

Refleksja oraz metaklasy w C++

W dążeniu do prostszego oraz efektywniejszego
kodu w C++

Motywujący przykład

- Projekt z wieloma strukturami, które ciągle się zmieniają i dochodzą nowe
- Wiele struktur musi mieć zapewnione pewne cechy, np. porównanie, serializacja, haszowanie
- Problem projektowy - jak zapewnić, aby te cechy były poprawnie zaimplementowane?

Motywujący przykład - przymiarka

```
class example {
    uint64_t id;
    uint64_t ts;
    std::string name;
    friend std::ostream& operator<<(std::ostream& os, const example& e) {
        return os << "example(id : " << e.id
            << " ts : " << e.ts
            << " name : " << e.name << ")";
    }
    std::string to_json() const {
        std::ostringstream output;
        output << "{" << "\"id\" : " << id
            << "\"ts\" : " << ts
            << "\"name\" : " << name << "}";
        return output.str();
    }
};
```

Motywujący przykład - propozycje

- Ręczna edycja nie wchodzi w grę - zbyt duży narzut na zmiany i testowanie, duże ryzyko błędów
- Użycie zewnętrznych narzędzi do generowania kodu - trudne/niemożliwe do rozszerzania
- Użycie preprocesora

Motywujący przykład - lepiej

```
create_class( example,  
    ((id) (uint64_t))  
    ((ts) (uint64_t))  
    ((name) (std::string))  
    , printable  
    comparable  
    json_serializable  
)
```

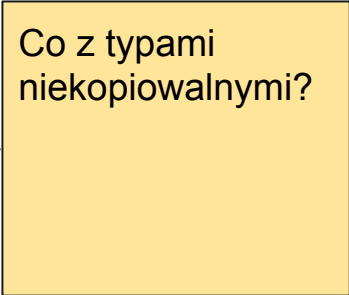
Makro `create_class`
posiada:

- nazwę klasy
- pola
- własności

create_class - get, set, ref

```
#define create_field_accessors(r, sep, i, elem) \  
    field_type(elem) BOOST_PP_CAT(get, field_name(elem))() const {\ \  
        return this->field_name(elem); \  
    } \  
    field_type(elem)& BOOST_PP_CAT(ref, field_name(elem))() const {\ \  
        return this->field_name(elem); \  
    } \  
    const field_type(elem)& BOOST_PP_CAT(ref, field_name(elem))() const {\ \  
        return this->field_name(elem); \  
    } \  
    void BOOST_PP_CAT(set, field_name(elem))(field_type(elem) el) { \  
        this->field_name(elem) = el; \  
    } \  
}
```

Co z typami
niekopiowalnymi?



create_class - printable

```
#define field_name(field) BOOST_PP_SEQ_ELEM(1, elem)
#define print_field(r, sep, i, elem) \
    BOOST_PP_IF(i ,<< ", ")\
    << BOOST_PP_STRINGIZE(field_name(elem)) << ": " \
    << record.BOOST_PP_CAT(ref,field_name(elem))()

#define create_printable(name, fields) \
    public: \
    friend std::ostream& operator<< (std::ostream& os, const name& record) \
    { \
        return os << name << '(' \
            BOOST_PP_SEQ_FOR_EACH_I(print_field,, fields) \
            << '); \
    }
```

Makro
create_class
tworzy dla
każdego pola
metody ref, get,
set.

create_class - json_serializable

```
#define json_field(r, sep, i, elem) \  
    BOOST_PP_IF(i, output << ", " ) \  
    output << "\"" << field_name(elem) << "\" : "; \  
    if constexpr (std::is_fundamental_v<field_type(elem)>) { \  
        output << this->BOOST_PP_CAT(ref, field_name(elem)); \  
    } /* MORE DISPATCHES */ \  
    else { output << field_name(elem).to_json(); } \  
#define create_json_serializer(name, fields) \  
    public: \  
        std::string to_json() const { \  
            std::ostringstream output; output << "{"; \  
            BOOST_PP_SEQ_FOR_EACH_I(json_field,,fields) \  
            output << "}"; \  
            return output.str(); \  
        } \  
    }
```

