Metadata exhibition

struct fields exposure

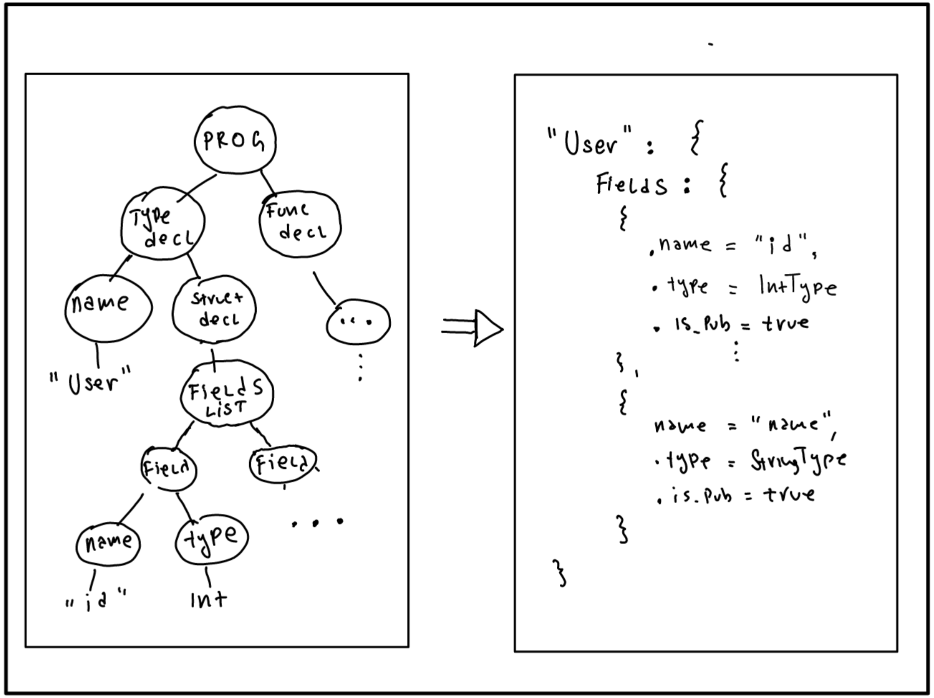
Struct

```cpp
#include <string>
#include <fmt/core.h>

struct User {
    int id;
    std::string name;
};

int main() {
    auto u = User {.id = 1, .name = "John"};
    fmt::print("id: {}, name: {}", u.id, u.name);
}
```

Struct

```cpp
#include <string>
#include <fmt/core.h>

struct User {
    int id;
    std::string name;
};

int main() {
    auto u = User {.id = 1, .name = "John"};
    fmt::print("id: {}, name: {}", u.id, u.name);
}
```

Compiler



WHAT HAPPENS IN

~~VEGAS~~ Compiler

STAYS IN

~~VEGAS~~ Compiler

Right?

Well ….

Exposing compiler internals to the user can be very handy

Example: Enum to string

```
template <typename E>
  requires std::is_enum_v<E>
constexpr std::string enum_to_string(E value) {
  template for (constexpr auto e : std::meta::members_of(^E)) {
    if (value == [:e:]) {
      return std::string(std::meta::name_of(e));
    }
  }

  return "<unnamed>";
}

enum Color { red, green, blue };
static_assert(enum_to_string(Color::red) == "red");
static_assert(enum_to_string(Color(42)) == "<unnamed>");
```

Struct metadata

OK, but why??

- Reduce boilerplate code (code generation)
- Powerful plug-n-play libraries
  - (de)Seriallization (JSON, YAML, protobuf, ...)
  - Domain-specific apstractions (SQL, html, ...)
- API design and documentation (ex. Web)
- Type level programming (TMP, Zig, Haskell)

Fetch struct “metadata”

Approaches:

- Reflection: Go, Zig, cpp2, c++26?..
- Macros (AST interception): Rust, Lisp, Nim
- Custom reflection system (library): C++, ….

Reflection

Reflection is a mechanism composed of two techniques:

- Introspection - The ability for a program to examine itself
- Intercession - The ability for a program to modify itself (his behaviour or his state)

Talk by Dusan Jovanovic: (Serbian audience only)
Object serialization in C++
https://youtu.be/Feu49_CmbDs

Reflection (Zig)

