Reflection Based Libraries to Look Forward To

SAKSHAM SHARMA

Reflection based libraries to look forward to

Saksham Sharma

CppCon 2024

Why me

- Director, Quant Research Tech at Tower Research Capital
  - High frequency trading firm based out of NYC
- Develop low latency trading systems
  - C++
- Develop high throughput research systems
  - C++ and Python
- Member of WG21 - ISO C++ Standardization Committee
  - Not an expert

Why me

- Talk a lot on Python and C++ (often in the same breath)
- Past life:
  - Program analysis research and functional programming
- Love performance, software abstractions, and clean APIs

Overview

- What is reflection
- Reflection in other languages (Go, Python, Java)
- Reflection in C++ as per P2996
  - Syntax and examples
- Reflection libraries!
  - Python bindings
  - ABI hashing (boost::abi_hash?)
  - A duck-typed std::any (boost::virtual_any?)
- Alternatives ways to achieve “reflection”

Reflection

In code.

Reflection?

Write code to

- Access information about other “code”
- Operate on that information

```
1 class MyClass {
2     int a;
3     int b;
4 };
5
6 for (auto member_info : gimme_class_members<MyClass>()) {
7     std::cout << "member - " << member_info.name() << std::endl;
8 }
```

How is this different from metaprogramming?

Reflection vs metaprogramming

- Not much :)
- Operate on values instead of types
  - Syntactic difference, not semantic
- Compiler can provide richer “information” about code
  - Some of it is already there
  - A neat and consistent “bag” of features to expand in the future

Other languages

Venturing into some alien worlds.

Python

My favorite example: Python

At runtime your code can

- Access class layout
- Modify your class to instrument function calls.
- Change what it means to access a field on an object.
- Add or remove methods or attributes from any object.

Reflecting Python

```python
def modify_cls(cls):
    if not hasattr(cls, "copy"):
        return cls
    orig_copy = cls.copy

    def _wrapped_copy(obj):
        print("Calling wrapped copy")
        attrs = obj.__dict__.keys()
        print("Attributes: " +
              " ".join(attrs))
        result = orig_copy(obj)
        return result

    cls.copy = _wrapped_copy
```

```python
class MyClass:
    def __init__(self, x):
        self.x = x
    def copy(self):
        return MyClass(self.x)

>>> modify_cls(MyClass)
>>> MyClass(2).copy()
Calling wrapped copy
Attributes: x
<__main__.MyClass object at 0x7f1a9
```

GoLang

Reflecting GoLang

- Golang is a compiled but duck-typed language
  - Well, structurally typed, but close enough
- Runtime reflection similar to python.
  - No special compile time constructs
  - Provides a package reflect to get “reflection values”.

Reflecting GoLang

```
type T struct {
    A string
    B int
}
t := T{"CppCon!", 24}

s := reflect.ValueOf(&t).Elem()
typeOfT := s.Type()

for i := 0; i < s.NumField(); i++ {
    f := s.Field(i)
    fmt.Printf("%d: %s %s = %v\n", i, typeOfT.Field(i).Name,
               f.Type(), f.Interface())
}
// 0: A string = CppCon!
// 1: B int = 24
```

*https://go.dev/blog/laws-of-reflection

Java

Reflecting Java

- Surprisingly, quite similar in feel to Python and Go.
- Reflection is “runtime”, in the sense that the object type saves type information accessible at program runtime.
- java.lang.reflect

Reflecting Java

```
1  // Surprising, lookup types with string!
2  Class cls = Class.forName("method1");
3  Method methlist[] = cls.getDeclaredMethods();
4
5  for (int i = 0; i < methlist.length; i++) {
6     Method m = methlist[i];
7     System.out.println("name = " + m.getName());
8     System.out.println("decl class = " + m.getDeclaringClass());
9  }
10
11 Class pvec[] = m.getParameterTypes();
12 for (int j = 0; j < pvec.length; j++)
13    System.out.println("param #" + j + " " + pvec[j]);
```

*https://www.oracle.com/technical-resources/articles/java/javareflection.html

Scary?

String to types!

Runtime cost!

Zero cost abstractions

- Prior examples had “reflection objects” at runtime.
- Despite the temptation, we must do compile time
  - RTTI, anyone?
- We can lean on consteval to avoid RTTI!

A new hope!

P2996

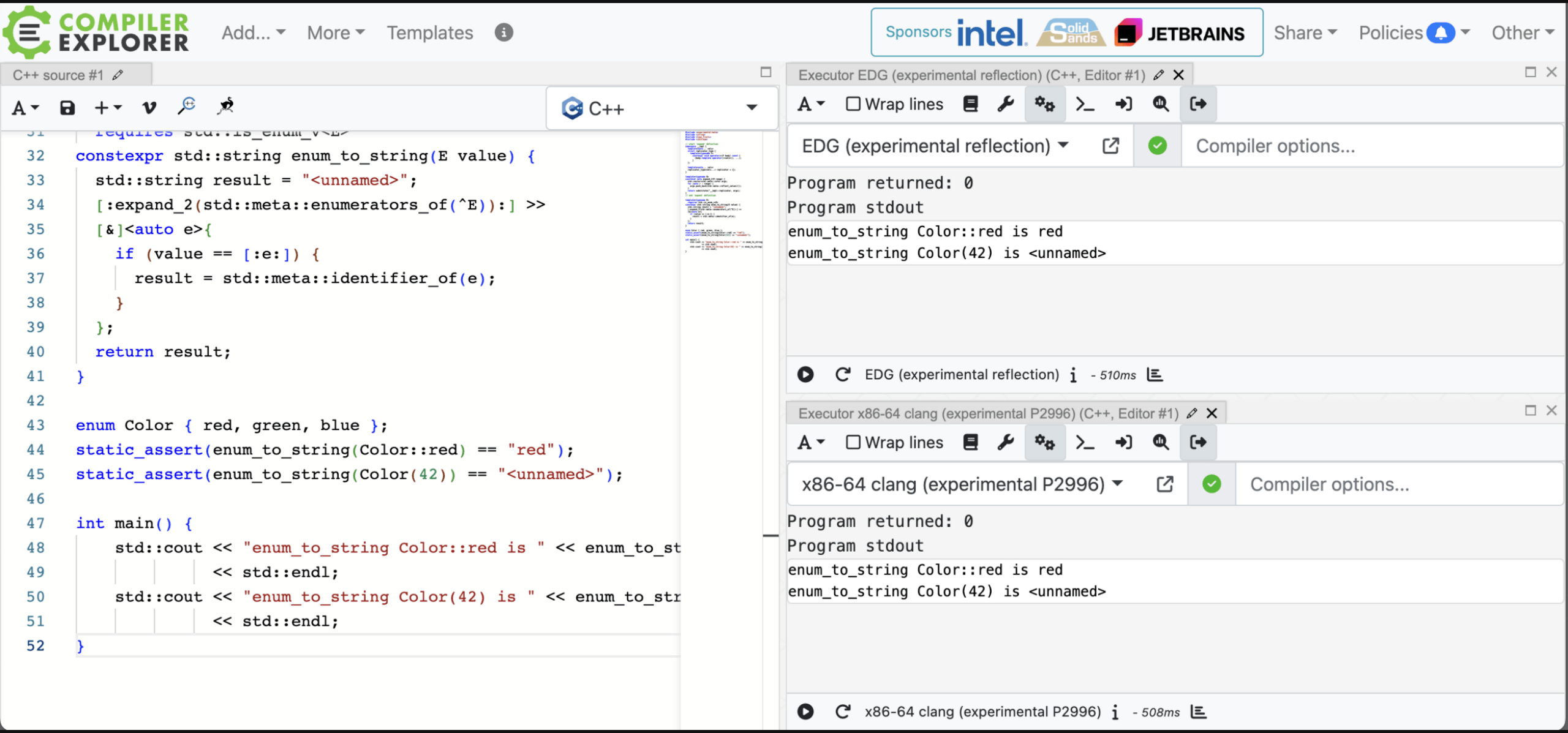
We have a design!

