Reflection is Good for (Code) Health

Saksham Sharma
Yours truly

- Quantitative research technology at Tower Research Capital
  - High frequency trading firm based out of NYC

- Develop low latency trading systems (C++)
  - Nanoseconds and microseconds

- Develop high throughput research systems (C++ and Python)
  - O(terabytes) data

- Program analysis and functional programming in a past life
- Love performance, software abstractions, and clean APIs
The boring part

This talk’s contents are mine and mine alone

- Not my employer’s :)
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?

Ability to write code in a language such that:

- Access information about other “code” in a programmatic form and operate on it

```cpp
class MyClass {
    int a;
    int b;
};
for (auto member_info : gimme_class_members<MyClass>()) {
    std::cout << "member - " << member_info.name() << std::endl;
}
```
Reflection?

How is this different from metaprogramming / templates?
Reflection?

- Not much :)

- Operate on constexpr 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
Runtime Reflection

Ability to access information about other “code” at runtime

- Could be done in interpreted or compiled languages
  - But generally seen in interpreted languages

```python
class MyClass:
    def __init__(self):
        self.x = 1
        self.y = 2

for member_name, member_value in MyClass().__dict__.items():
    print(f"{member_name}: {member_value}")
```
Compile Time Reflection / Static reflection

Ability to access information about other “code”:

- At compile time

```cpp
class MyClass {
    int a;
    int b;
};
for (auto member_info : gimme_class_members<MyClass>()) {
    if (is_ptr(member_info)) { ... }
}
```
Examples from other languages

Venturing into some alien worlds
My favorite example: Python

At runtime your code can:

- Modify other code to do entirely different things when a method is called.
- Say you run some code and it can change what a method on your object actually does.
- Change what it means to access a field on an object.
- Add new methods or attributes to any object.
Fun with reflection - 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
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 0x7f1a9e6>
```
Golang

- Golang is a compiled but duck-typed language
  - Well, structurally typed, but close enough
- Relies heavily on interfaces
- Runtime reflection similar to python.
  - No special compile time constructs
  - Uses the `reflect` package’s methods to get “reflection values”.

---

Golang is a compiled but duck-typed language. While it's structurally typed, it's close enough to be considered as such. Golang heavily relies on interfaces. It also has runtime reflection similar to Python, but it doesn't have special compile-time constructs. Instead, it uses the `reflect` package’s methods to get “reflection values”.
Duck typing?
type Vehicle interface {
    Start() string
}

type Car struct {
    Make string
}

func (this Car) Start() { // Car::Start(Car* this)
    return "Brrr"
}

type Truck struct {
    WheelCount int
}

func (this Truck) Start() {
    return "Brhhhhhhhhhh"
}

func (v Vehicle) SelfDrive() {
    fmt.Println(v.Start())
}

func main() {
    vehicles := []Vehicle{
        Car{"Toyota"},
        Truck{4},
    }
    for _, v := range vehicles {
        SelfDrive(v)
    }
}
Fun with reflection - Golang

type T struct {
    A int
    B string
}
t := T{23, "skidoo"}
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, ofTypeT.Field(i).Name, f.Type(), f.Interface())
}
// 0: A int = 23
// 1: B string = skidoo

*https://go.dev/blog/laws-of-reflection*
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
Class cls = Class.forName("method1");  // Surprising, lookup types with string!

Method methlist[] = cls.getDeclaredMethods();

for (int i = 0; i < methlist.length; i++) {
    Method m = methlist[i];
    System.out.println("name = " + m.getName());
    System.out.println("decl class = " + m.getDeclaringClass());
}

Class pvec[] = m.getParameterTypes();

for (int j = 0; j < pvec.length; j++)
    System.out.println("param #" + j + " " + pvec[j]);

*https://www.oracle.com/technical-resources/articles/java/javareflection.html*
Scary?
Zero cost abstractions

- Prior examples had “reflection objects” at runtime.
- Despite the temptation, we must do compile time
  - No one likes the cost of RTTI
- We can lean on constexpr algorithms and metaprogramming!
  - Compile time programming is turing complete after all :)
A NEW HOPE!

P2996
We have a syntax and an implementation!

- P2996 is a promising paper gaining traction.
  - Wyatt Childers, Peter Dimov, Dan Katz, Barry Revzin, Andrew Sutton, Faisal Vali, Daveed Vandevoorde
- 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/cGK4Eo6K1
We have a syntax and an implementation!

- Good consensus on syntax and some semantics.
- Value based reflection is an interesting choice
  - We’ll talk about it soon :)
- Prior work: Reflection TS
  - David Sankel – Type based reflection
We have a **reflection operator**

unary operator \(^\)
“Lifts” into reflection land
i.e. produces a reflection value

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