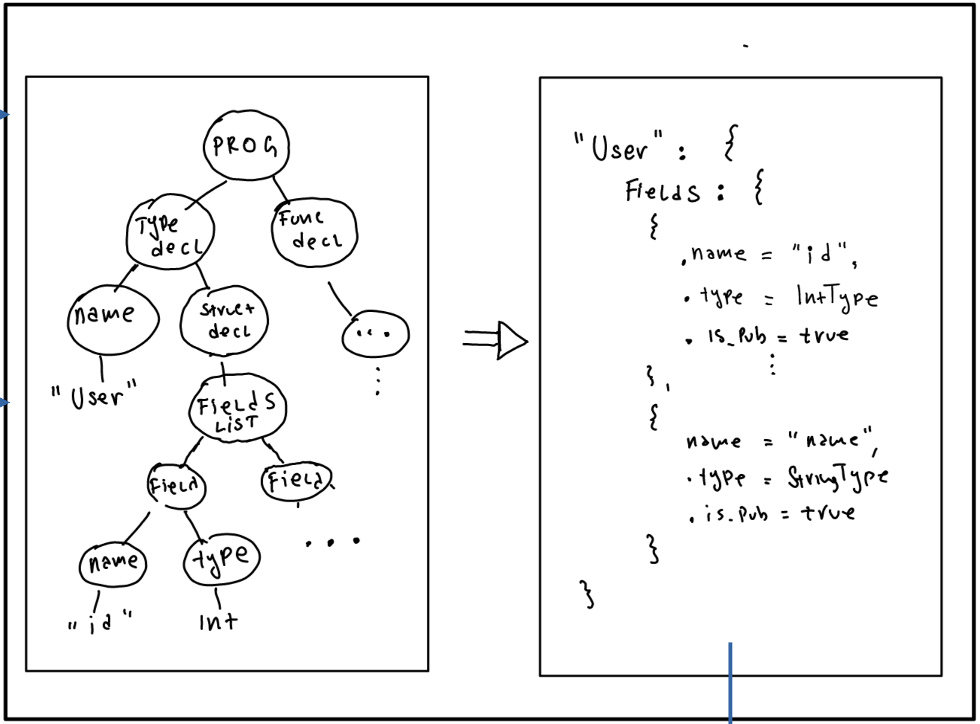
```zig
const std = @import("std");

const User = struct {
    id: i32,
    name: []const u8,
};

pub fn main() void {

    const user = User{ .id = 1, .name = "John"};

    inline for (std.meta.fields(User)) |f| {

        std.debug.print("name: {s}, ", .{f.name});
        std.debug.print("type: {s}, ", .{@typeName(f.field_type)});

        var value = @field(user, f.name);

        switch (f.field_type) {
            i32 => std.debug.print("value: {}", .{value}),
            [] const u8 => std.debug.print("value: \"{s}\"", .{value}),
            else => @compileError("Unsupported type"),
        }

        std.debug.print("\n", .{});
    }
}
```

Output:
    name: id, type: i32, value: 1
    name: name, type: []const u8, value: "John"

Reflection (Zig)

```zig
const std = @import("std");

const User = struct {
    id: i32,
    name: []const u8,
};

pub fn main() void {

    const user = User{ .id = 1, .name = "John"};

    inline for (std.meta.fields(User)) |f| {

        std.debug.print("name: {s}, ", .{f.name});
        std.debug.print("type: {s}, ", .{@typeName(f.field_type)});

        var value = @field(user, f.name);

        switch (f.field_type) {
            i32 => std.debug.print("value: {}", .{value}),
            [] const u8 => std.debug.print("value: \"{s}\"", .{value}),
            else => @compileError("Unsupported type"),
        }

        std.debug.print("\n", .{});
    }
}
```

Compiler



Demo time
Golang

Macros

What macros are?

