C++26 reflection

Tonni Tielens

Freelance software developer and trainer

Programming mostly in C++ and C#

and Java

and Kotlin

and Rust

and Python

and TypeScript

```
<div>But I have no clue how
    to center a div</div>
```

Teaching mostly (Modern) C++

and Kotlin

and Clean Code

and Test-Driven Development

“Talk idea?”

“Reflection seems cool!”

Let’s start with some self-reflection

Thanks

Lieven de Cock
linkedin.com/in/lieven-de-cock-94535a2
✉ lieven@cppdriven.com

Reflection

Why reflection?

- Endless reinvention of the wheel:
  - Enum to string
  - to_string() functions for objects
  - Serialization systems
  - Command line argument parsing
  - ORMs
  - GUI/property editors
  - RPC frameworks
  - Enum to string
  - Game engines
  - Dependency injection frameworks
  - Test/mock frameworks
  - Did I already mention enum to string?
- Leading to:
  - Complex template meta programming (SFINAE, recursive std::enable_if, etc.)
  - Compiler hacks (anybody remember MAGIC_ENUM?)
  - External code generators (Qt moc, Unreal Header Tool, Python)
  - RTTI and vtables (limited and runtime)

What is reflection?

“Reflection is the ability of a program to **inspect** and **manipulate** its own structure: types, members, functions”

Difference with other languages

- In most languages (e.g. Java and C#), reflection happens at runtime
- In C++, reflection happens at compile time
  - More specifically, constant evaluation time
- Advantages
  - Performant
- Disadvantages
  - Bit harder to reason about, especially if you don’t understand compile time evaluation
  - Harder to mix with runtime

```
std::puts("Provide a function name: ");
std::string functionName;
std::cin >> functionName;

// How to call a function with the name provided by the user?
```

Compiler support

- ⩾ GCC 16.1
- clang-p2996 fork

Show me!

The lift operator ^^

From code to metadata

```cpp
struct Square {
    int width;
    int height;
    int getSurface() const;
};
void doSomething();
int i;
```

```cpp
constexpr std::meta::info intType = ^^int; // type
constexpr std::meta::info func = ^^doSomething; // function
constexpr std::meta::info method = ^^Square::getSurface; // member function
constexpr std::meta::info structType = ^^Square; // type again, this time struct
constexpr std::meta::info memberField = ^^Square::width; // member field
constexpr std::meta::info var = ^^i; // variable
```

A single opaque type

- A single opaque `std::meta::info`
- Reasoning/advantages
  - Protecting language evolution
  - Compiler neutral
  - Trivial extensibility
  - Heterogeneous collections without the need to fall back to inheritance, pointers or the risk of object slicing
- Disadvantages
  - Little use of autocomplete

Reflection kinds

- a value of scalar type
- an object of static storage duration
- a variable
- a structured binding
- a function
- a function parameter
- an enumerator
- an annotation
- a type alias
- a type
- a class member
- an unnamed bit-field
- a class template
- a function template
- a variable template
- an alias template
- a concept
- a namespace alias
- a namespace
- a direct base class relationship or
- a data member description

- is_value
- is_object
- is_variable
- is_structured_binding
- is_function
- is_parameter
- is_enumerator
- is_annotation
- is_type_alias
- is_type
- is_class_member
- is_bit_field
- is_class_template
- is_function_template
- is_variable_template
- is_alias_template
- is_concept
- is_namespace_alias
- is_namespace
- is_base
- is_data_member_spec

```cpp
int i;
constexpr std::meta::info iVar = ^^i;

if constexpr (is_variable(iVar))
    std::println("It's a variable!");
else
    std::println("It's something different!");

// Output:
// It's a variable!
```

The splice operator [: :]

From metadata to code

```
constexpr auto intType = ^^int; // type
typename [: intType :] j = 42;
```

```
constexpr auto func = ^^doSomething; // function
[: func :]();
```

```cpp
Square square;
constexpr auto method = ^^Square::getSurface; // member function
square.[: method :]();
```

```
Square square;
constexpr auto memberField = ^^Square::width; // member field
square.[: memberField :] = 43;
```

```
constexpr auto ns = ^^std;
[: ns :]::string str{"Hello, World!"};
```

```cpp
int i = 43;
constexpr auto var = ^^i;
[: var :] = 44;
std::println("i = {}", i);

// Output:
// i = 44
```