```cpp
auto magic_reflection_method(std::meta::info obj) {
  .. do something with “meta” info ..
}

auto result = magic_reflection_method(^MyType);
```
We have a `std::meta::info`

A formless type that describes all meta-information about a type / member / method / etc.

```cpp
auto magic_reflection_method(std::meta::info obj) {
  .. do something with “meta” info ..
}

auto result = magic_reflection_method(^MyType);
```
We have a splice operator

\[
\text{: r :} \Rightarrow \\
\text{\hspace{1em} Takes a std::meta::info} \\
\text{\hspace{1em} Constant expression} \\
\text{\hspace{1em} Splices it back into your} \\
\text{\hspace{1em} regular code}
\]
We have a `std::meta::define_class`.

We can create classes in thin air! (almost)

Need to declare it first, need to name all the members and their types.

```cpp
struct S;
static_assert(is_type(define_class(^S, {
    data_member_spec(^int, {
      .name="i", .align=64
    }),
    data_member_spec(^int, {
      .name="j", .align=64
    })
  }));
```
We have a `std::meta::define_class`

Equivalent of....

```c++
struct S;
struct S {
    alignas(64) int i;
    alignas(64) int j;
};
```
We have a lot of constexpr methods to go with it

namespace std::meta {

    // [meta.reflection.names], reflection names and locations
    constexpr string_view name_of(info r);
    constexpr string_view qualified_name_of(info r);

    ...

    constexpr bool is_function(info r);
    constexpr bool is_variable(info r);
    constexpr bool is_type(info r);

    // [meta.reflection.member.queries], reflection member queries
    template<class... Fs>
    constexpr vector<info> members_of(info type, Fs... filters);

    ...
}

Elevator pitch

We really do need this in C++!
enum class MyEnum { VALUE_1, VALUE_2, MAX_VALUES };

template <typename EnumT> constexpr std::string enum_to_string(EnumT enum_value) {
    template for (constexpr auto e : std::meta::enumerators_of(EnumT)) {
        if (enum_value == [:e:]) return std::string(std::meta::name_of(e));
    }
    return "<unnamed>";
}

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)
    );
}
```
**Easy cmdline parsing**

```cpp
template <typename Opts> auto parse_options(ArgT args) -> Opts {
    Opts opts;
    template for (constexpr auto dm : nonstatic_data_members_of (^

    auto it = std::ranges::find_if(args, ... match string ...);
    auto iss = std::ispanstream(it[1]);
    if (iss >> opts.[:dm:]; !iss) {
        std::print(stderr, "Failed to parse option {}\n", *it);
        std::exit(EXIT_FAILURE);
    }

    return opts;
}
```

*copied from P2996*
**Better cmdline parsing**

```c
struct Args : Clap {
    Option<std::string, {.use_short=true, .use_long=true}> name;
    Option<int, {.use_short=true, .use_long=true}> count = 1;
};

int main(int argc, char** argv) {
    auto opts = Args{}.parse(argc, argv);
    for (int i = 0; i < opts.count; ++i) { // opts.count has type int
        std::print("Hello {}", opts.name); // opts.name has type std::string
    }
}
```

*copied from P2996*
consteval auto get_layout() {
    constexpr auto members = nonstatic_data_members_of (^S);
    std::array<my_descriptor, members.size()> layout;
    for (int i = 0; i < members.size(); ++i) {
        layout[i] = {.offset = offset_of(members[i]), .size = size_of(members[i])};
    }
    return layout;
}
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*
Why this 'formless' `std::meta::info'?
Type vs Value based reflection

- Type based reflection is more intuitive
  - `std::meta::method / std::meta::variable / std::meta::class`
  - Can imagine writing template specializations etc
  - Maybe ranges and algorithms make life better with value based?
Type vs Value based reflection

- Type based reflection is more intuitive
  - `std::meta::method / std::meta::variable / std::meta::class`
  - Can imagine writing template specializations etc
  - Maybe ranges and algorithms make life better with value based?