- P2996 - the paper
  - Wyatt Childers, Peter Dimov, Dan Katz, Barry Revzin, Andrew Sutton, Faisal Vali, Daveed Vandevoorde
- Design approved by EWG at St. Louis
- Under review by LEWG for library design aspect

- Prior work: Reflection TS
  - David Sankel - Type based reflection (2021)
  - P1240 - Scalable Reflection in C++ (2018-2022)
  - ... much more ...

We have an implementation(s)!

- Two working implementations already!
  - Edison Design Group (EDG) compiler on Godbolt
  - Clang fork by Bloomberg on Godbolt and GitHub
  - Godbolt link for both: https://godbolt.org/z/fPo4bsG83


COMPILER EXPLORER
Add...
More
Templates
Sponsors intel Solid Sands JETBRAINS
Share
Policies
Other
C++ source #1
C++
requires std::is_enum_v<E>
constexpr std::string enum_to_string(E value) {
std::string result = "<unnamed>";
[:expand_2(std::meta::enumerators_of(^E)):] >>
[&]<auto e>{
if (value == [:e:]) {
result = std::meta::identifier_of(e);
}
};
return result;
}
enum Color { red, green, blue };
static_assert(enum_to_string(Color::red) == "red");
static_assert(enum_to_string(Color(42)) == "<unnamed>");
int main() {
std::cout << "enum_to_string Color::red is " << enum_to_st
<< std::endl;
std::cout << "enum_to_string Color(42) is " << enum_to_str
<< std::endl;
}
Executor EDG (experimental reflection) (C++, Editor #1)
Wrap lines
EDG (experimental reflection)
Compiler options...
Program returned: 0
Program stdout
enum_to_string Color::red is red
enum_to_string Color(42) is <unnamed>
EDG (experimental reflection) - 510ms
Executor x86-64 clang (experimental P2996) (C++, Editor #1)
Wrap lines
x86-64 clang (experimental P2996)
Compiler options...
Program returned: 0
Program stdout
enum_to_string Color::red is red
enum_to_string Color(42) is <unnamed>
x86-64 clang (experimental P2996) - 508ms

We have a reflection operator!

unary operator ^

“Lifts” into reflection land

- function
- type
- variable and friends
- non-static data member
- template
- namespace
- ...

```
1 using namespace std::meta;
2
3 auto magic_reflection_method(info obj) {
4   .. do something with “meta” info ..
5 }
6
7 // ^Lift^ MyType to reflection land
8 auto result =
9     magic_reflection_method(^MyType);
```

We have a reflection operator!

`unary operator ^^`

(perhaps)

```cpp
using namespace std::meta;

auto magic_reflection_method(info obj) {
  .. do something with “meta” info ..
}

// ^^Lift^^ MyType to reflection land
auto result =
    magic_reflection_method(^^MyType);
```

We have a std::meta::info!

A formless type

Describes all meta-info about a type / member / method etc.

```
1  using namespace std::meta;
2
3  // std::meta::info with std::meta
4  // associated namespace
5  auto magic_reflection_method(info obj)
6   .. do something with “meta” info ..
7  }
8
9  auto result =
10     magic_reflection_method(^MyType);
```

We have a splice operator!

```
[: r :] =>
```

Takes a std::meta::info
Constant expression

Splices it back into
your regular code

```cpp
struct MyStruct {
   static int a;
   static int b;
};

constexpr auto elem = ^MyStruct::a;
std::cout << [:elem:] << std::endl;
```

We have a std::meta::define_class!

We can create classes in thin air!
(almost)

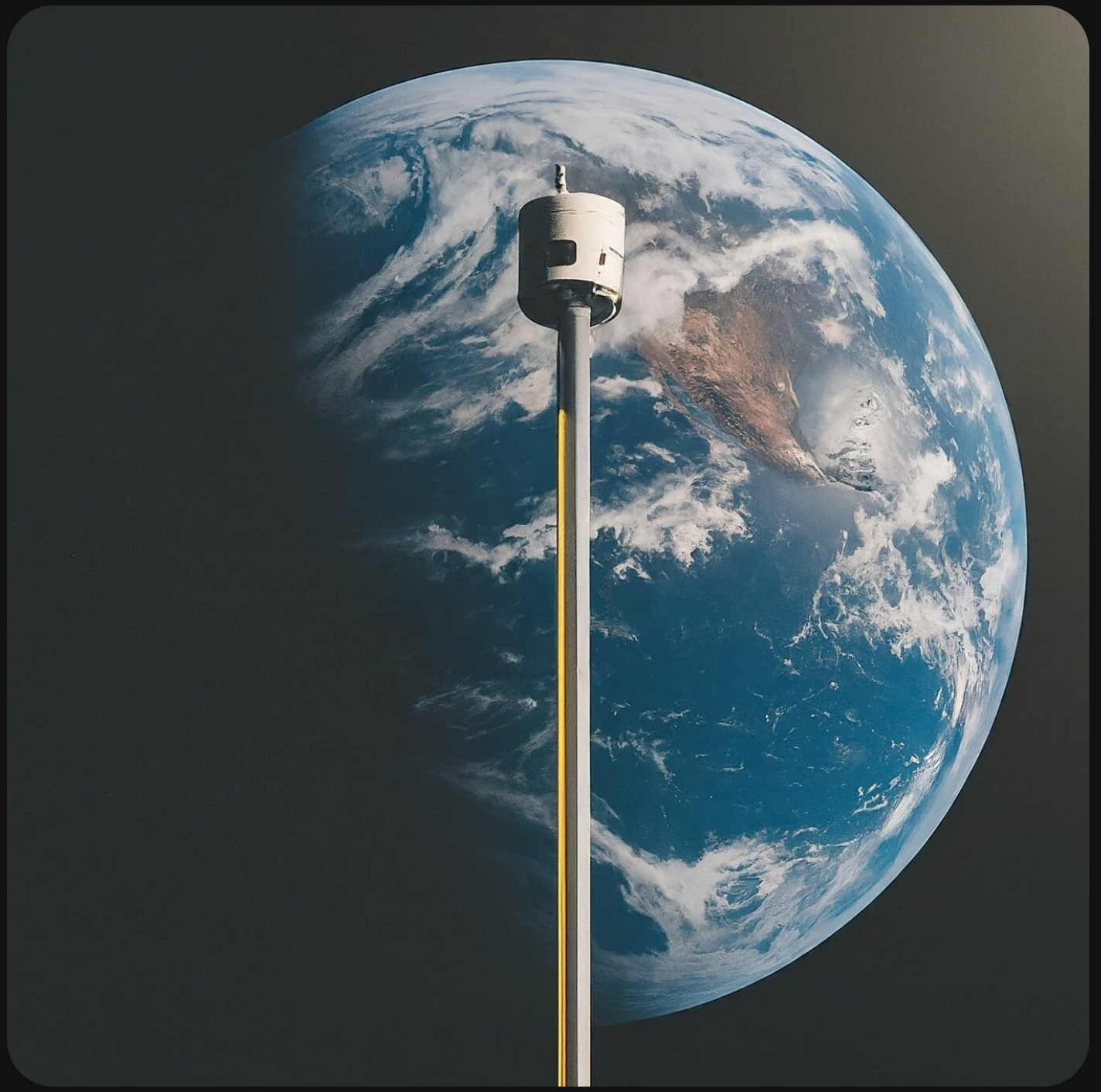
- Need to declare it first
- Need to name all members and their types.

```
1 // Equivalent to
2
3 struct S;
4 struct S {
5    alignas(64) int i;
6    alignas(64) int j;
7 };
```

