Почти прикладная рефлексия

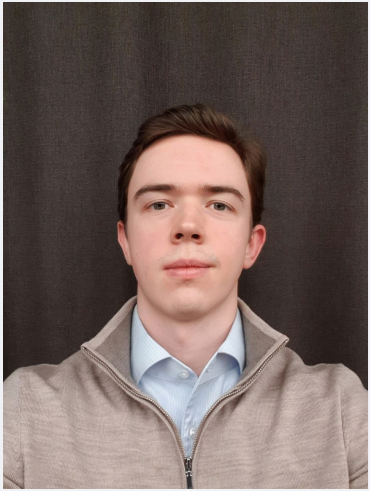
Формализуем паттерны программирования на C++26

C++ Russia 2026
Москва

Александр Романов

О себе

- Инженер в "Синтакор"
- Группа компиляторов и инструментов разработки
- Контрибутор
  - Генератор тестов llvm-snippy
  - Инфраструктура LLVM
  - Симулятор riscv-isa-sim



Что делает программа?

Что делает программа?

- Программа производит изменение над

Что делает программа?

- Программа производит изменение над
  - Данными - обычное программирование

Что делает программа?

- Программа производит изменение над
  - Данными - обычное программирование
  - Типами - метапрограммирование

Что делает программа?

- Программа производит изменение над
  - Данными - обычное программирование
  - Типами - метапрограммирование
    - SFINAE (специализация)

Что делает программа?

- Программа производит изменение над
  - Данными - обычное программирование
  - Типами - метапрограммирование
    - SFINAE (специализация)
  - Программами

Что делает программа?

- Программа производит изменение над
  - Данными - обычное программирование
  - Типами - метапрограммирование
    - SFINAE (специализация)
  - Программами
    - Макросы?

- Макросы позволяют вставить шаблонную строку в программу

```
1 # define ever (;;)
2
3 for ever {
4     hello();
5 }
```

О макросах

Генерация кода для **enum**

```
#define SHAPES                    \
 SHAPE_DECL(CIRCLE, circle)       \
 SHAPE_DECL(TRIANGLE, triangle)   \
 SHAPE_DECL(RECTANGLE, rectangle)

#define SHAPE_DECL(name, str)     \
  name,

enum ShapeKind {
  SHAPES
};
```

О макросах

■ Генерация кода для **enum**

```
#define SHAPES                    \
 SHAPE_DECL(CIRCLE, circle)       \
 SHAPE_DECL(TRIANGLE, triangle)   \
 SHAPE_DECL(RECTANGLE, rectangle)

#define SHAPE_DECL(name, str)     \
  name,

enum ShapeKind {
  SHAPES
};
```

```
// Somewhere else
# define SHAPE_DECL(name, str)     \
 case ShapeKind::name: return #str;

std::string to_string(ShapeKind K) {
  switch (K) {
    SHAPES
  }
  return "";
}
```

О макросах

- Генерация кода для **enum**

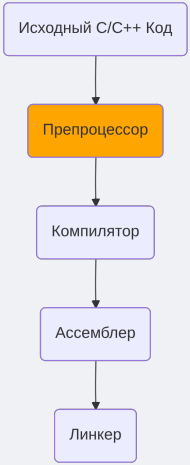
```
#define SHAPES                    \
 SHAPE_DECL(CIRCLE, circle)       \
 SHAPE_DECL(TRIANGLE, triangle)   \
 SHAPE_DECL(RECTANGLE, rectangle)

#define SHAPE_DECL(name, str)     \
  name,

enum ShapeKind {
  SHAPES
};
```

```
// Somewhere else
# define SHAPE_DECL(name, str)     \
 case ShapeKind::name: return #str;

std::string to_string(ShapeKind K) {
  switch (K) {
    SHAPES
  }
  return "";
}
```

- Научились конвертировать **enum** в строку
  - Но только для наших **enum**

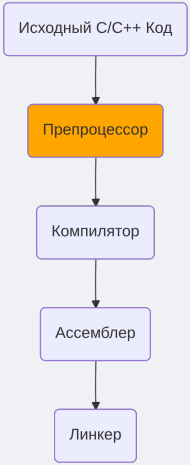
О макросах

- Не являются частью грамматики языка



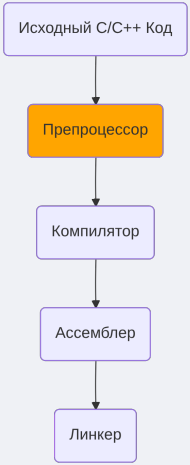
О макросах

- Не являются частью грамматики языка
- Работают над текстом программы, а не над программой