Warunkowa
produkcja kodu w
czasie działania
preprocessora

Warunkowa
kompilacja kodu

create_class - korzyści

- Zunifikowany interfejs klasy
 - Automatyczne dostosowanie się do konwencji projektowych
- Prosta rozszerzalność
 - Potrzebujemy, aby struktura była serializowana do protokołu X? Implementujemy generyczny X_serializable

create_class - wady

- Mocno nieczytelny kod implementacji własności klas
- Bardzo wydłużone czasy kompilacji
 - Rozwijanie makr preprocesora z boosta jest wolne
- Podawanie domyślnych implementacji jest uciążliwe
 - Dodanie nowej własności, np.
example_json_serializable

Alternatywa dla preprocesora

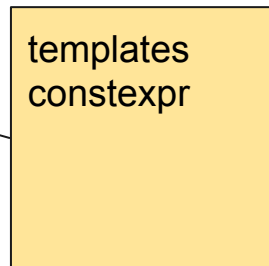
Do stworzenia efektywnej alternatywy potrzebujemy mechanizmów:

- Refleksji
- Syntezy/iniekcji kodu
- Kontroli przepływu informacji

Alternatywa dla preprocesora

Do stworzenia efektywnej alternatywy potrzebujemy mechanizmów:

- Refleksji
- Syntezy/iniekcji kodu
- Kontroli przepływu informacji



Propozycje refleksji

- Type syntax [\[P0385\]](#)
- Heterogeneous value syntax [\[P0590\]](#)
- Homogeneous value syntax [\[P0598\]](#)

Dwa słowa o refleksji

- Metaobiekt a obiekt
- Introspekcja
- Reifikacja

Refleksja - type syntax

Informacje o bytach zawarte są w typach:

```
using example_m = $reflect(example);  
using example_members_m = reflect::get_data_members_t<example_m>;  
reflect::for_each<example_members_m>(  
    [](auto example_field) {  
        using example_field_m = decltype(example_field);  
        cout << reflect::get_name_v<example_field_m> << "\n";  
    }  
);
```

Chwilowa
“materializacja”
meta-objektów

Refleksja - Heterogeneous VS

Informacje o bytach znajdują się w constexprowych obiektach o różnych typach

```
constexpr auto example_m = $reflect(example);  
constexpr auto example_members_m = example_m.get_data_members();  
for... (auto field_member : example_members_m) {  
    cout << field_member.name() << '\n';  
}
```

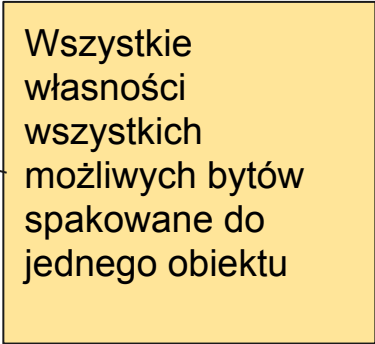
Pola na przykład mogą posiadać funkcje:

```
bool is_mutable()  
T C::* pointer()  
auto type()  
const char* name()  
access_t access()
```


Refleksja - Homogeneous VS

Informacje o bytach znajdują się
w obiektach zawsze tego samego typu

```
constexpr meta::info example_m = $reflect(example);  
constexpr constexpr_vector<meta::info> example_members  
= example_m.get_data_members(example_m);  
for (auto field_member : example_members) {  
    cout << field_member.name().value_or("unnamed");  
}
```



Wszystkie
własności
wszystkich
możliwych bytów
spakowane do
jednego obiektu

Refleksja - Homogeneous VS

Informacje o bytach znajdują się
w obiektach zawsze tego samego typu

```
constexpr meta::info example_m = $reflect(foo);  
constexpr constexpr_vector<meta::info> example_members  
= get_public_data_members(example_m);  
for (auto field_member : example_members) {  
    cout << field_member.name().value_or("unnamed");  
}
```