We have a lot of consteval methods to go with it

```
namespace std::meta {
  // [meta.reflection.names], reflection names and locations
  consteval string_view identifier_of(info r);
  consteval string_view u8identifier_of(info r);
  ...
  consteval auto is_function(info entity) -> bool;
  consteval auto is_variable(info entity) -> bool;
  consteval auto is_type(info entity) -> bool;
  ...
  // [meta.reflection.member.queries], reflection member queries
  consteval vector<info> members_of(info type);
  consteval vector<info> bases_of(info type);
  consteval vector<info> static_data_members_of(info type);
  consteval vector<info> nonstatic_data_members_of(info type);
}
```

Elevator pitch



Enum to String

```cpp
enum class MyEnum { VALUE_1, VALUE_2, MAX_VALUES };

template <typename EnumT> constexpr std::string enum_to_string(EnumT enum_value) {
    std::string result = "<unnamed>";
    [:expand(std::meta::enumerators_of(^E)):] >> [&]<auto e>{
        if (value == [:e:]) {
            result = std::meta::identifier_of(e);
        }
    };
    return result;
}

template <typename EnumT> constexpr EnumT string_to_enum(std::string enum_str) {
  // some blackmagic
}
```

Easy cmdline parsing

```cpp
struct MyOpts {
    std::string file_name = "input.txt";  // Option "--file_name <string>"
    int count = 1;                        // Option "--count <int>"
};

int main(int argc, char* argv[]) {
    MyOpts opts = parse_options<MyOpts>(
      std::vector<std::string_view>(argv + 1, argv + argc)
    );
}
```

*Copied from P2996

Easy cmdline parsing

```
1  template <typename Opts> auto parse_options(ArgT args) -> Opts {
2      Opts opts;
3      template for (constexpr auto dm : nonstatic_data_members_of (^Opts)) {
4          auto it = std::ranges::find_if(args, ... match string ...);
5          auto iss = std::ispanstream(it[1]);
6          if (iss >> opts.[:dm:]; !iss) {
7              std::print(stderr, "Failed to parse option {}\n", *it);
8              std::exit(EXIT_FAILURE);
9          }
10     }
11     return opts;
12 }
```

*Copied from P2996

Array of structs to struct of arrays

```cpp
struct point {
   float x;
   float y;
};
using points = struct_of_arrays<point, 30>;

// equivalent to:
// struct points {
//   std::array<float, 30> x;
//   std::array<float, 30> y;
// };
```

*Copied from P2996

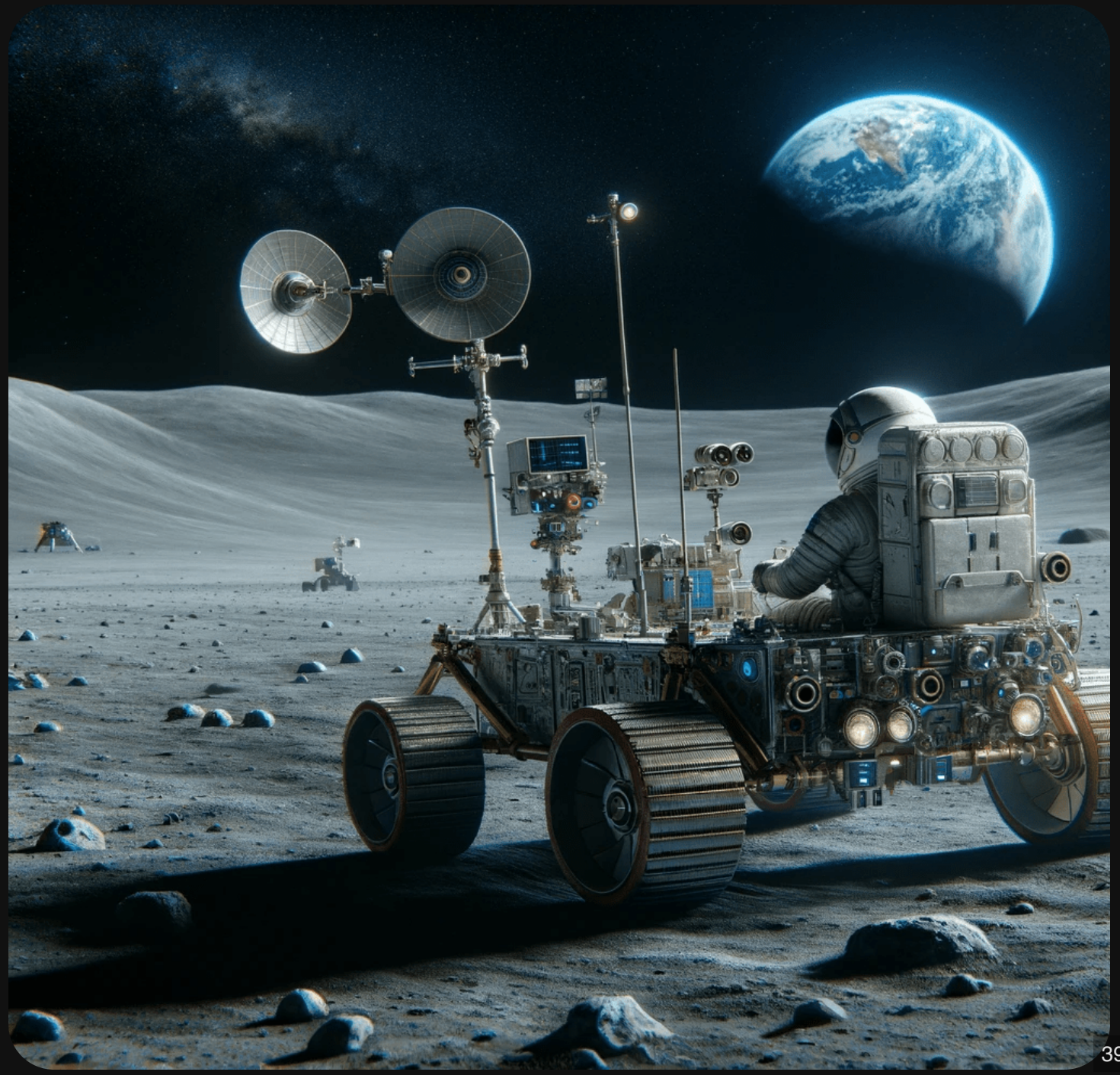
Array of structs to struct of arrays

```
1  template <typename T, std::size_t N>
2  struct struct_of_arrays_impl;
3
4  consteval auto make_struct_of_arrays(std::meta::info type,
5                                       std::meta::info N) -> std::meta::info {
6    std::vector<std::meta::info> old_members = nonstatic_data_members_of(type);
7    std::vector<std::meta::info> new_members = {};
8    for (std::meta::info member : old_members) {
9      new_members.push_back(...);
10   }
11   return std::meta::define_class(
12     substitute(^struct_of_arrays_impl, {type, N}),
13     new_members);
14 }
15
16 template <typename T, size_t N>
17 using struct_of_arrays = [: make_struct_of_arrays(^T, ^N) :];
```

*Copied from P2996

Pretty nice huh?