- Slower to compile
  - Lot more types to handle for the compiler
Type vs Value based reflection - Compile times

constexpr int f() {
    int i = 0;
    for (int k = 0; k < 10000; ++k)
        i += k;
    return i / 10000;
}

template <int N> struct S {
    static constexpr int sm = S<N - 1>::sm + f();
};
template <> struct S<0> {
    static constexpr int sm = 0;
};
constexpr int r = S<200>::sm;

struct Int { int v; };
constexpr int f() {
    Int i = {0};
    for (Int k = {0}; k.v < 10000; ++k.v)
        i.v += k.v;
    return i.v / 10000;
}

template <int N> struct S {
    static constexpr int sm = S<N - 1>::sm + f();
};
template <> struct S<0> {
    static constexpr int sm = 0;
};
constexpr int r = S<200>::sm;
Type vs Value based reflection

- Type based reflection is more intuitive
  - `std::meta::method / std::meta::variable / std::meta::class`
  - Can imagine writing template specializations etc
  - Maybe ranges and algorithms make life better with value based?

- Slower to compile
  - Lot more types to handle for the compiler

- Makes language **rigid** by preventing future breakages
  - Major issue, downstream code can start depending on which type a specific reflection would be.
  - Sometimes might not be clear which type to classify a reflection into
Type vs Value based reflection

- We want users to write “duck typed” reflection code
  - They check if that reflection value has the feature they want from it
  - If so, you can use it in your desired context.
- Flexible because every usage would ideally be gated by a feature check
  - Easier to add feature checks compared to renaming meta classes
- Feature checks have better cross-interaction than inheritance
What do we know so far?

<table>
<thead>
<tr>
<th>Language</th>
<th>Runtime or compile-time</th>
<th>Type or value based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>Runtime</td>
<td>?</td>
</tr>
<tr>
<td>Golang</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Java</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>C++26</td>
<td>Compile-time</td>
<td>Value (std::meta::info)</td>
</tr>
</tbody>
</table>
Let’s tabularize this

<table>
<thead>
<tr>
<th>Language</th>
<th>Runtime or compile-time</th>
<th>Type or value based (loosely)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>Runtime</td>
<td>Value (with a sprinkle of Type)</td>
</tr>
<tr>
<td>Golang</td>
<td>Runtime</td>
<td>Value (with a sprinkle of Type)</td>
</tr>
<tr>
<td>Java</td>
<td>Runtime</td>
<td>Type</td>
</tr>
<tr>
<td>C++26</td>
<td>Compile Time</td>
<td>Value (std::meta::info)</td>
</tr>
</tbody>
</table>
How do we use this new superpower?

No one likes chaos.
Libraries!

- Makes it easier to write general-purpose / boilerplate-reducing libraries
- Libraries would require fewer redundant inputs from the user
  - A CLI parsing library won’t need you to enumerate all objects in your struct each time.
  - Maybe it could take a spec to parse and return a struct to you storing that spec?
Libraries!

Python bindings, ABI hashing, A better std::any
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

```cpp
struct Item {
    int id;
    PyObject* getPyValue() const;
};
struct Row {
    const auto& items() const { return items_; }  
    auto nanotime() const { return nanotime_; }
};
class RowReader : public BinaryListener {
public:
    RowReader(const std::string& filename) : reader_(filename);
    std::string getIdName(int id) const;
    const auto& getRows();
};
```
A fan favorite, and one of the most common wishlist items