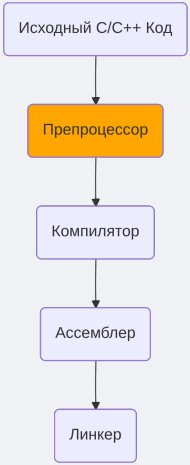
О макросах

- Не являются частью грамматики языка
- Работают над текстом программы, а не над программой
- Не гарантирует единственности вычислений



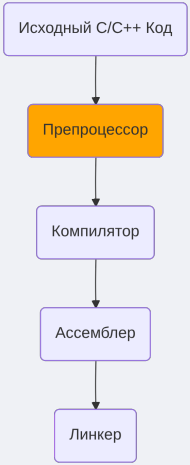
О макросах

- Не являются частью грамматики языка
- Работают над текстом программы, а не над программой
- Не гарантирует единственности вычислений
- Не гарантируют последовательности вычислений



О макросах

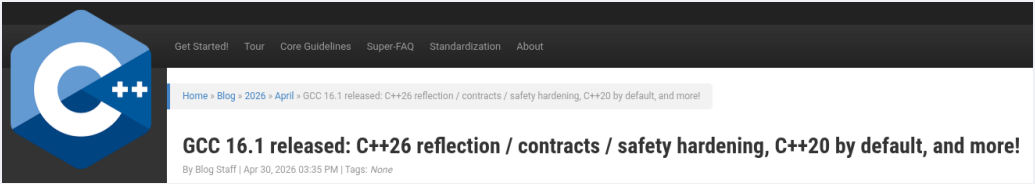
- Не являются частью грамматики языка
- Работают над текстом программы, а не над программой
- Не гарантирует единственности вычислений
- Не гарантируют последовательности вычислений
- Не позволяют сложной манипуляции



О рефлексии

Рефлексия - возможность манипулировать программой, которую вы пишете

- Reflection for C++26 **p2996** (*https://isocpp.org/files/papers/P2996R13.html*)
  - Уже в C++26
  - Поддержка
    - gcc-16 уже вышел
    - bloomberg/clang



О рефлексии

Язык

- **^^** - оператор кошачьих ушек
- **[::]** - сплайс оператор
- **template for** - expansion
- **[[=annotation]]** - аннотации

Библиотека

- **<meta>** library
- **std::meta** namespace

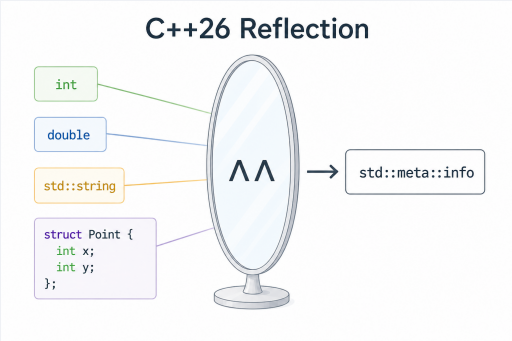
Быстрая рефлексия

operator ^^

Отображает сущности (типы, функции, пространства имён и т.д.) в единый непрозрачный тип **std::meta::info**

```
1 #include <meta>
2 struct S1 {};
3 template <typename T> struct S2 {};
4
5 std::meta::info int_info = ^^int;
6 std::meta::info stdns = ^^::std;
7 std::meta::info s1_refl = ^^S1;
8 std::meta::info s2int_refl = ^^S2<int>;
9 std::meta::info s2_refl = ^^S2;
```