How do we use this new superpower?



Reflection libaries!

- Reflection is a really powerful language feature
  - With great power comes great responsibility
- Easier to write general-purpose / boilerplate-reducing libraries
- Solve multiple pain-points through a single feature
  - The hallmark of a useful language feature

Reflection libaries!

- Libraries would require fewer redundant inputs from the user
  - No need to enumerate all members in your struct for CLI parsing.
  - Libraries could take a spec to parse and return a struct to you storing that spec?

- Beginner user experience improves upon finding easier-to-use libs

Reflection libaries!

- Hopefully users of the language don't use reflection much
  - People writing business logic
  - Beginners
- And hopefully library designers design great APIs that never even let users look at reflection code
  - This is possible!

Now let's do some actual library design

- Automatic python bindings
- Automatic type ABI hash
- A better std::any with duck typing

Python bindings

A fan favorite, and one of the most common wishlist items

- We have a C++ class that we'd like to expose to Python
- Background:
  - You can run python and C++ code in the same process.
  - CPython is a C library and application.
  - Can run in the same process that is running your C++ code
  - We have to define how to "expose" C++ objects in python

Python bindings

```
 1 struct Item {
 2    int id;
 3    PyObject* getPyValue() const;
 4 };
 5 struct Row {
 6    const auto& items() const { return items_; }
 7    auto nanotime() const { return nanotime_; }
 8 };
 9
10 class RowReader : public BinaryListener {
11  public:
12    RowReader(const std::string& filename)
13        : reader_(filename);
14    std::string getIdName(int id) const;
15    const auto& getRows();
16 };
```

Python bindings - what we don't want

- Most of the code is just repetition of what we already know
- Non-obvious stuff:
  - Whether a function's return type should be wrapped as a reference or a value copy
  - How to name a type in Python

```
1  BOOST_PYTHON_MODULE(binary_reader_bpy) {
2      class_<Item>("Item", no_init)
3          .def_readonly("id", &Item::id)
4          .def("value", &Item::getPyValue);
5
6      class_<Row, std::shared_ptr<Row>>("Row", no_
7          .def("nanotime", &Row::nanotime)
8          .def("items", &Row::items,
9               return_internal_reference<>());
10
11     class_<RowReader, boost::noncopyable>(
12       "RowReader", init<std::string>(arg("filena
13     )
14         .def("getRows", &RowReader::getRows)
15         .def("getIdName", &RowReader::getIdName)
16 }
```

Python bindings - what we want

This is what we can aspire to

```
1 BOOST_PYTHON_MODULE(binary_reader_bpy) {
2     make_python_type<Item>();
3
4     make_python_type<Row>();
5
6     make_python_type<RowReader>();
7 }
```

Python bindings work easy peasy

```
1  template <typename T> object make_python_type() {
2      std::string cls_name{meta::identifier_of (^T)};
3      auto type_obj = class_<T>(cls_name.c_str(), no_init);
4
5      [:expand(meta::members_of (^T)):] >> [&]<auto e> {
6          if constexpr(!meta::is_public(e)) {
7              return;
8          }
9
10         std::string name{meta::identifier_of(e)};
11         if constexpr(meta::is_nonstatic_data_member(e)) {
12             type_obj.def_readwrite(name.c_str(), &[:e:]);
13         }
14
15         if constexpr(meta::is_function(e) && !meta::is_constructor(e) &&
16                      !meta::is_destructor(e)) {
17             using return_t = typename return_type<decltype(&[:e:])>::type;
```

Python bindings with reflection

- Can be done
- Still need to figure out how to customize things that can’t be defaulted properly
  - Return types with reference
  - Picking overloads for functions
- How to name types (and member functions in case of overloads)
- Docstrings

Customizing the default behavior

Specifying customizations manually while binding:

- Not ideal, but has benefits
- Can annotate even if you don't have control on source code

```
1 constexpr auto customizations = {
2     {^Row::items,
3      return_value_policy::reference_internal},
4     ...
5 };
6
7 make_python_type<Row>(customizations);
```

*mentioned in P2911

Customizing the default behavior

User defined attributes?

- Proposed in P1887, discussed in P2911
- Helpful in tagging information at the place of definition
- Requires control on the source code of the type

```
1 class Row {
2 public:
3     [[return_policy("reference_internal")]]
4     const auto& items() { ... }
5 };
```

*mentioned in P2911

User defined attributes - a digression

One of my favorite features from Golang

- Adding information at the point of definition
- Has its place, as opposed to adding code at the time of use
- Beautiful when done well

```
1 type User struct {
2     Name string `json:"name" required:"true"`
3 }
4 user := User{"John"}
5 field, ok := reflect.TypeOf(user).Elem().FieldByName("Name")
6 fmt.Println(field.Tag, field.Tag.Get("required"))
7 // json:"name" required:"true" true
```

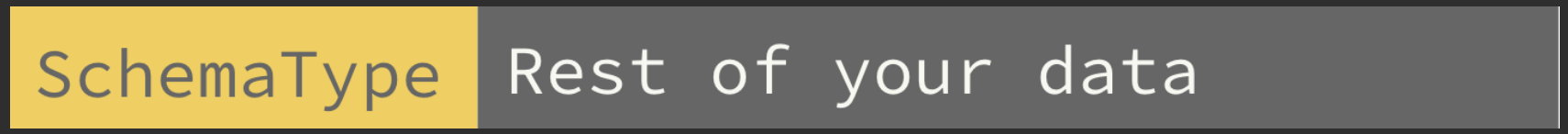
Python bindings - Summary

- Default behavior can be done easily
- Customizations are trickier / not DRY
  - We have two potential approaches

ABI hashing

Setting the stage - Why is a type's hash useful?

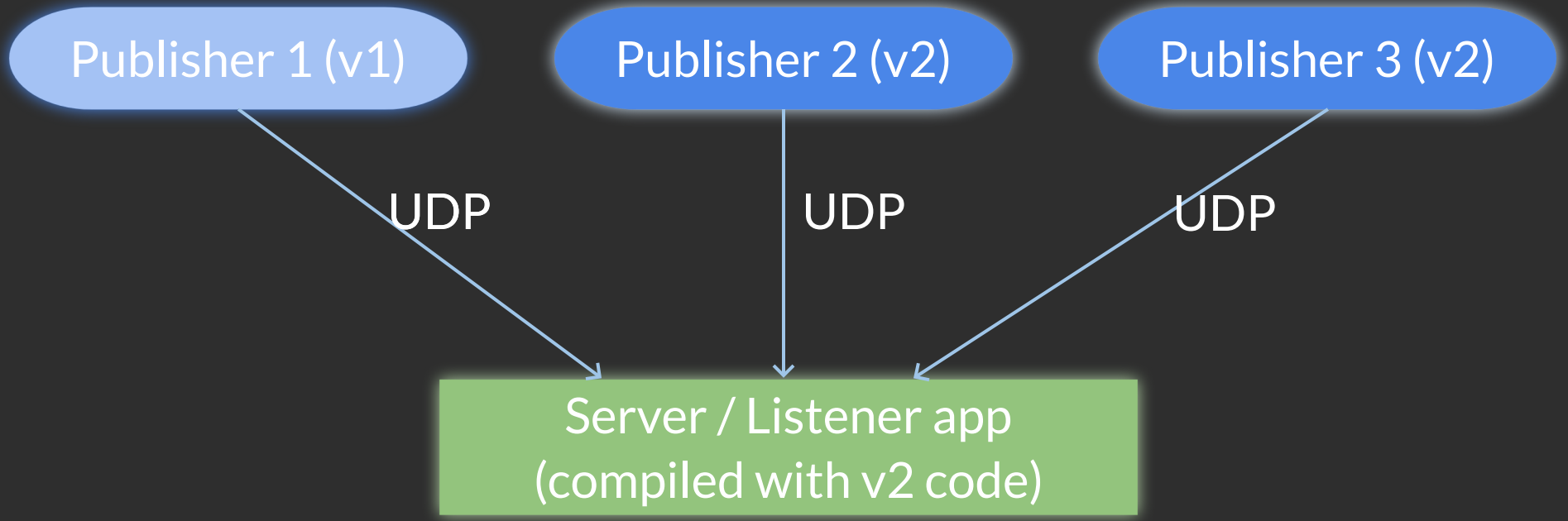
- Say you had a unique hash for a type's memory layout.
  - What would you do?