- We don’t want to write this
- Most of the code is just repetition of what we already know
- Some of the things are non-obvious
  - Whether a function’s return type should be wrapped as a reference or a value copy
  - How to name a type in Python

```cpp
BOOST_PYTHON_MODULE(binary_reader_bpy) {
    class_<Item>("Item", no_init)
        .def_readonly("id", &Item::id)
        .def("value", &Item::getPyValue);

    class_<Row, std::shared_ptr<Row>>("Row", no_init)
        .def("nanotime", &Row::nanotime)
        .def("items", &Row::items,
             return_internal_reference<>());

    class_<RowReader, boost::noncopyable>(
        "RowReader", init<std::string>(arg("filename")))
        .def("getRows", &RowReader::getRows)
        .def("getIdName", &RowReader::getIdName);
}
```
template <typename T> object make_python_type() {
    std::string cls_name{meta::name_of (^T)};
    auto type_obj = class_<T>(cls_name.c_str(), init<int, int>();
    [:expand(meta::members_of (^T)):] >> [&]<auto e> {
        if constexpr (!meta::is_public(e)) { return; }
        std::string name{meta::name_of(e)};
        if constexpr (meta::is_nonstatic_data_member(e))
            type_obj.def_readwrite(name.c_str(), &[:e:]);
        if constexpr (meta::is_function(e) && !meta::is_constructor(e) && !meta::is_destructor(e)) {
            if constexpr (!std::is_reference_v<
                typename return_type<
                    decltype(&[:e:])::type>
            )
                type_obj.def(name.c_str(), &[:e:]);
        }
    };
    return type_obj;
}
Kind of...
template <typename T> object make_python_type() {
    std::string cls_name{meta::name_of (^T)};
    auto type_obj = class_<T>(cls_name.c_str(), init<int, int>())[::-expand(meta::members_of (^T)):] >> [&]<auto e> {
        if constexpr(!meta::is_public(e)) { return; }
        std::string name{meta::name_of(e)};
        if constexpr(meta::is_nonstatic_data_member(e))
            type_obj.def_readwrite(name.c_str(), &[:e:]);
        if constexpr(meta::is_function(e) && !meta::is_constructor(e) && !meta::is_destructor(e)) {
            if constexpr(!std::is_reference_v<typename return_type<decltype(&[:e:])-1::type>>) {
                type_obj.def(name.c_str(), &[:e:]);
            }
        }
    };
    return type_obj;
}
Loop over the members and methods

template <typename T> object make_python_type() {
    std::string cls_name{meta::name_of (^T)};
    auto type_obj = class_<T>(cls_name.c_str(), init<int, int>();

    template for (auto e : meta::members_of (^T)) {
        if constexpr(!meta::is_public(e)) { return; }
        std::string name{meta::name_of(e)};
        if constexpr(meta::is_nonstatic_data_member(e))
            type_obj.def_readwrite(name.c_str(), &[:e:]);
        if constexpr(meta::is_function(e) && !meta::is_constructor(e) && !meta::is_destructor(e)) {
            if constexpr(!std::is_reference_v<typename return_type<decltype(&[:e:])>::type>)
                type_obj.def(name.c_str(), &[:e:]);
        }
    }

    return type_obj;
}
Loop over the members and methods

template <typename T> object make_python_type() {
    std::string cls_name{meta::name_of (^T)};
    auto type_obj = class_<T>(cls_name.c_str(), init<int, int>())
        [:expand(meta::members_of (^T)):] >> [&]<auto e> {
            if constexpr(!meta::is_public(e)) { return; }
            std::string name{meta::name_of(e)};
            if constexpr(meta::is_nonstatic_data_member(e))
                type_obj.def_readwrite(name.c_str(), &[:e:]);
            if constexpr(meta::is_function(e) && !meta::is_constructor(e) && !meta::isDestructor(e)) {
                if constexpr(!std::is_reference_v<typename return_type<decltype(&[:e:]):type>>)
                    type_obj.def(name.c_str(), &[:e:]);
            }
        };
    return type_obj;
}
template <typename T> object make_python_type() {
    std::string cls_name{meta::name_of (^T)};
    auto type_obj = class_<T>(cls_name.c_str(), init<int, int>())
        [:expand(meta::members_of (^T)):] >> [&]<auto e> {
            if constexpr(meta::is_public(e)) {
                std::string name{meta::name_of(e)};
                if constexpr(meta::is_nonstatic_data_member(e)) {
                    type_obj.def_readwrite(name.c_str(), &[:e:]);
                } else if constexpr(meta::is_function(e) && !meta::is_constructor(e) && !meta::is_destructor(e)) {
                    if constexpr(!std::is_reference_v<typename return_type<decltype(&[:e:])>::type>) {
                        type_obj.def(name.c_str(), &[:e:]);
                    }
                } else {
                    if constexpr(!std::is_reference_v<typename return_type<decltype(&[:e:])>::type>) {
                        type_obj.def(name.c_str(), &[:e:]);
                    }
                }
            } else {
                return;
            }
        };