https://godbolt.org/z/ash68nE77



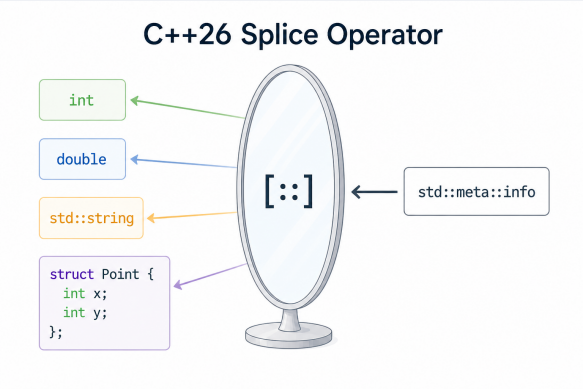
Быстрая рефлексия

splice оператор

Конвертирует **std::meta::info** обратно в привычные языковые конструкции

```cpp
#include <meta>

constexpr auto r = ^^int;
// Same as: int x = 42;
typename[:r:] x = 42;
// Same as: char c = '*';
typename[:^^char:] c = '*';
```

https://godbolt.org/z/ajGbbnb3E



Итерируемся по std::tuple

История: **boost::hana**

```
1 #include <boost/hana.hpp>
2
3 namespace hana = boost::hana;
4
5 auto tup = hana::make_tuple(0, 'a', 3.14);
6 hana::for_each(tup, [](auto elem) {
7   std::cout << elem << std::endl;
8 });
```

Итерируемся по std::tuple

История: **boost::hana**

```
1 #include <boost/hana.hpp>
2
3 namespace hana = boost::hana;
4
5 auto tup = hana::make_tuple(0, 'a', 3.14);
6 hana::for_each(tup, [](auto elem) {
7   std::cout << elem << std::endl;
8 });
```

👓 Раскрывается в:

```
1 auto tup = hana::make_tuple(0, 'a', 3.14);
2 auto func = [](auto elem) {
3   std::cout << elem << std::endl;
4 };
5 func(std::get<0>(tup));
6 func(std::get<1>(tup));
7 func(std::get<2>(tup));
```

Инстанцируем функцию для каждого элемента **std::tuple**

https://godbolt.org/z/aMGdYaPT6

Итерируемся по std::tuple

C++26 expansion statement

```
1 auto tup = std::make_tuple(0, 'a', 3.14);
2 template for (auto elem : tup) {
3   std::cout << elem << std::endl;
4 }
```

Итерируемся по std::tuple

■ C++26 expansion statement

```
1 auto tup = std::make_tuple(0, 'a', 3.14);
2 template for (auto elem : tup) {
3   std::cout << elem << std::endl;
4 }
```

■ 👓 Раскрывается в:

```
 1 auto tup = std::make_tuple(0, 'a', 3.14);
 2 auto &&[e0, e1, e2] = tup;
 3 {
 4   std::cout << e0 << std::endl;
 5 }
 6 {
 7   std::cout << e1 << std::endl;
 8 }
 9 {
10   std::cout << e2 << std::endl;
11 }
```

■ Повторяет (expand) тело цикла для каждого элемента

https://godbolt.org/z/18qEravrP

Итерируемся по структуре

C++26 expansion statement

```
1 struct Point {
2   int x, y;
3   std::string color;
4 };
5
6 Point p = {1, 2, "red"};
7 template for (auto elem : p) {
8   std::cout << elem << std::endl;
9 }
```

Итерируемся по структуре

C++26 expansion statement

```cpp
struct Point {
  int x, y;
  std::string color;
};

Point p = {1, 2, "red"};
template for (auto elem : p) {
  std::cout << elem << std::endl;
}
```

👓 Раскрывается в:

```cpp
Point p = {1, 2, "red"};
auto &&[e0, e1, e2] = p;
{
  std::cout << e0 << std::endl;
}
{
  std::cout << e1 << std::endl;
}
{
  std::cout << e2 << std::endl;
}
```

Итерируемся по структуре

C++26 expansion statement

```
1 struct Point {
2   int x, y;
3   std::string color;
4 };
5
6 Point p = {1, 2, "red"};
7 template for (auto elem : p) {
8   std::cout << elem << std::endl;
9 }
```

👓 Раскрывается в:

```
 1 Point p = {1, 2, "red"};
 2 auto &&[e0, e1, e2] = p;
 3 {
 4   std::cout << e0 << std::endl;
 5 }
 6 {
 7   std::cout << e1 << std::endl;
 8 }
 9 {
10   std::cout << e2 << std::endl;
11 }
```

