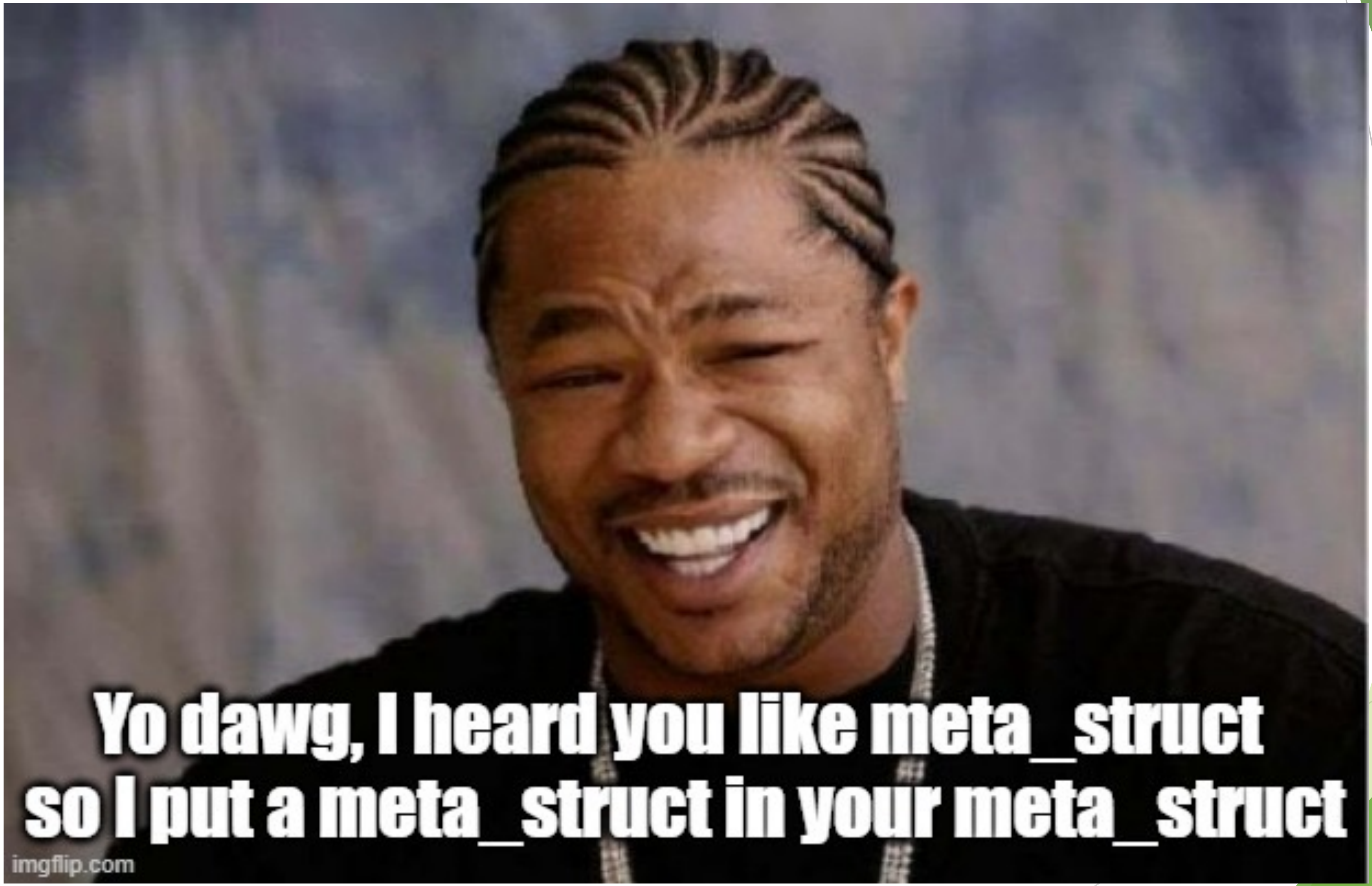
# Attributes

```
enum class encoding : int { fixed = 0, variable = 1 };

int main() {
 using Person = meta_struct<                         //
   member<"id", int, required, {arg<"encoding"> = encoding::variable}>,  //
   member<"name", std::string, required>,                //
   member<"score", int, [](auto& self) { return get<"id">(self) + 1; }>  //
   >;

 constexpr auto attributes = get_attributes<"id", Person>();

 if constexpr (has<"encoding">(attributes) &&
        get<"encoding">(attributes) == encoding::variable) {
  std::cout << "Encoding was variable";
 } else {
  std::cout << "Encoding was fixed";
 }
```


Yo dawg, I heard you like meta_struct
so I put a meta_struct in your meta_struct
imgflip.com