    return type_obj;
}
template <typename T> object make_python_type() {
    std::string cls_name{meta::name_of (^T)};
    auto type_obj = class_<T>(cls_name.c_str(), init<int, int>();
    [:expand(meta::members_of (^T)):] >> [&]<auto e> {
        if constexpr(!meta::is_public(e)) { return; }
        std::string name{meta::name_of(e)};
        if constexpr(meta::is_function(e) && !meta::is_constructor(e) && !meta::is_destructor(e)) {
            if constexpr(!std::is_reference_v<typename return_type<decltype(&[:e:])>::type>)
            type_obj.def(name.c_str(), &[:e:]);
        }
    };
    return type_obj;
}

Inform boost::python how to access the members??
template <typename T> object make_python_type() {
    std::string cls_name{meta::name_of (^T)};
    auto type_obj = class_<T>(cls_name.c_str(), init<int, int>())
        [:expand(meta::members_of (^T)):] >> [&]<auto e> {
            if constexpr(!meta::is_public(e)) { return; }
            std::string name{meta::name_of(e)};
            if constexpr(meta::is_function(e) && !meta::is_constructor(e) && !meta::is_destructor(e)) {
                if constexpr(!std::is_reference_v<typename return_type<decltype(&[:e:])>::type>)
                    type_obj.def(name.c_str(), &[:e:]);
            }
        }
    return type_obj;
}

Inform boost::python how to access the members
template <typename T> object make_python_type() {
    std::string cls_name{meta::name_of (^T)};
    auto type_obj = class_<T>(cls_name.c_str(), init<int, int>());
    [:expand(meta::members_of (^T)):] >> [&]<auto e> {
        if constexpr(!meta::is_public(e)) { return; }
        std::string name{meta::name_of(e)};
        if constexpr(meta::is_nonstatic_data_member(e))
            type_obj.def_readwrite(name.c_str(), &[:e:]);
        if constexpr(meta::is_function(e) && !meta::is_constructor(e) && !meta::is_destructor(e)) {
            if constexpr(!std::is_reference_v<
typename return_type<decltype(&[:e:])>::type>)
                type_obj.def(name.c_str(), &[:e:]);
        }
    };
    return type_obj;
}

Inform boost::python about methods
template <typename T> object make_python_type() {
    std::string cls_name{meta::name_of (^T)};
    auto type_obj = class_<T>(cls_name.c_str(), init<int, int>())
        [:expand(meta::members_of (^T)):] >> [&]<auto e> {
            if constexpr (!meta::is_public(e)) { return; }
            std::string name{meta::name_of(e)};
            if constexpr (meta::is_nonstatic_data_member(e))
                type_obj.def_readwrite(name.c_str(), &[:e:]);
            if constexpr (meta::is_function(e) && !meta::is_constructor(e) && !meta::isDestructor(e)) {
                if constexpr (!std::is_reference_v<typename return_type<decltype(&[:e:]):type>})
                    type_obj.def(name.c_str(), &[:e:]);
            }
        };
    return type_obj;
}

It works :)
Python bindings

- 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 methods in case of overloads)
- Docstrings
CUSTOMIZING THE DEFAULT BEHAVIOR

- Defining template specializations!
- Not ideal, but has benefits
  - Can annotate even if you don’t have control on source code

```cpp
constexpr auto customizations = {
  {^Row::items,
   return_value_policy::reference_internal},
  ...
};
```

*mentioned in P2911R0
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

```cpp
class Row {
public:
    const auto& items() { ... }
};
```

*mentioned in P2911R0*
USER DEFINED ATTRIBUTES

• One of my favorite features from Golang
• Annotating at the point of definition
  ○ Has its place, as opposed to annotating at the time of use

```go
type User struct {
    Name string `json:"name" required:"true`
}

user := User{"John"}
field, ok := reflect.TypeOf(user).Elem().FieldByName("Name")
fmt.Println(field.Tag, field.Tag.Get("required"))
// json:"name" required:"true" true
```

*https://www.makeuseof.com/reflection-in-go/*
Summary

● Default behavior can be done easily
● Customizations are trickier / not DRY
● User defined attributes might help!
  ○ In serialization / deserialization / CLI parsing too!
ABI HASHING

Why and how
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?
- Example message sent between two modules communicating.
  
