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

But I still don’t know how to center a div

Teaching mostly (Modern) C++

and Kotlin

and Clean Code

and Test-Driven Development

“Talk idea?”

“Sure. Reflection seems cool!”

Reflection

Thanks

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

Why reflection?

- Fighting the language, and keep building the same things externally:
  - Enum conversion
  - Serialization systems
  - to_string() functions for objects
  - Command line argument parsing
  - ORMs
  - GUI/property editors
  - RPC frameworks
  - Enum conversion
  - Game engines
  - Dependency injection frameworks
  - Test frameworks
  - Did I already mention enum conversion?
- Leading to:
  - Hard to understand template meta programming
  - MACROs (anybody remember MAGIC_ENUM?)
  - Misuse of RTTI (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?
```

Show me!

the lift operator ^^

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


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
```

^^ can be used on

- 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.

the splice operator [: :]

Allows splicing back reflections into source 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 :]();
```

```cpp
Square square;
constexpr std::meta::info memberField = ^^Square::width; // member field
square.[: memberField :] = 43;
```

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

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

Stay with me for a bit more raw theory

identifier_of

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

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

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

// Compiler output:
// error: call to consteval function 'std::meta::identifier_of(^^int)' is not a constant
// expression
// error: error: uncaught exception of type 'std::meta::exception'; 'what()': 'reflection
// with has_identifier false'
```

has_identifier

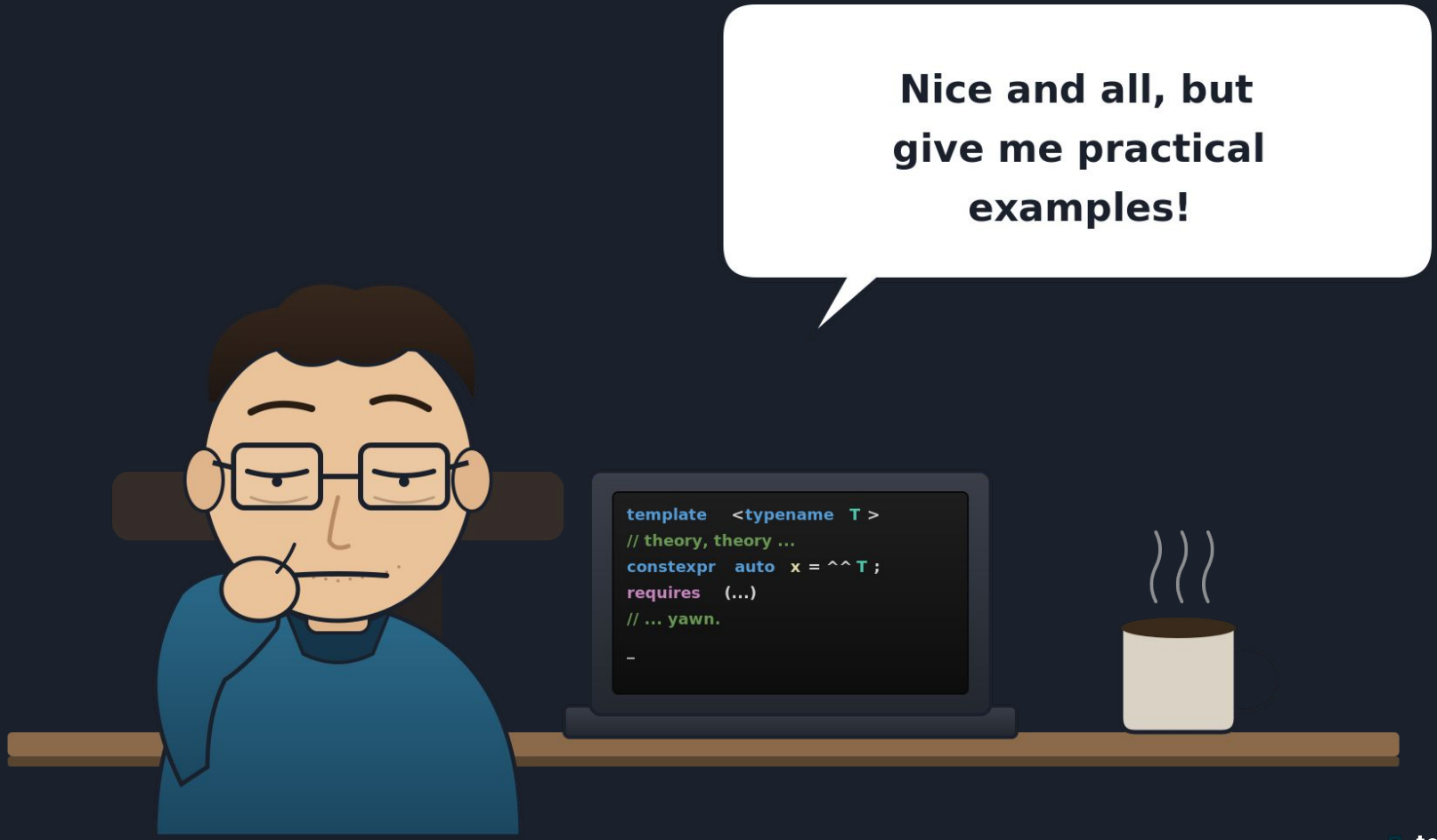
```cpp
constexpr auto intType = ^^int; // type

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

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

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()`
```