- We'd like to minimize the space we're using to describe the data's schema in a given message.
  - Ideally good to keep the comparison fast as well.
  - Example message sent between two modules communicating.

Setting the stage - Why is a type's hash useful?

- Say you had a unique hash for a type's memory layout.
  - What would you do?



Setting the stage - Why is a type's hash useful?

- Say you had a unique hash for a type's memory layout.
  - What would you do?
- Why not compare the "type" in text form?
  - Situations where speed is important
    - Cannot compare full type layouts each time for speed
    - Handshakes can help
  - Situations where size is important
    - Messages written to disk or sent over the network

Setting the stage - Why is a type's hash useful?

- Say you had a unique hash for a type's memory layout.
  - What would you do?

- Send this over UDP for handshake-less connections
  - Connections with handshakes can use full description of layout
    - That's hard enough already without protobuf etc.
  - Sending over a hash is reasonably size-efficient

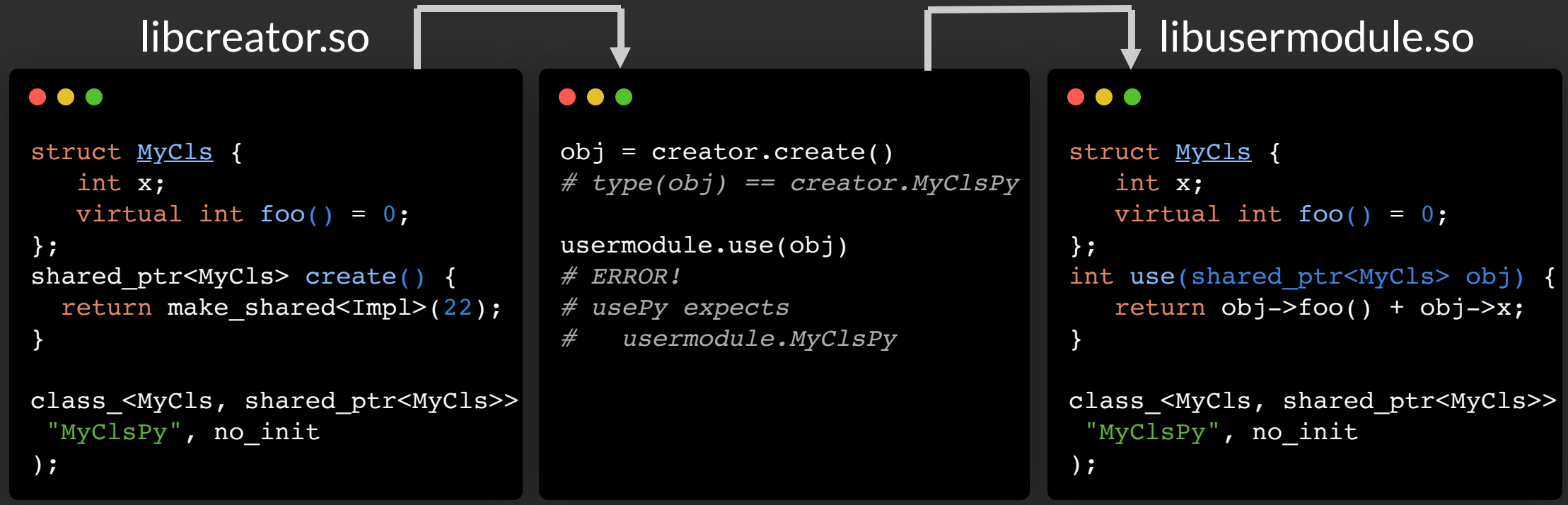
Setting the stage - Why is a type's hash useful?

- Say you had a unique hash for a type's memory layout.
  - What would you do?

- Use it to safely type-cast across module boundaries
  - Multiple python modules?
  - Different .so files talking to each other with non-trivial types at the ABI boundary.

Setting the stage - Why is a type's hash useful?

- Say you had a unique hash for a type's memory layout.
  - What would you do?



Issues with a simple version field

- Doesn't handle quirks that humans may miss (eg: attributes)
- Easy to forget to change version
  - Especially with transitive dependencies
- Can't do it generically in a library, user has to handle

```
1 struct BaseT {
2  int a;
3 } __attribute__((packed));
4
5 struct MyMessage : public BaseT {
6  int b;
7 };
```

Okay, so let's hash the type layout

Reflection!

Reflection to iterate a type's layout

- Is a decent test of the capabilities of reflection (P2996).
- Requires recursively computing the hash of types. Avoid cycles!
- Requires a compile time hashing function.
- Requires full visibility into the class' data layout
  - Sounds scary actually, private members!

Reflection to iterate a type's layout

Also, requires some level of configurability

```
1 struct ABIHashingConfig {
2     static constexpr int MINIMUM_SUPPORTED_VERSION = 0;  // To allow future rollover
3     static constexpr int MAXIMUM_SUPPORTED_VERSION = 0;  // To gracefully error out
4     uint8_t version : 4 = 0;
5     bool include_nsdm_names : 1 = true;
6     bool include_indirections : 1 = false;  // Only relevant in intra-process
7 } __attribute__((packed));
```

ABI hashing - The test case

```cpp
struct Order { int side = 1; size_t quantity = 0; };

template <typename T> struct MyList {
   T* start = nullptr;
   T* end = nullptr;
   bool valid : 1 = false;
};

struct OrderBook {
   MyList<Order> buy_orders;
   MyList<Order> sell_orders;
};
```

ABI hashing - The API

```
1 template <typename T, std::meta::abi::ABIHashingConfig config>
2 consteval size_t get_abi_hash();
3
4 // ...or...
5
6 consteval size_t get_abi_hash(std::meta::info R,
7                               std::meta::abi::ABIHashingConfig config);
```

ABI hashing - The core

```
consteval size_t _get_abi_hash_impl(
    std::meta::info R,
    meta::abi::ABIHashingConfig config = meta::abi::ABIHashingConfig{}) {

    // Recurse over any base classes
    size_t hash = 0;
    for (auto e : bases_of(R)) {
        hash = hash_combine(hash, get_abi_hash(e, config));
    };

    // Loop over non-static data members
    for (auto e : nonstatic_data_members_of(R)) {
        auto elem_type = meta::type_of(e);

        // Check for indirection types
        auto is_indirect_ref = (type_is_pointer(elem_type) ||
                                type_is_reference(elem_type);
        if (!config.include_indirections && is_indirect_ref) {
            // Maybe even warn or throw, since no use-case.
            continue;
```

Anything left?