  ```
  SchemaType  Rest of your data
  ```

- 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.
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 a full description of the 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?

```cpp
struct MyCls {
    int x;
    virtual int foo() = 0;
};
shared_ptr<MyCls> create() {
    return make_shared<Impl>(22);
}

class_<MyCls, shared_ptr<MyCls>>(
    "MyClsPy", no_init
);
```

```cpp
obj = creator.create()
# type(obj) == creator.MyClsPy
usermodule.use(obj)
# ERROR!
# usePy expects
#   usermodule.MyClsPy
```

```cpp
struct MyCls {
    int x;
    virtual int foo() = 0;
};
int use(shared_ptr<MyCls> obj) {
    return obj->foo() + obj->x;
}

class_<MyCls, shared_ptr<MyCls>>(
    "MyClsPy", no_init
);
```
**Issues with a simple version field**

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

```c
struct BaseT {
    int a;
} __attribute__((packed));

struct MyMessage : public BaseT {
    int b;
};
```
Okay how do we hash a type’s layout?

Reflection!!
Reflection to iterate a type’s layout

- Is a decent test of the capabilities of reflection as proposed in P2996.
- Requires recursively computing the hash of types of member variables of a class. Avoid cycles!
- Requires a constexpr hashing function.
- Requires full visibility into the class’ internals
  - Sounds scary actually, we can now write code that can know about private members of a class
Reflection to iterate a type’s layout

- Also, requires some level of configurability

```c
struct ABIHashingConfig {
    static constexpr int MINIMUM_SUPPORTED_VERSION = 0; // To allow future rollover
    static constexpr int MAXIMUM_SUPPORTED_VERSION = 0; // To gracefully error out
    uint8_t version : 4 = 0;
    bool include_nsdm_names : 1 = true;
    bool include_indirections : 1 = false; // Only relevant in intra-process
} __attribute__((packed));
```
ABI hashing - The test case

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

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

template <typename T> struct MyList2 : public MyList<T> {};