Stay with me for a bit more dry theory

identifier_of

```cpp
int i = 43;
constexpr auto var = ^^i;
std::println("identifier = {}", identifier_of(var));

// Output:
// identifier = i
```

```cpp
constexpr auto personType = ^^Person; // type
std::println("identifier = {}", identifier_of(personType));

// Output:
// identifier = Person
```

```
constexpr auto intType = ^^int; // type
std::println("identifier = {}", identifier_of(intType));

// Compiler output:
// error: uncaught exception of type 'std::meta::exception'; 'what()': 'reflection
// with has_identifier false'
```

has_identifier

```
constexpr auto intType = ^^int; // type

if constexpr (has_identifier(intType))
    std::println("identifier = {}", identifier_of(intType));
```

display_string_of

```cpp
constexpr auto intType = ^^int; // type
std::println("display_string = {}", display_string_of(intType));

constexpr auto memberField = ^^Square::width; // member field
std::println("display_string = {}", display_string_of(memberField));

// Output:
// display_string = int
// display_string = Square::width
```

type_of

```cpp
int i = 3;
constexpr auto iVar = ^^i;
constexpr auto iType = type_of(iVar);

std::println("The type of {} = {}", identifier_of(iVar), display_string_of(iType));

// Output:
// The type of i = int
```

source_location_of

```cpp
int i = 43;
constexpr auto var = ^^i;
std::source_location location = source_location_of(var);
std::println("{} ({}:{}) `{}`", location.file_name(), location.line(), location.column(),
    location.function_name());

// Output:
// /app/example.cpp (94:9) `int main()`
```

tonni tielens
single solution

to_string for objects

```
template <typename T>
std::string to_string(const T& obj) {
```

```cpp
template <typename T>
std::string to_string(const T& obj) {
    std::string result;

    return result;
}
```

tonni tielens
single solution

```cpp
template <typename T>
std::string to_string(const T& obj) {
    std::string result;
    auto out = std::back_inserter(result);
    std::format_to(out, "{}{{", identifier_of(^^T));

    return result;
}
```

```cpp
template <typename T>
std::string to_string(const T& obj) {
    std::string result;
    auto out = std::back_inserter(result);
    std::format_to(out, "{}{{", identifier_of(^^T));
    bool isFirst = true;
    template for (
      constexpr auto member : define_static_array(nonstatic_data_members_of(^^T, std::meta::access_context::current()))) {
    }
    std::format_to(out, "}}");

    return result;
}
```

current() → normal access rules from current location
unchecked() → everything
unprivileged() → public only

```cpp
template <typename T>
std::string to_string(const T& obj) {
    std::string result;
    auto out = std::back_inserter(result);
    std::format_to(out, "{}{{", identifier_of(^^T));
    bool isFirst = true;
    template for (
        constexpr auto member : define_static_array(nonstatic_data_members_of(^^T, std::meta::access_context::current()))) {

        if (!isFirst)
            std::format_to(out, ", ");
        else
            isFirst = false;

        std::format_to(out, "{}: {}", identifier_of(member), obj.[: member :]);
    }
    std::format_to(out, "}}");

    return result;
}
```

```cpp
template <typename T>
std::string to_string(const T& obj) {
    std::string result;
    auto out = std::back_inserter(result);
    std::format_to(out, "{}{{", identifier_of(^^T));
    bool isFirst = true;
    template for (
        constexpr auto member : define_static_array(nonstatic_data_members_of(^^T, std::meta::access_context::current()))) {

        if (!isFirst)
            std::format_to(out, ", ");
        else
            isFirst = false;

        std::format_to(out, "{}: {}", identifier_of(member), obj.[: member :]);
    }
    std::format_to(out, "}}");

    return result;
}

Square sqr{ .width = 4, .height = 3 };
auto str = to_string(sqr);
std::println("{}", str);

// Output:
// Square{width: 4, height: 3}
```

tonni tielens
single solution

Looks cool,
but you promised
enum to string!
</>

to_string for enums

```cpp
template <typename T>
requires std::is_enum_v<T>
std::string_view to_string(T val)
{
    std::string_view result = "unknown";
    template for (constexpr auto enumConstant : define_static_array(enumerators_of(^^T)))
    {
        if (val == [: enumConstant :])
        {
            result = identifier_of(enumConstant);
        }
    }
    return result;
}
```

```cpp
template <typename T>
requires std::is_enum_v<T>
std::string_view to_string(T val)
{
    std::string_view result = "unknown";
    template for (constexpr auto enumConstant : define_static_array(enumerators_of(^^T)))
    {
        if (val == [: enumConstant :])
        {
            result = identifier_of(enumConstant);
        }
    }
    return result;
}

Color color = Color::Green;
std::println("{}", to_string(color));

// Output:
// Green
```

tonni tielens
single solution

More?