JURASSI
JURASSIC PARK

# Applications

# Named arguments

```
using substr_args = meta_struct<                   //
    member<"str", const std::string&, required>,      //
    member<"offset", std::size_t, [] { return 0; }>,  //
    member<"count", std::size_t,
        [](auto& self) {
          return get<"str">(self).size() - get<"offset">(self);
        }>  //
    >;
auto substr(substr_args args) {
  return get<"str">(args).substr(get<"offset">(args), get<"count">(args));
}

int main() {
  std::string s = "Hello World";
  auto pos = s.find(' ');
  auto all = substr({arg<"str"> = std::ref(s)});
  auto first = substr({arg<"str"> = std::ref(s), arg<"count"> = pos});
  auto second = substr({arg<"str"> = std::ref(s), arg<"offset"> = pos + 1});
}
```

# Array of Structures vs Structure of Arrays

```
struct Person {
  int id = 0;
  std::string name;
  int score = 0;
};

std::vector<Person> persons_aos;

struct Persons {
  std::vector<int> id;
  std::vector<std::string> name;
  std::vector<int> score;
};

Persons persons_soa;
```

# Array of Structures vs Structure of Arrays

| | | | |
|---|---|---|---|
| Id | Name | Score | Id |
| Name | Score | Id | Name |
| Score | Id | Name | Score |

| | | | |
|---|---|---|---|
| Id | Id | Id | Id |
| Name | Name | Name | Name |
| Score | Score | Score | Score |

# Soa Vector

```
using Person = meta_struct<                                   //  member<"id", int, required>,  //
   member<"name", std::string, required>,                             //
   member<"score", int>  //
   >;
  soa_vector<Person> v;

  v.push_back(Person{arg<"name"> = "John", arg<"id"> = 1, arg<"score"> = 10});
  v.push_back(Person{arg<"name"> = "Lisa", arg<"id"> = 2, tag<"score"> = 12});
  auto person_ref = v[1];
  assert(get<"name">(person_ref) == "Lisa");

  std::span<int> scores = get<"score">(v);
  assert(*std::max_element(scores.begin(),scores.end()) == 12);
```

# Duck Typing for Structs

- We may have a function that doesn’t care about the type of the struct, but just the name and types of the members.
- We can get duck typing in C++ using templates

# Duck Typing with Templates

```
template <typename P>
void display_person(const P& p) {
  std::cout << "The person has an id of " << p.id << " and name " << p.name
            << " and scored " << p.score << "\n";
}
```

# Duck Typing with Templates

```
struct MyPerson {
  std::string name;
  int id = 0;
  int score = 0;
};

struct YourPerson {
  int id = 0;
  int score = 0;
  std::string name;
};

int main() {
  MyPerson p1;
  YourPerson p2;

  display_person(p1);
  display_person(p2);
}
```

# Duck Typing for Structs

- We require a template function
- This means we can’t use separate compilation

# Duck Typing for meta_struct

```
using person_ref = meta_struct<                 //
   member<"name", std::string_view, required>,  //
   member<"id", const int&, required>,         //
   member<"score", const int&, required>       //
   >;

void display_person_meta(person_ref p) {
  std::cout << "The person has an id of " << get<"id">(p) << " and name "
         << get<"name">(p) << " and scored " << get<"score">(p) << "\n";
}
```

# Duck Typing for meta_struct

```
using MyPersonMeta = meta_struct<  //
    member<"id", int>,             //
    member<"name", std::string>,   //
    member<"score", int>           //
    >;

using YourPersonMeta = meta_struct<  //
    member<"id", int>,               //
    member<"score", int>,            //
    member<"name", std::string>      //
    >;

int main() {
  MyPersonMeta pm1;
  YourPersonMeta pm2;

  display_person_meta(pm1);
  display_person_meta(pm2);
}
```

# Duck Typing for meta_struct

- You can actually use a non-template function that can be separately compiled
- The order of the members or extra members does not matter
- As long has the meta_struct has those names and convertible types, it can be passed to the function.

# Generating Meta Structs from Compile Time Strings

# Compile Time Regular Expressions

https://github.com/hanickadot/compile-time-regular-expressions

```
struct date {
  std::string_view year;
  std::string_view month;
  std::string_view day;
 };

 std::optional<date> extract_date(std::string_view s) noexcept {
  using namespace ctre::literals;
  if (auto [whole, year, month, day] =
      ctre::match<"(\\d{4})/(\\d{1,2})/(\\d{1,2})">(s);
    whole) {
   return date{year, month, day};
  } else {
   return std::nullopt;
  }
 }
```

# Compile Time Regular Expressions

Possible API (not implemented)

```
using date = meta_struct<              //
    member<"year", std::string_view>,   //
    member<"month", std::string_view>,  //
    member<"day", std::string_view>     //
    >;

std::optional<date> extract_date(std::string_view s) noexcept {
  using namespace ctre::literals;
  if (auto [whole, groups] =
        ctre::match<"(?<year>\\d{4})/(?<month>\\d{1,2})/(?<day>\\d{1,2})">(s);
    whole) {
   return groups;
  } else {
   return std::nullopt;
  }
}
```

# SQL

C++20 ❤️ SQL

https://github.com/google/cpp-from-the-sky-down/tree/master/meta_struct_20/cppcon_version