struct OrderBook {
    MyList2<Order> buy_orders;
    MyList2<Order> sell_orders;
};
```
template <typename T, std::meta::abi::ABIHashingConfig config>
consteval size_t get_abi_hash();

// ...or...

consteval size_t get_abi_hash(std::meta::info R,
                           std::meta::abi::ABIHashingConfig config);
consteval size_t _get_abi_hash_impl(
    std::meta::info R,
    meta::abi::ABIHashingConfig config = meta::abi::ABIHashingConfig{} ) {
    size_t hash = 0;
    for (auto e : bases_of(R)) {
        hash = hash_combine(hash, get_abi_hash(meta::type_of(e), config));
    }
}
for (auto e : nonstatic_data_members_of(R)) {
    auto elem_type = meta::type_of(e);
    auto is_indirect_ref = (meta::test_type(^std::is_pointer_v, elem_type) ||
                              meta::test_type(^std::is_reference_v, elem_type));
    if (!config.include_indirections && is_indirect_ref) {
        // Maybe even warn or throw, since no use-case.
        continue;
    }
}
for (auto e : nonstatic_data_members_of(R)) {
    if (config.include_nsdm_names) {
        auto name = std::string(meta::name_of(e));
        hash = hash_combine(hash, HASH_STR(name.c_str()));
    }
    if (meta::is_bit_field(e)) {
        hash = hash_combine(hash, std::meta::offset_of(e),
                             std::meta::bit_offset_of(e), std::meta::bit_size_of(e));
    } else {
        hash = hash_combine(hash, std::meta::offset_of(e), std::meta::size_of(e));
    }
}
for (auto e : nonstatic_data_members_of(R)) {
    if (meta::test_type (^std::is_class_v, elem_type)) {
        hash = hash_combine(hash, get_abi_hash(meta::type_of(e), config));
    } else {
        if (meta::test_type (^std::is_pointer_v, elem_type)) {
        } else if (meta::test_type (^std::is_reference_v, elem_type)) {
        } else {
            auto name = std::string(meta::name_of(meta::type_of(e)));
            hash = hash_combine(hash, HASH_STR(name.c_str()));
        }
    }
}
for (auto e : nonstatic_data_members_of(R)) {
    if (meta::test_type (\^std::is_class_v, elem_type)) {
    } else {
        if (meta::test_type (\^std::is_pointer_v, elem_type)) {
            if (config.include_indirections) {
                hash = hash_combine(hash, HASH_STR("pointer"));
                constexpr auto ctype =
                    std::meta::substitute(\^std::remove_pointer_t, {elem_type});
                hash = hash_combine(hash, get_abi_hash(ctype, config));
            }
        }
    }
}
// Despite all members being the size, attributes like `packed` may change
// the size of the struct. Not everyone would be concerned with padding at
// the end though. Can consider making this optional via a config param.

if (std::meta::size_of(R) > 0) {
    hash = hash_combine(hash, std::meta::size_of(R));
}
consteval size_t get_abi_hash(std::meta::info R,
    meta::abi::ABIHashingConfig config = meta::abi::ABIHashingConfig{},
    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, this might have looked like this with type based code
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...>();
    }
}
A simple way to ensure compatibility of data layout across processes, without using protobuf etc

A solution that solves 80% of the common set of requirements of users
  - 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 ends

Inflexible and hard to modify / handle unique situations

ABI hashing - So what do we have now
ABI hashing - Where do we go from here?

- Full ABI textual representation!
  - Or, an ABI for describing the ABI
- 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.
A pythonic \texttt{std::any}

Hot take incoming
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!

// any type

std::any a = 1;
std::cout << a.type().name() << " : " << std::any_cast<int>(a) << '
';
a = 3.14;
std::cout << a.type().name() << " : " << std::any_cast<double>(a) << '
';
a = true;
std::cout << a.type().name() << " : " << std::any_cast<bool>(a) << '
';

// No duck typing sadly, but we got the equivalent of std::variant<everything>
// Can’t do much though :)

https://en.cppreference.com/w/cpp/utility/any
BOOST::VIRTUAL::ANY WITH DUCK TypING

STD::ANY

BOOST::PYTHON::OBJECT
How would an ideal code look?

class MyDuck {
public:
    int x;
    MyDuck(int x) : x(x) {}  
    std::string do_quack() const { return "quack"; }
};

auto a = make_virtual_any<int>(7);
std::cout << "Printing " << a << std::endl;

a = make_virtual_any<MyDuck>(MyDuck(3));
std::cout << "My obj " << a << " has .x == " << a.attr("x")
    << ", .do_quack() == " << a.attr("do_quack")() << std::endl;
WE CAN DO THIS!
Holding class: virtual_any

class virtual_any_interface;
class virtual_any {
    std::shared_ptr<virtual_any_interface> _impl;

public:
    virtual_any(std::shared_ptr<virtual_any_interface> elem) : _impl(elem) {};
    virtual_any attr(const std::string& name);
    virtual_any operator()();
    friend std::ostream& operator<<(std::ostream& os, const virtual_any& self);
};

class virtual_any_interface {
    public:
    virtual virtual_any attr(const std::string& name) = 0;
    virtual virtual_any call() = 0;
    virtual std::ostream& stream(std::ostream& os) const = 0;
};
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!
- Handle some special function operators natively
  - operator<<
  - operator()
  - operator+
  - Can handle more, but need to figure out how to handle variadic args
Implementation class : `virtual_any_impl`

```cpp
template <typename T>
class virtual_any_impl : public virtual_any_interface {
    T _value;
    constexpr std::vector<std::pair<std::string, virtual_any>> get_attrs(); // All members and methods of T