ABI hashing - Size of the struct

```
1 // Despite all members being the size, attributes like `packed` may change
2 // the size of the struct. Not everyone would be concerned with padding at
3 // the end though. Can consider making this optional via a config param.
4
5 if (std::meta::size_of(R) > 0) {
6     hash = hash_combine(hash, std::meta::size_of(R) - parent_size_sum);
7 }
```

ABI hashing - Avoiding cycles

```cpp
consteval size_t get_abi_hash(
      std::meta::info R,
      ABIHashingConfig config = {},
      std::vector<std::meta::info> active_types = {}) {

    auto it = std::ranges::find_if(
        active_types, [R](const auto& elem) { return elem == R; }
    );

    if (it == active_types.end()) {
        active_types.push_back(R);
        return _get_abi_hash_impl(R, config, active_types);
    } else {
        // Cycle detected! Return the index since we must modify the hash still.
        return (it - active_types.begin());
    }
}
```

And just for fun, same code with type based code

ABI hashing - Avoiding cycles

```cpp
template <typename T, ABIHashingConfig config, typename... ActiveTypes>
consteval size_t get_abi_hash() {

    constexpr ssize_t type_index = mp11::mp_find<mp_list<ActiveTypes...>, T>();

    if constexpr(type_index != mp11::mp_size<mp_list<ActiveTypes...>>()) {
        return static_cast<size_t>(type_index);
    } else {
        return _get_abi_hash_impl<T, config, ActiveTypes...>();
    }
}
```

ABI hashing - So what do we have now

- Compatibility of data layout across processes, without protobuf
  - https://godbolt.org/z/hjxfYb8d9
- Solves 80% of the common set of requirements
  - Can a python binding library cast one type to the other safely?
  - Are these two networked binaries using the same data layout?
- Requires minimal-to-no work on the user's end
- Inflexible and hard to modify / handle unique situations

ABI hashing - Where do we go from here?

- Full ABI textual representation of the layout
- You could generate a “schema” file for your structs
  - Could be some JSON based schema
  - Could be a pre-existing schema like Apache Avro
  - This way we get a full ecosystem (with cross language support) for free.

And now to my personal favorite

A pythonic std::any

Boost.Python had a dream

```
object f(object x, object y) {
    if (y == "foo")
        x.slice(3,7) = "bar";
    else
        x.attr("items") += y(3, x);
    return x;
}

// Duck typing and a completely untyped type!
```

https://www.boost.org/doc/libs/1_66_0/libs/python/doc/html/tutorial/tutorial/object.html

std::any had a dream

```cpp
// any type
std::any a = 1;
std::cout << a.type().name() << ": " << std::any_cast<int>(a) << '\n';

a = 3.14;
std::cout << a.type().name() << ": " << std::any_cast<double>(a) << '\n';

a = true;
std::cout << a.type().name() << ": " << std::any_cast<bool>(a) << '\n';

// No duck typing sadly, but we got the equivalent of std::variant<everythi
// Can't do much though :)
```

https://en.cppreference.com/w/cpp/utility/any


virtual_any
(with duck typing)
std::any
Boost.Python

How would an ideal code look?

```
1  class MyDuck {
2  public:
3     int x;
4     MyDuck(int x) : x(x) {}
5     std::string do_quack() const { return "quack"; }
6  };
7
8  auto a = make_virtual_any<int>(7);
9  std::cout << "Printing " << a << std::endl;
10
11 a = make_virtual_any<MyDuck>(MyDuck(3));
12 std::cout << "My obj " << a << " has .x == " << a.attr("x")
13           << ", .do_quack() == " << a.call("do_quack") << std::endl;
```

We can do this!

Holding class : virtual_any

```
 1 class virtual_any_interface;
 2 class virtual_any {
 3     std::shared_ptr<virtual_any_interface> _impl;
 4     bool valid = false;
 5
 6    public:
 7     virtual_any() = default;
 8     template <typename T> virtual_any(T&& value);
 9     virtual_any(const virtual_any& other) = default;
10     virtual_any(virtual_any&& other) = default;
11     virtual_any& operator=(const virtual_any& other) = default;
12     virtual_any& operator=(virtual_any&& other) = default;
13
14     virtual_any attr(const std::string& name);
15
16     template <typename... ArgsT>
17     virtual_any call(const std::string& name, ArgsT&&... args);
18
19     friend std::ostream& operator<<(std::ostream& os, const virtual_any& self);
```

virtual_any_interface

```
1 class virtual_any_interface {
2    public:
3     virtual ~virtual_any_interface() = default;
4     virtual virtual_any attr(const std::string& name) = 0;
5     virtual virtual_any call(const std::string& name,
6                              const std::vector<virtual_any>& args) = 0;
7     virtual std::ostream& stream(std::ostream& os) const = 0;
8 };
```

virtual_any_interface

- virtual_any calls a virtual method on the held type
  - virtual_any_interface
- We implement virtual_any_interface for each type
  - Using reflection!
  - No code generation needed for this :)
- Handle some special function operators natively
  - operator<<, operator(), operator+

Implementation class : virtual_any_impl

```
 1 template <typename T>
 2 class virtual_any_impl : public virtual_any_interface {
 3     T _value;
 4
 5   public:
 6     virtual_any_impl(T&& value);
 7     virtual virtual_any attr(const std::string& name) override;
 8
 9     virtual virtual_any call(const std::string& name,
10                              const std::vector<virtual_any>& args) override;
11     virtual std::ostream& stream(std::ostream& os) const override;
12     T& ref();
13     const T& ref();
14 };
```

Add allocators : virtual_any_impl

```
template <typename T, typename Allocator = std::allocator<T>>
class virtual_any_impl : public virtual_any_interface {
    T* _value;
    Allocator _allocator;

  public:
    virtual_any_impl(T&& value);
    ~virtual_any_impl();
    virtual virtual_any attr(const std::string& name) override;

    virtual virtual_any call(const std::string& name,
                             const std::vector<virtual_any>& args) override;
    virtual std::ostream& stream(std::ostream& os) const override;
    T& ref();
    const T& ref();
};
```

virtual_any_impl: Get data members

```
template <typename T>
virtual_any virtual_any_impl<T>::attr(const std::string& name) {
    if constexpr(!meta::type_is_class (^T)) {
        throw std::runtime_error("<helpful error>");
    } else {
        // Actual logic
        virtual_any result;
        [:expand(meta::nonstatic_data_members_of (^T)):] >> [&]<auto e> {
            if constexpr(!meta::is_public(e) || !meta::has_identifier(e)) {
                return;
            } else {
                if (result.is_valid()) {
                    return;
                }
                auto elem_name = std::string(std::meta::identifier_of(e));
                if (name != elem_name) {
                    return;
                }
                result = make_virtual_any(_value.[:e:]);
```

virtual_any_impl: Implementing operators

```
1 template <typename T> ostream& virtual_any_impl<T>::stream(ostream& os) const {
2     if constexpr(requires { os << _value; }) {
3         os << _value;
4     } else {
5         os << "(non-streamable object of type " << meta::display_string_of(^T) << ")";
6     }
7     return os;
8 }
```

And there it is!