Możemy jak
obecnie
programować w
constexpr

Wszystkie
własności
wszystkich
możliwych bytów
spakowane do
jednego obiektu

Iniekcja kodu

- Bezpośrednia iniekcja tokenów/kodu
- Programowalne API
- Metaklasy

Bezpośrednia iniekcja tokenów

Za pomocą operatora `-> {}` możemy wstrzyknąć tokeny do obecnego zakresu

```
template <typename F, typename T>
void for_each(F&& f, T&& t) {
    constexpr auto members_m = $reflect(T).get_data_members();
    for... (auto member : members_m) {
        -> { f( t.get(. member.name() .)() ); };
    }
}
auto printer = [](auto v) { cout << v << '\n'; };
for_each(printer, example{1, 2, "Jan"});
```

Operator `(. .)` pozwala na wstrzyknięcie wyniku operacji wewnątrz operatora.

Programowalne API

Wstrzykiwanie kodu odbywa się za pomocą API

```
template <typename F, typename T>
void for_each(F&& f, T&& t) {
    constexpr auto members_m = $reflect(T).get_data_members();
    for... (auto member : members_m) {
        meta::queue(
            meta::expr_statement(meta::call_by_name("f",
                meta::member_access("t", meta::concat("get",
                    meta::name(member)
                ))
            ))
        );
    }
}
```

Iniekcja kodu



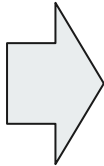
`expr_statement(...)`
`concat(...)`
`call_by_value(...)`

-> `{(.)}`
-> `{(.)}`
-> `{(.)}`
-> `{(.)}`
-> `{(.)}`
-> `{(.)}`

Pierwsza metaklasa - class

Kompilator

```
class Point {  
  int x, y;  
};
```



```
for ( m : members ) {  
  if (!m.has_access())  
    if (is_class())  
      m.make_private();  
  else  
    m.make_public();  
}  
  
for ( f : functions ) {  
  // ...  
}  
  
// wg21.link/standard
```



```
class Point {  
private:  
  int x, y;  
public:  
  Person() = default;  
  ~Person() noexcept = default;  
  Point(const Point&) = default;  
  Point& operator=(const Point&) = default;  
  Point(Point&&) = default;  
  Point& operator=(Point&&) = default;  
};
```

Nowe dodatki - \$reflect

- Operator \$reflect przeprowadza introspekcję
 - Zamiana konkretnego obiektu na metaobiekt
- \$reflect(main) da nam metaobiekt funkcji main
 - Możliwe funkcje w metaobiekcie Function to:
 - `bool is_constexpr()`, `bool is_deleted()`
`bool is_noexcept()`, `Tuple parameters()`
`bool is_defined()`, `T(*)(...) pointer()`
`bool is_inline()`

Nowe dodatki - `->{}` oraz `(. .)`

- Operator `->{}` oznacza iniekcję
- `-> { int abc; };`
 - W czasie kompilacji wstrzykuje blok kodu, który zawiera zmienną `abc`
 - `(. .)` wykonuje operację wewnątrz i jej wynik jest przekazywany do wstrzyknięcia
- Może być tylko użyty tylko w czasie kompilacji

Nowe dodatki - constexpr {}

- constexpr do tworzenia obliczeń w czasie kompilacji
- constexpr możemy mieć:
 - obiekty
 - lambdy, pętle
 - if-y, switche
- Kolejną konsekwencją jest constexpr block
 - `constexpr {stmts} = constexpr [](){stmts}();`

Nowe dodatki - koncepty

- Pozwalają nam robić zapytania dla dowolnego typu
 - Czy wyrażenie E dla typu T jest poprawne?
 - `template <typename T>`
`concept bool Addable = requires (T a) { a + a };`
 - `template <Addable T>`
`T add(T a, T b) { return a + b; }`
- Ładniejsze błędy diagnostyczne