Работает как structured binding и повторяет код для каждого элемента

https://godbolt.org/z/YdK5jqs7c

Быстрая рефлексия: expansion statement

Рефлексия встречает template for

```
1 struct Point {
2   int x, y;
3   std::string color;
4 };
5
6 Point p = {1, 2, 3};
7 constexpr auto ctx = std::meta::access_context::current();
8 constexpr auto members = std::meta::nonstatic_data_members_of(^^Point, ctx);
9 template for (constexpr auto m : members) {
10   auto type_name = std::meta::display_string_of(std::meta::type_of(m));
11   auto name = std::meta::identifier_of(m);
12   std::println("{} {} = {}", type_name, name, s.[:m:]);
13 }
```

Быстрая рефлексия: expansion statement

Рефлексия встречает template for

```
1 struct Point {
2   int x, y;
3   std::string color;
4 };
5
6 Point p = {1, 2, 3};
7 constexpr auto ctx = std::meta::access_context::current();
8 constexpr auto members = std::meta::nonstatic_data_members_of(^^Point, ctx);
9 template for (constexpr auto m : members) {
10  auto type_name = std::meta::display_string_of(std::meta::type_of(m));
11  auto name = std::meta::identifier_of(m);
12  std::println("{} {} = {}", type_name, name, s.[:m:]);
13 }
```

```
<source>:12:34: error: could not compute size of expansion
   12 | template for (constexpr auto m : members) {
      |                                  ^
```

https://godbolt.org/z/fnq7bEo4s

Быстрая рефлексия: expansion statement

- Сделаем контейнер заранее известного размера

```
1  struct Point {
2    int x, y;
3    std::string color;
4  };
5
6  Point p = {1, 2, 3};
7  constexpr auto ctx = std::meta::access_context::current();
8  constexpr auto members = std::define_static_array(
9                             std::meta::nonstatic_data_members_of(^^Point, ctx));
10 template for (constexpr auto m : members) {
11   auto type_name = std::meta::display_string_of(std::meta::type_of(m));
12   auto name = std::meta::identifier_of(m);
13   std::println("{} {} = {}", type_name, name, s.[:m:]);
14 }
```

Быстрая рефлексия: expansion statement

Сделаем контейнер заранее известного размера

```
1  struct Point {
2    int x, y;
3    std::string color;
4  };
5
6  Point p = {1, 2, 3};
7  constexpr auto ctx = std::meta::access_context::current();
8  constexpr auto members = std::define_static_array(
9                             std::meta::nonstatic_data_members_of(^^Point, ctx));
10 template for (constexpr auto m : members) {
11   auto type_name = std::meta::display_string_of(std::meta::type_of(m));
12   auto name = std::meta::identifier_of(m);
13   std::println("{} {} = {}", type_name, name, s.[:m:]);
14 }
```

```
int x = 1
int y = 2
std::string color = 3
```

https://godbolt.org/z/Yjesfj9b1

Быстрая рефлексия: expansion statement

- Используем алиас на пространство имён

```
1 namespace meta = std::meta;
```

Быстрая рефлексия: expansion statement

Используем алиас на пространство имён

```
namespace meta = std::meta;
```

```
constexpr auto ctx = meta::access_context::current();
constexpr auto members = std::define_static_array(
                           meta::nonstatic_data_members_of(^^S, ctx));
template for (constexpr auto m : members) {
  auto type_name = meta::display_string_of(meta::type_of(m));
  auto name = meta::identifier_of(m);
  std::println("{} {} = {}", type_name, name, s.[:m:]);
}
```

Быстрая рефлексия

Операции: **std::meta::substitute**

Подставляем шаблонные параметры в рефлексию шаблона

```
namespace std::meta {
template<reflection_range R>
  consteval info substitute(info templ, R&& arguments);
}
```

Быстрая рефлексия

■ Операции: **std::meta::substitute**

Подставляем шаблонные параметры в рефлексию шаблона

```
namespace std::meta {
template<reflection_range R>
  consteval info substitute(info templ, R&& arguments);
}
```

▩ Используем для **std::array**

```
constexpr auto arr_refl = ^^std::array;
constexpr auto size = meta::reflect_constant(5);
constexpr auto instantiated = meta::substitute(arr_refl, {^^int, size});
static_assert(std::is_same_v<typename[:instantiated:], std::array<int, 5>>);
```

https://godbolt.org/z/noxchf3r6

Быстрая рефлексия: expansion statement

Распечатаем все элементы **enum**

```
1 enum ShapeKind {
2   CIRCLE,
3   TRIANGLE,
4   RECTANGLE,
5 };
6
7 template for(constexpr auto e: meta::enumerators_of(^^ShapeKind)) {
8   std::println("{}", meta::identifier_of(e));
9 }
```