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;
}
```

```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;
}
```

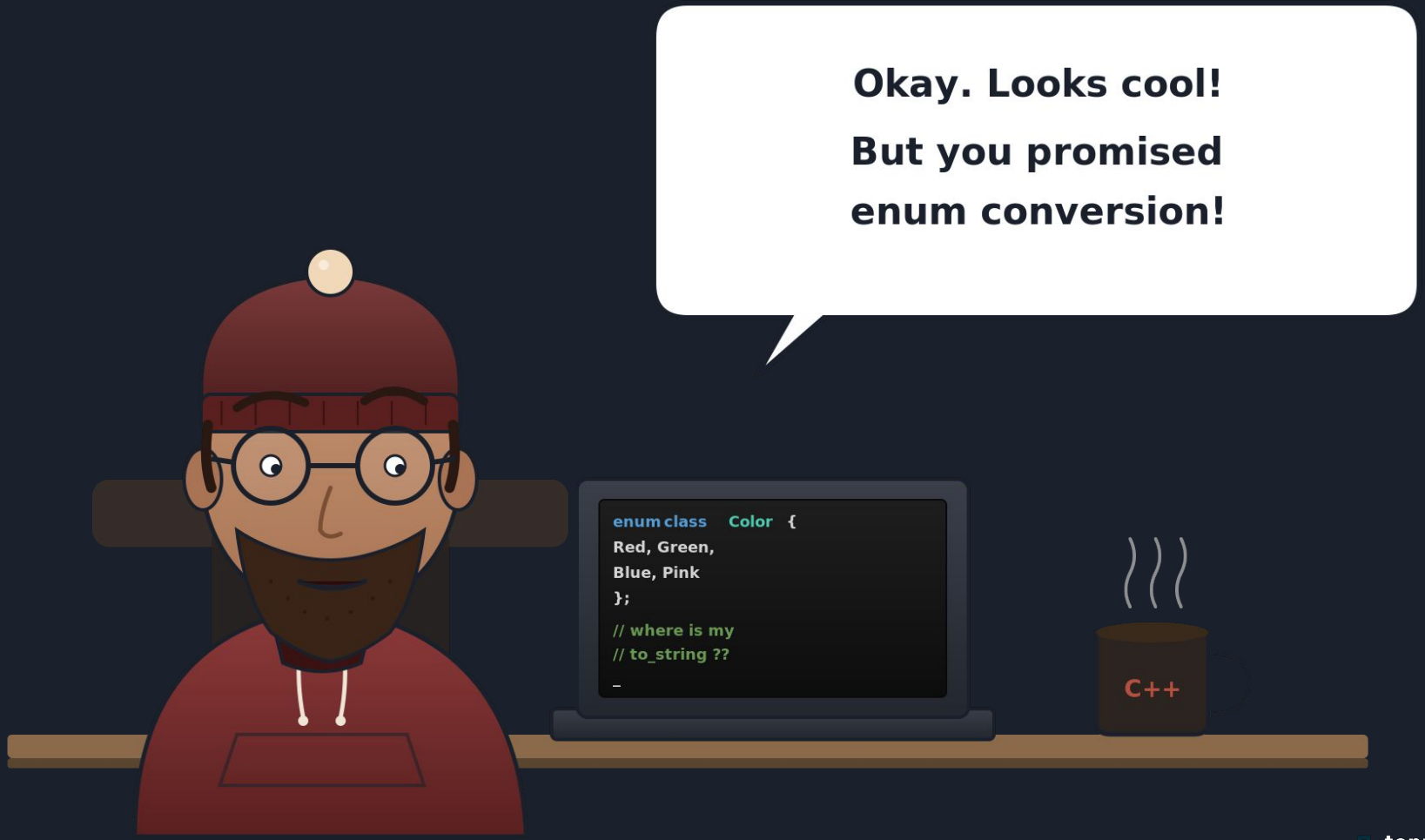
```cpp
template <typename T>
std::string to_string(const T& obj)
{
    std::string result;
    auto out = std::back_inserter(result);

    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}
```



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

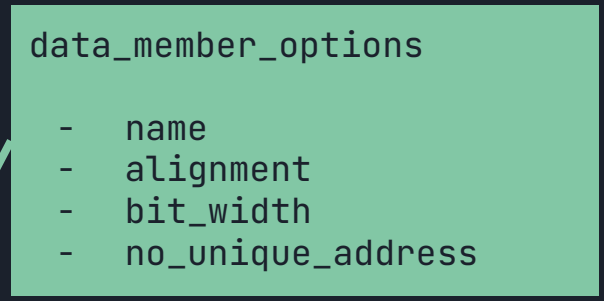
More?

Code ‘generation’

```
struct MyStruct;
```

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

MyStruct myObj;
```

```
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()
{
    std::vector<std::meta::info> members;

    template for (constexpr auto m : define_static_array(nonstatic_data_members_of(^^T,
        std::meta::access_context::current()))) {

        members.push_back(
            data_member_spec(
                type_of(m),
                { .name = identifier_of(m) }
            )
        );
    }

    return define_static_array(members);
}

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

There’s more. Much more!

More is coming to C++

- Code injection (maybe?)
- Annotations
- std::meta utilities

Conclusion

- Reflection lets your code reason about its own structure at compile time: no macros, no 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.

More

- Reflection for C++26 paper: https://isocpp.org/files/papers/P2996R13.html

Questions?

Thanks!

 www.tonni.nl

 tonni@tonni.nl