public:
    virtual_any_impl(T& value) : _value(std::forward<T>(value)) {}
    virtual virtual_any attr(const std::string& name) override;
    virtual virtual_any call() override; // .attr could've returned a std::function<virtual_any(void)>
    virtual std::ostream& stream(std::ostream& os) const override;
};
```
template <typename T>
constexpr std::vector<std::pair<std::string, virtual_any>> virtual_any_impl<T>::get_attrs() {
    using T2 = std::remove_cvref_t<T>;
    if constexpr (!meta::test_type(^std::is_class_v, ^T)) {
        return {};
    } else if constexpr (std::is_same_v<T2, std::string> ||
                         std::is_same_v<T2, std::function<virtual_any(void)>>) {
        return {};
    } else {
        std::vector<std::pair<std::string, virtual_any>> attrs;
        // Actual logic
        return attrs;
    }
}
template <typename T>
constexpr std::vector<std::pair<std::string, virtual_any>> virtual_any_impl<T>::get_attrs() {
    [:expand(meta::members_of(^T)):] >> [&]<auto e> {
        if constexpr (!meta::is_public(e)) return;
        else {
            auto name = std::string(meta::name_of(e));
            if constexpr (meta::is_nonstatic_data_member(e)) {
                attrs.push_back({name, make_virtual_any(_value.[:e:])});
            } else if constexpr (meta::is_function(e) && !meta::is_constructor(e) && !meta::is_destructor(e) && !meta::is_special_member(e)) {
                std::function<virtual_any(void)> l = [this]() { return make_virtual_any(_value.[:e]()); }; 
                attrs.push_back({name, make_virtual_any(l)});
            }
        }
    };
}
template <typename T>
virtual_any virtual_any_impl<T>::attr(const std::string& name) {
    auto attrs = get_attrs();
    if (attrs.size() == 0) {
        throw std::runtime_error("No attributes found");
    }
    for (const auto& [name2, e] : attrs) {
        if (name2 == name) {
            return e;
        }
    }
    std::stringstream ss;
    ss << "Attribute " << name << " not found in object of type " << std::meta::name_of(^T);
    throw std::runtime_error(ss.str());
}
**VIRTUAL ANY_IMPL: IMPLEMENT OTHER HELPER METHODS**

```cpp
template <typename T> virtual_any virtual_any_impl<T>::call() {
    if constexpr (std::is_same_v<T, std::function<virtual_any(void)>>) {
        return _value();
    } else {
        std::stringstream ss;
        ss << "Object of type " << std::meta::name_of(^T) << " is not callable";
        throw std::runtime_error(ss.str());
    }
}

template <typename T> std::ostream& virtual_any_impl<T>::stream(std::ostream& os) const {
    if constexpr (is_streamable<std::stringstream, T>::value) {
        os << _value;
    } else {
        os << "(non-streamable object of type " << std::meta::name_of(^T) << ")";
    }
    return os;
}
```
And there it is!
It works :)  

```cpp
int main() {
    auto a = make_virtual_any<int>(7);
    std::cout << "Printing " << a << std::endl;

    a = make_virtual_any<MyDuck>(MyDuck(3));
    std::cout << "My obj " << a << " has .x == " << a.attr("x") << std::endl;
    std::cout << "My obj " << a << " has .do_quack() == " << a.attr("do_quack"))() << std::endl;
}
```

```
$ ./virtual_any/main.out
Printing 7
My obj MyDuck(3) has .x == 3
My obj MyDuck(3) has .do_quack() == quack quack quack
```
Discussion

- Name is solid I know ;)
- Slow to run
  - Not the worst thing for writing script like code from time to time
  - Especially useful when dealing with containers and dictionaries and lists like in pythonic code
- Slow to compile
  - This one’s actually sad and hard to get around
  - Compilation can’t even be shared across compilation units
    - Unless... Language idioms
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 DSLs in C++?
- Removes need for complex ABI compatibility at module boundary
  - Sound familiar?
Alternatives to reflection

How did we survive without this till now?
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 only makes for cleaner APIs
  - There’s no “new” semantic thing we can do now that we couldn’t do
  - 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
enum VehicleType {
    UNDEFINED = 0;
    CAR = 1;
    TRUCK = 2;
    MOTORCYCLE = 3;
}

message Vehicle {
    string model = 1;
    VehicleType type = 2;
    repeated string features = 3;
}
{ "namespace": "example", "type": "record", "name": "Vehicle", "fields": [ { "name": "model", "type": "string" }, { "name": "type", "type": { "type": "enum", "name": "VehicleType", "symbols": [...] } }, { "name": "features", "type": { "type": "array", "items": "string" } } ] },
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](https://github.com/boost-ext/reflect)
  - Template metaprogramming to its limits
[FIN]

Saksham Sharma