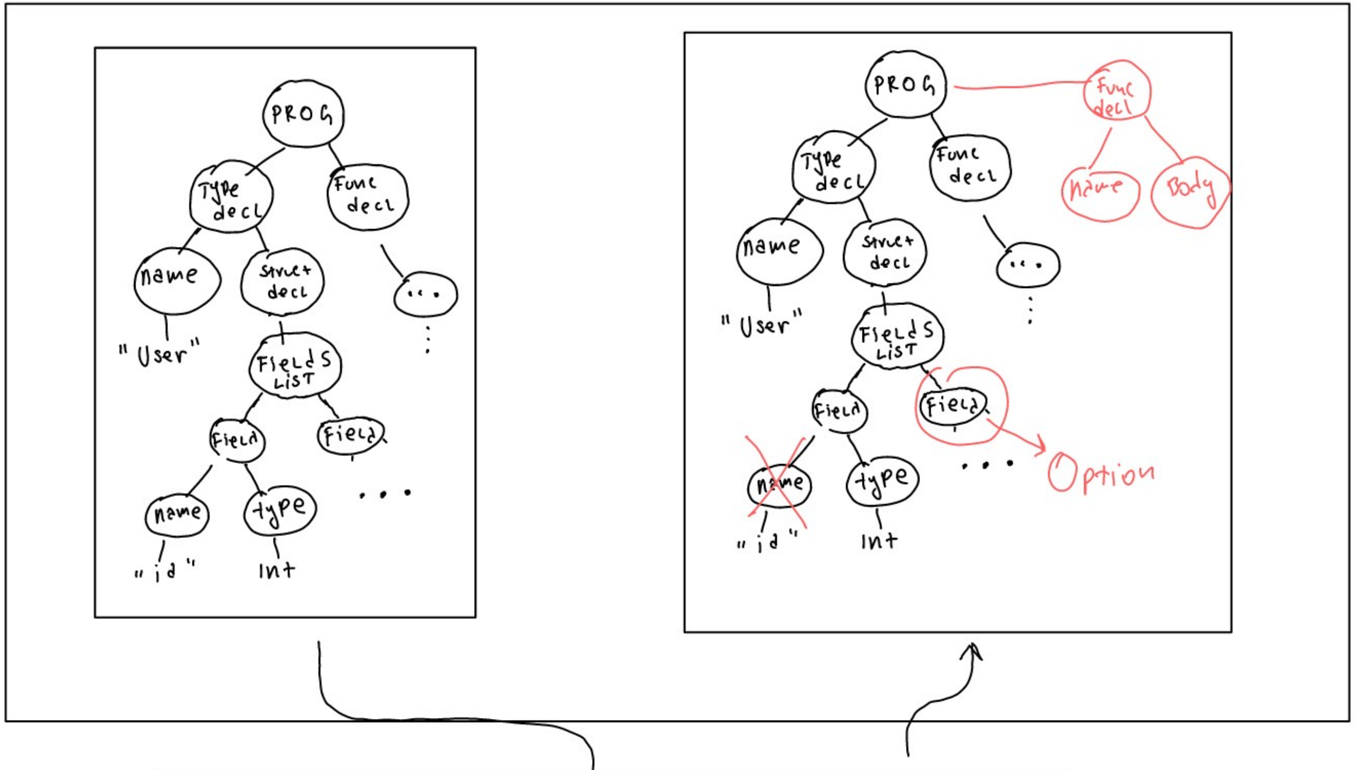
Macros are reusable code snippets or transformations, allowing developers to write more concise, flexible, and maintainable code.

Expansion of a macro occurs during compilation or preprocessing to generate the corresponding code.

Rust macros

- Declarative Macros (macro_rules!)
- Procedural Macros
  - Function-like proc macro
  - Derive proc macro
  - Attribute-like proc macro

Rust (behind scene)




```rust
pub fn my_macro(input: TokenStream) -> TokenStream {
    // ... macro definition
}
```

Rust: declarative macro

```rust
// Taken from:
// https://doc.rust-lang.org/beta/reference/macros-by-example.html

macro_rules! print_anything {
    ($input:tt) => {
        let output = stringify!($input);
        println!("{}", output);
    };
}

pub fn main() {
    print_anything!(foo0);
    print_anything!(1bar);
}
```

Rust: Procedural macros

```rust
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct User {
    id: i32,
    name: String,
}

fn main() {
    let user = User { id : 1, name: String::from("John") };

    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&user).unwrap();
    println!("serialized = {}", serialized);

    // Convert the JSON string back to User
    let deserialized: User = serde_json::from_str(&serialized).unwrap();

    // Print user by leveraging Debug macro
    println!("deserialized = {:?}", deserialized);
}
```

Rust: Procedural macros

```rust
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize, Debug)]
struct User {
    id: i32,
    name: String,
}

fn main() {
    let user = User { id : 1, name: String::from("John") };

    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&user).unwrap();
    println!("serialized = {}", serialized);

    // Convert the JSON string back to User
    let deserialized: User = serde_json::from_str(&serialized).unwrap();

    // Print user by leveraging Debug macro
    println!("deserialized = {:?}", deserialized);
}
```

```
Output >:
  serialized = {"id":1,"name":"John"}
  deserialized = User { id: 1, name: "John" }
```

CPP

C++ (with boost::hana)

```cpp
#include <fmt/core.h>
#include <boost/hana.hpp>
#include <boost/hana/adapt_struct.hpp>

struct User {
    uint64_t id;
    std::string firstname;
};

BOOST_HANA_ADAPT_STRUCT(User, id, firstname);

int main()  {

    User u {
        .id = 0,
        .firstname = "Nebojsa",
    };

    auto umap = boost::hana::to_map(u);
    boost::hana::for_each(umap, [](const auto& p) {
            fmt::print("{} -> {}\n", boost::hana::first(p).c_str(), boost::hana::second(p));
    });

    return 0;
}
```

The End