Nowa metaklasa - interface

```
$class interface {  
    virtual ~interface() noexcept = default;  
    constexpr {  
        compiler.require($reflect(prototype).member_variables().empty(), "interfaces may not contain data members");  
        for... (auto f : $reflect(prototype).member_functions()) {  
            compiler.require(f.is_public(), "interface functions must be public");  
            compiler.require(!f.is_copy() && !f.is_move(), "interfaces may not copy or move; consider a virtual clone()");  
            compiler.require(!f.is_defined(), "interface functions should not contain default definition");  
            f.make_pure_virtual();  
            -> { f };  
        }  
    }  
}
```

Interface - użycie

```
interface Shape {  
    int area() const;  
    void scale_by(double factor);  
};  
  
class Rectangle : public Shape {  
    int area() const override { return 42; }  
    void scale_by(double) override {}  
};  
  
int main() {  
    Rectangle r;  
    compiler.debug($Shape);  
    compiler.debug($Rectangle);  
    compiler.debug($r);  
}
```

OUT

```
struct Shape {  
    virtual ~Shape() noexcept = default;  
    virtual int area() const = 0;  
    virtual void scale_by(double factor) = 0;  
}  
  
class Rectangle : public Shape {  
public:  
    int area() const override {  
        return 42;  
    }  
    void scale_by(double) override {  
    }  
}  
  
Rectangle r
```

Interface - korzyści

- Prostota
 - W razie pomyłki dostajemy ładny błąd kompilatora
 - Zamknęliśmy kilka wymagań oraz transformację w prostym, czytelnym kodzie
- Używanie interface jasno komunikuje czytającemu kod czym jest nasza struktura

create_class w lepszym wydaniu

Wspaniale by było jakbyśmy mogli napisać:

```
project_class example
```

```
    : printable, comparable, json_serializable
```

```
{
```

```
    uint64_t id;
```

```
    uint64_t ts;
```

```
    std::string name;
```

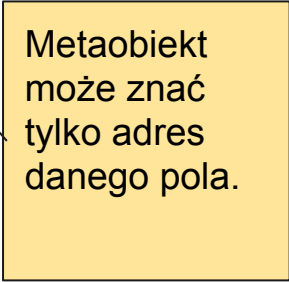
```
};
```

metaklasa printable

```
$class printable {  
    friend std::ostream& operator<<(std::ostream& os, const printable&  
that) {  
        for... (auto x : $reflect(prototype).member_variables()) {  
            os << that.*(x.pointer());  
        }  
        return os;  
    }  
};
```


metaklasa printable

```
$class printable {  
    friend std::ostream& operator<<(std::ostream& os, const printable&  
that) {  
        for... (auto x : $reflect(printable).member_variables()) {  
            os << that.*(x.pointer());  
        }  
        return os;  
    }  
};
```



Metaobiekt
może znać
tylko adres
danego pola.

metaklasa comparable

```
$class comparable {  
  bool operator==(const comparable& that) const {  
    for... (auto x : $reflect(prototype).member_variables())  
      if ((*this).*(x.pointer())) != that.*(x.pointer()))  
        return false;  
    return true;  
  }  
  bool operator!=(const comparable& that) const {  
    return !(*this == that);  
  }  
};
```

metaklasa json_serializable

```
$class json_serializable {
  std::string to_json() const {
    std::ostringstream output;
    output << "{";
    int members_count = $json_serializable.member_variables().size();
    int member_index = 1;
    for... (auto x : $reflect(prototype).member_variables()) {
      output << "\"" << x.name() << "\" : ";
      if constexpr(std::is_fundamental_v<x.type()>) {
        output << (*this).*(x.pointer());
      }
      /* MORE DISPATCHES ON TYPES */
      else {
        output << x.to_json();
      }
      if (member_index < members_count) {
        output << ", ";
        member_index++;
      }
    }
    output << "}";
    return output;
  }
};
```

metaklasa `project_class` - założenia

- Wersja minimum
- funkcje `get`, `set`, `ref` dla każdego pola
 - co jeżeli pole nie jest kopiowalne? - problem w oryginalnym `create_class`