Code ‘generation’

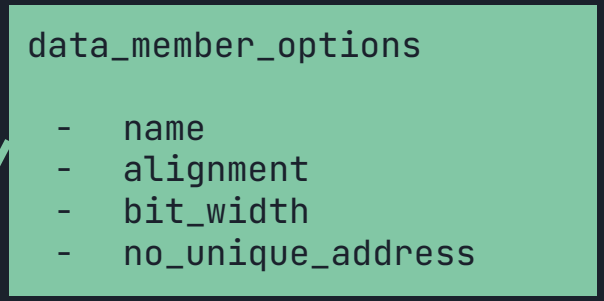
```
struct MyStruct;
```

```
struct MyStruct;
consteval {
    define_aggregate(^^MyStruct, {});
}

MyStruct myObj;
```

```cpp
struct MyStruct;
consteval {
    define_aggregate(^^MyStruct, {
        data_member_spec(^^int, { .name = "age" }),
        data_member_spec(^^char, { .name = "type" }),
        data_member_spec(^^double, { .name = "amount" }),
    });
}

MyStruct myObj{ .age = 45, .type = 'D', .amount = 3.14 };
```



Allows for many things, for example ‘Wire’ structs

```cpp
template <typename T>
consteval auto clone_layout() {
   return
       nonstatic_data_members_of(^^T, std::meta::access_context::current()) |
           std::views::transform([](auto field) {
               return data_member_spec(
                   type_of(field),
                   { .name = identifier_of(field) }
               );
           });
}

struct MyStructWired;
consteval {
   define_aggregate(^^MyStructWired, clone_layout<MyStruct>());
}
```

Annotations

Annotations

- C++ already had attributes ([[nodiscard]], [[fallthrough]], ...)
- C++26 introduces annotations which are strongly typed C++ expressions [[= expr]]
- Can be queried with std::meta::annotations_of and std::meta::annotations_of_with_type

```
namespace json {
    inline constexpr struct {} Skip;
    struct Rename {
        char name[32]{};
    };
}

struct Person {
    [[= json::Skip]]
    int id{};

    int age{};

    [[= json::Rename("name")]]
    std::string full_name;
};
```

```cpp
template <typename T>
std::string serialize(const T& obj) {
    std::string result;
    auto out = std::back_inserter(result);
    std::format_to(out, "{{");
    bool isFirst = true;

    template for (constexpr auto field : define_static_array(nonstatic_data_members_of(^^T, std::meta::access_context::current()))) {
        // Ignore fields that must be skipped.
        if constexpr (!annotations_of_with_type(field, type_of(^^json::Skip)).empty())
            continue;

        if (!isFirst)
            std::format_to(out, ", ");
        else
            isFirst = false;

        // Use custom rename if required.
        std::string_view field_name;
        constexpr auto renames = define_static_array(annotations_of_with_type(field, ^^Rename));
        if constexpr (!renames.empty()) {
            // Extract typed value from reflection handle
            field_name = std::meta::extract<Rename>(renames[0]).name;
        } else {
            field_name = identifier_of(field);
        }

        std::format_to(out, R"("{}": {:?})", field_name, obj.[: field :]);
    }

    std::format_to(out, "}}");

    return result;
}
```

tonni tielens
single solution

```
struct Person {
    [[= json::Skip]]
    int id{};

    int age{};

    [[= json::Rename("name")]]
    std::string full_name;
};
```

```cpp
struct Person {
    [[= json::Skip]]
    int id{};

    int age{};

    [[= json::Rename("name")]]
    std::string full_name;
};

int main() {
    Person p{.id = 1, .age = 45, .full_name = "Tonni Tielens"};

    std::print("{}", json::serialize(p));
}
```

```cpp
struct Person {
    [[= json::Skip]]
    int id{};

    int age{};

    [[= json::Rename("name")]]
    std::string full_name;
};

int main() {
    Person p{.id = 1, .age = 45, .full_name = "Tonni Tielens"};

    std::print("{}", json::serialize(p));
}

// Output:
// {"age": 45, "name": "Tonni Tielens"}
```

The possibilities are endless

Conclusion

- Reflection lets your code reason about its own structure at compile time: no macros, no external codegen, no runtime costs.
- The ^^ operator turns a language construct into the opaque std::meta::info. [: :] turns it back.
- Common use cases are: serialization, enum to string, logging, test factories, ORM. All with zero boilerplate per type.
- It's compile time, read-only, and standardized. No RTTI overhead. Errors at compile time, not runtime.
- C++26 is phase 1. More is coming.

Thanks!

www.tonni.nl

tonni@tonni.nl