It works!

```
1  int main() {
2     auto a = make_virtual_any<int>(7);
3     std::cout << "Printing " << a << std::endl;
4
5     a = make_virtual_any<MyDuck>(MyDuck(3));
6     std::cout << "My obj " << a << " has .x == "
7               << a.attr("x") << std::endl;
8  }
9
10 $ ./virtual_any/main.out
11 Printing 7
12 My obj MyDuck(3) has .x == 3
```

Now on to member functions

Need complete type-erasure

```cpp
class MyDuck {
  public:
    std::string do_quack() const;
    int do_quack_n_m(int n, int m) const;
    virtual_any do_quack_n_virtual(virtual_any n) const;
};

int main() {
    auto a = make_virtual_any<MyDuck>(MyDuck(3));
    std::cout << ".do_quack()           == "
              << a.call("do_quack") << std::endl;
    std::cout << ".do_quack_n_m()         == "
              << a.call("do_quack_n_m", int{3}, int{4}) << std::endl;
    std::cout << ".do_quack_n_virtual() == "
              << a.call("do_quack_n_virtual", make_virtual_any<int>(4)) << std::endl;
}
```

Get member functions with arity 0

```
 1 struct Callable {
 2     std::function<virtual_any(const std::vector<virtual_any>&)> func;
 3     size_t arity;
 4 };
 5
 6 Callable virtual_any_impl<T>::get_member_function(const std::string& name) {
 7     ...
 8     Callable result;
 9     bool found = false;
10
11     [:expand(meta::members_of (^T)):] >> [&]<auto e> {
12         if constexpr(!meta::is_public(e) || !meta::has_identifier(e)) {
13             return;
14         } else {
15             if (found) {
16                 return;
17             }
18             auto elem_name = std::string(std::meta::identifier_of(e));
19             if (name != elem_name) {
20                 return;
```

What about higher arity?

Member functions that take arguments

- How do we find function signature?
  - P3096 helps :) "Function Parameter Reflection"
  - Can also do metaprogramming (Boost.CallableTraits etc)

- Runtime virtual interface means cannot use template types

```
1 class virtual_any {
2     template <typename... ArgsT>
3     virtual_any call(const std::string& name, ArgsT&&... args);
4
5 };
6
7 class virtual_any_interface {
8     virtual virtual_any call(const std::string& name, const std::vector<virtual_any>& args) = 0;
9 };
```

Member functions that take arguments

- How do we find function signature?
  - P3096 helps :) "Function Parameter Reflection in Reflection"
  - Can also do metaprogramming (Boost.CallableTraits etc)

- Runtime virtual interface means cannot use template types

- Type erasure means cannot go from virtual_any to T without RTTI
  - std::any
  - dynamic_cast?
  - Some new ideas?

Get member functions arity > 0

```
namespace ct = boost::callable_traits;

Callable get_member_function(const std::string& name) {
    ...
    [:expand(meta::members_of (^T)):] >> [&]<auto e> {
        ... same as before ...

        using Fn = decltype(&[:e:]);
        using ArgsT = ct::args_t<Fn>;
        auto arity = std::tuple_size<ArgsT>::value;
        auto l = [this](const std::vector<virtual_any>& any_args) -> virtual_any {
            auto args_final = get_args_from_any_vector<ArgsT>::get(any_args);
            return make_virtual_any(
                utils::apply_invoke(&[:e:], _value, args_final));
        };
        result = {l, arity - 1};
        found = true;
    };
    ...
```

get_args_from_any_vector

```cpp
template <typename FirstArg, typename... ArgsT>
struct get_args_from_any_vector<std::tuple<FirstArg, ArgsT...>> {
    static std::tuple<ArgsT...> get(const std::vector<virtual_any>& any_args) {
        if (any_args.size() != sizeof...(ArgsT)) {
            throw std::runtime_error("<nice error>");
        }
        std::array<virtual_any, sizeof...(ArgsT)> arr;
        std::copy(any_args.begin(), any_args.end(), arr.begin());
        return utils::any_array_to_tuple<ArgsT...>(arr);
    }
};

template <typename... Ts>
std::tuple<Ts...> any_array_to_tuple(const std::array<virtual_any, sizeof...(Ts)>& any_
    return std::apply(
        [](const auto&... args) {
            return std::tuple<Ts...>(virtual_any_cast<Ts>(args)...);
        },
        any_array);
}
```

get_underlying_element_ref

- Sounds like RTTI :/
- Let's try anyways

```cpp
template <typename T>
T& virtual_any::get_underlying_element_ref() {
    if constexpr(std::is_same_v<T, virtual_any>) {
        return *this;
    }
    auto* ptr = _impl.get();
    if (auto* derived_ptr = dynamic_cast<virtual_any_impl<T>*>(ptr);
            derived_ptr != nullptr) {
        return derived_ptr->ref();
    }
    std::stringstream ss;
    ss << "Element " << *this << " not of type " << std::meta::display_string_of (^T);
    throw std::runtime_error(ss.str());
}
```

It works at least!

```cpp
class MyDuck {
  public:
    std::string do_quack() const;
    int do_quack_n_m(int n, int m) const;
    virtual_any do_quack_n_virtual(virtual_any n) const;
};

int main() {  // Prints shortened for readability
    auto a = make_virtual_any<MyDuck>(MyDuck(3));
    std::cout << a.call("do_quack") << std::endl;
    std::cout << a.call("do_quack_n_m", 3, 4) << std::endl;
    std::cout << a.call("do_quack_n_virtual", make_virtual_any<int>(4)) << std::endl;
}

// output
.do_quack()           == quack quack quack
.do_quack_n_m()       == 12
.do_quack_n_virtual() == 4
```

Even for reference types

```cpp
class MyDuck {
  public:
    std::string do_quack() const;
    int do_quack_n_m(int& n, int m) const;
    virtual_any do_quack_n_virtual(virtual_any n) const;
};

int main() {  // Prints shortened for readability
    auto a = make_virtual_any<MyDuck>(MyDuck(3));
    int n = 3;
    std::cout << a.call("do_quack_n_m", n, 4) << std::endl;
}

// output
.do_quack_n_m()       == 12
```

Any other non-RTTI ideas?

- Same problem as std::any unfortunately
  - Need RTTI to compare two types at runtime

- What about ABI hashing?
  - Instead of using dynamic_cast to validate!
  - Use type's hash to validate!
  - Hash collisions might be sad.

Any other non-RTTI ideas?

- Per-type statics
  - Could work under constrained conditions
  - Relies on linker merging same types' GetTypeId
  - Wouldn't have false positives at least :)

```
#pragma once

// In header, to avoid ODR violations
template <typename T>
const void* GetTypeId() {
    static const char unique;
    return &unique;
}
```

Discussion on virtual_any

- Name is solid I know ;)
- Slow to run
  - Not the worst thing for writing script like code from time to time
- Slow to compile
  - Hopefully more reflection features like P3096 help with this
- Function calling API is a bit limited
- Works as of P2996R5! https://godbolt.org/z/xdzes5EYf

What do we get out of this?