metaklasa accessors

```
$class accessors {  
constexpr {  
  for... (auto m : $reflect(prototype).member_variables()) {  
    if constexpr (std::is_copy_constructible_v<m.type()>) {  
      -> { public:  
        (. m.type() .) (. "get_"s + m.name() .) () const {  
          return (. m.name() .);  
        }  
      }  
    }  
  }  
-> { public:  
  const (. m.type() .)& (. "ref_"s + m.name() .) () const {  
    return (. m.name() .);  
  }  
}
```

```
(. m.type() .)& (. "ref_"s + m.name() .)() {  
  return (. m.name() .)  
}  
void (. "set_"s + m.name() .) ((. m.type() .) value) {  
  (. m.name() .) = value;  
}  
}  
}};
```

metaklasa accessors

```
$class accessors {  
constexpr {  
  for... (auto m : $reflect(prototype).member_variables()) {  
    if constexpr (std::is_copy_constructible_v<m.type()>) {  
      -> { public:  
        (. m.type() .) (. "get_"s + m.name() .) () const {  
          return (. m.name() .);  
        }  
      }  
    }  
  }  
  -> { public:  
    const (. m.type() .)& (. "ref_"s + m.name() .) () const {  
      return (. m.name() .);  
    }  
  }  
}
```

```
(. m.type() .)& (. "ref_"s + m.name() .)() {  
  return (. m.name() .)  
}  
void (. "set_"s + m.name() .) ((. m.type() .) value) {  
  (. m.name() .) = value;  
}  
}  
};
```



metaklasa project_class

```
$class project_class : accessors, ordered, json_serializable {};
```

Użycie:

```
project_class example {
```

```
  uint64_t id;
```

```
  uint64_t ts;
```

```
  std::string name;
```

```
};
```

```
example my_foo {1, 2, "Jan"}, my_baz {1, 2, "Janek"};
```

```
bool cmp = my_foo < my_baz;
```

```
std::string json_data = my_foo.to_json(); // {"id" : 1, "ts" : 2, "name" : "Jan"}
```

W pochwalę metaklas

- Dzięki metaklasom intencja programisty jest jasna, czytający kod może się spodziewać co dana klasa robi
- Rozsądna możliwość debugowania
 - Mały błąd w pisaniu makr lub pomyłka w użyciu mogą skutkować totalnie nie pomocną diagnostyką
 - Jedyne sposoby to output preprocesora i patrzenie co jest nie tak
 - Diagnostyka jest przejęta przez kompilator - lepsze błędy diagnostyczne - `compiler.require`

W pochwalę metaklas

- Możliwość sprawdzenia własności typów
 - koncepty
- Dowolna transformacja typów (w tym całych klas)
- Warunkowa generacja kodu
 - Pola w klasie nie możemy skopiować? Nie generujemy gettera lub przerywamy kompilację z ładnym błędem

W pochwalę metaklas

- Możliwość testowania jak każdego innego kodu
 - Propozycja operatora `$my_class.is($other_class)`

W pochwalę metaklas

- Możliwość testowania jak każdego innego kodu
 - Propozycja operatora `$my_class.is($other_class)`
 - Wstrzyknięcie nowych zależności
 - Zwykła struktura X
 - W testach potrzebujemy ją wypisać na stderr
 - `using printable_X = $X.as(printable);`

W pochwalę metaklas

- newtype z innych języków dostępny za darmo
 - `$class newtype {};`
 - `using first_name = $std::string.as(newtype);`
 - `using last_name = $std::string.as(newtype);`
 - `void f(first_name fn, last_name ln);`

Długa droga przed metaklasami

Główna [\[Propozycja\]](#) metaklas bazuje na innych propozycjach:

- Refleksja [\[A design for static reflection\]](#) i inne
 - Introspekcja
 - Reifikacja
- Constexpr operator/block [\[The constexpr Operator\]](#) i inne
- Koncepty [\[C++ extensions for Concepts\]](#)
 - Niedługo w standardzie

Dziękuję za uwagę