- A fun new experiment with Reflection?
- Offers a clean duck-typed syntax, setup at compile time
  - Could start implementing better inter-module DSLs in C++?
- Potentially allows a more dynamic interface between modules
  - Similar to motivation for std::any, but even more relaxed

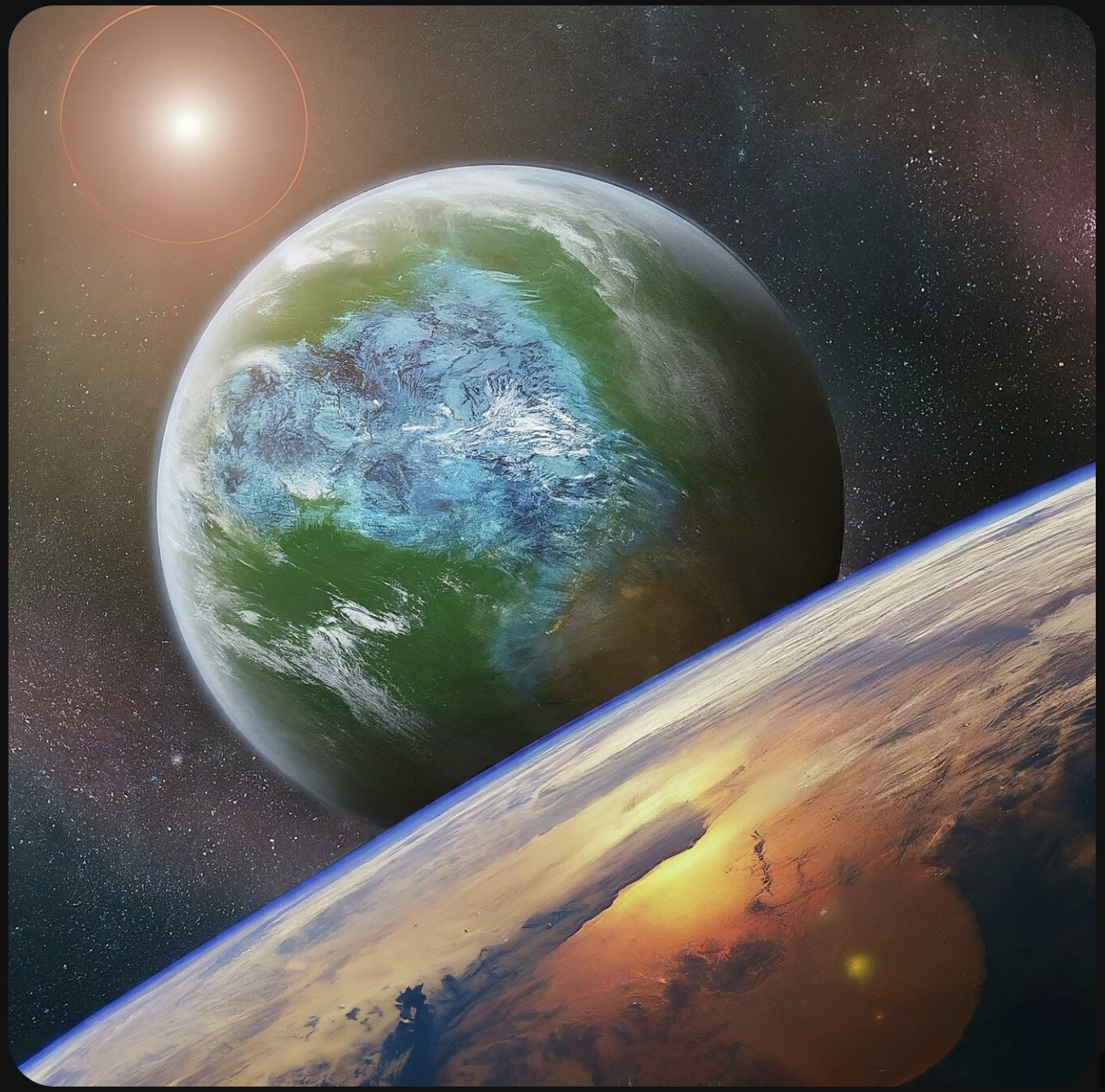
Object lifetimes

- We skipped over object lifetimes
  - No garbage collection, but we come close
  - Still room to work on the object model for this!
- We could have also borrowed Python's machinery for this
  - Could re-implement a garbage collector within this subsystem
- Hopefully we aren't using this in performance sensitive code :)

That concludes `virtual_any`

Alternatives

How did we live without this?



Human reflection

- Well, we've been writing boilerplate
  - Python bindings
  - Serialization / deserialization / CLI parsers
- Our code, and our libraries are worse off
- Compile-time reflection makes for cleaner APIs
  - We *can* do "new" syntactic things to avoid redundancy in code

Sourcegen!

- Protobuf
  - (in)famous library loved and hated by everyone at the same time
  - Lets you do some primitive “reflection-like” stuff
    - Enum <=> String
- Apache Avro
  - Sourcegen with a “handshake” / “header”
  - Better “reflection-like” API than protobuf

Sourcegen, ft. Protobuf

```
enum VehicleType {
 UNDEFINED = 0;
 CAR = 1;
 TRUCK = 2;
 MOTORCYCLE = 3;
}

message Vehicle {
 string model = 1;
 VehicleType type = 2;
 repeated string features = 3;
}
```

```
enum VehicleType { UNDEFINED = 0, CAR, TRUCK, MOTORCYCLE };
bool VehicleType_IsValid(int value);

const ::google::protobuf::EnumDescriptor* VehicleType_descr

inline const ::std::string& VehicleType_Name(VehicleType va

inline bool VehicleType_Parse(const ::std::string& name, Ve

class Vehicle {
public:
   const std::string& model() const;
   void set_model(const std::string& value); ...
   const ::google::protobuf::RepeatedPtrField<std::string>&

private:
   std::string model_;
   VehicleType type_;
   ::google::protobuf::RepeatedPtrField<std::string> featur
};
```

Sourcegen, ft. Avro

```
{
 "namespace": "example",
 "type": "record",
 "name": "Vehicle",
 "fields": [
   {"name":"model","type":"stri
   {"name": "type",
    "type": {
      "type": "enum",
      "name": "VehicleType",
      "symbols": [...]
   }},
   {"name": "features",
    "type": {
      "type":"array","items":"s
   }}
 ]
}
```

```
struct Vehicle {
    std::string model;
    VehicleType type;
    std::vector<std::string> features;

    void serialize(std::ostream& out) const {
        auto schema = avro::compileJsonSchemaFromString(sche
        avro::EncoderPtr encoder = avro::binaryEncoder();
        encoder->init(out);
        avro::encode(*encoder, *this);
    }

    void deserialize(std::istream& in) {
        auto schema = avro::compileJsonSchemaFromString(sche
        avro::DecoderPtr decoder = avro::binaryDecoder();
        decoder->init(in);
        avro::decode(*decoder, *this);
    }
};
```

Sourcegen, ft. LLMs!

```
struct ConfigOptions {
    bool enable;
    int count;
    std::string mode;
};

int main() {
    boost::program_options po;
    po.add_options()
        ("enable", boost::program_options::value<bool>(), "enable")
        ("count", boost::program_options::value<int>(), "count")
        ("mode", boost::program_options::value<std::string>(), "mode");
}
```

Among others...

- Classdesc
  - Another form of sourcegen
- https://github.com/boost-ext/reflect
  - Template metaprogramming to its limits

Summary

- Reflection is really good for the C++ ecosystem!
- It is not (too...) complicated to understand in its current form
  - Definitely easier than some parts of TMP
- Let's get hyped up!
  - Crazy libraries coming soon to a codebase near you :)

[:fin:]

Saksham Sharma
linkedin.com/saksham-sharma
x.com/sakshamarma