Быстрая рефлексия: expansion statement

■ Распечатаем все элементы **enum**

```
1 enum ShapeKind {
2   CIRCLE,
3   TRIANGLE,
4   RECTANGLE,
5 };
6
7 template for(constexpr auto e: meta::enumerators_of(^^ShapeKind)) {
8   std::println("{}", meta::identifier_of(e));
9 }
```

■ Проблема. Снова неизвестный размер

```
<source>:6:34: error: could not compute size of expansion
    6 |   template for(constexpr auto e: enumerators_of(^^ShapeKind)) {
      |                                  ^
```

https://godbolt.org/z/41T77ehah

Быстрая рефлексия: expansion statement

Снова используем **std::define_static_array**

```
enum ShapeKind {
  CIRCLE,
  TRIANGLE,
  RECTANGLE,
};

template for(constexpr auto e: std::define_static_array(
                                  meta::enumerators_of(^^ShapeKind))) {
  std::println("{}", meta::identifier_of(e));
}
```

```
CIRCLE
TRIANGLE
RECTANGLE
```

https://godbolt.org/z/dbMP14Mdq

Пример: enum to string

Напишем **to_string** без препроцессора

```
template<typename E>
constexpr std::string_view to_string(E value) {
  template for (constexpr auto e :
                std::define_static_array(meta::enumerators_of(^^E)))
    if (value == [:e:])
      return meta::identifier_of(e);
  return "<unknown>";
}
```

Пример: enum to string

Напишем **to_string** без препроцессора

```
template<typename E>
constexpr std::string_view to_string(E value) {
  template for (constexpr auto e :
                std::define_static_array(meta::enumerators_of(^^E)))
    if (value == [:e:])
      return meta::identifier_of(e);
  return "<unknown>";
}
```

```
std::println("{}", to_string(ShapeKind::CIRCLE));
std::println("{}", to_string(ShapeKind::TRIANGLE));
std::println("{}", to_string(ShapeKind::RECTANGLE));
```

```
CIRCLE
TRIANGLE
RECTANGLE
```

https://godbolt.org/z/4rnecssGx

Рефлексия - просто программа

- Мы можем проделать любые преобразования над именами

```
1 template<typename E>
2 constexpr std::string_view to_string_lowercase(E value) {
3   template for (constexpr auto e :
4                 std::define_static_array(meta::enumerators_of(^^E)))
5     if (value == [:e:])
6       return to_lowercase(meta::identifier_of(e));
7   return "<unknown>";
8 }
```

Рефлексия - просто программа

Мы можем проделать любые преобразования над именами

```
template<typename E>
constexpr std::string_view to_string_lowercase(E value) {
  template for (constexpr auto e :
                std::define_static_array(meta::enumerators_of(^^E)))
    if (value == [:e:])
      return to_lowercase(meta::identifier_of(e));
  return "<unknown>";
}
```

```
std::println("{}", to_string_lowercase(ShapeKind::CIRCLE));
std::println("{}", to_string_lowercase(ShapeKind::TRIANGLE));
std::println("{}", to_string_lowercase(ShapeKind::RECTANGLE));
```

```
circle
triangle
rectangle
```

https://godbolt.org/z/rhvTY7Te9

Архитектурная модель программы

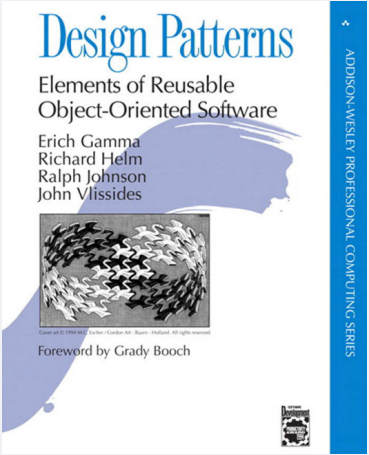
- Согласно GOF "Design Patterns"

Имя паттерна

Решаемая проблема

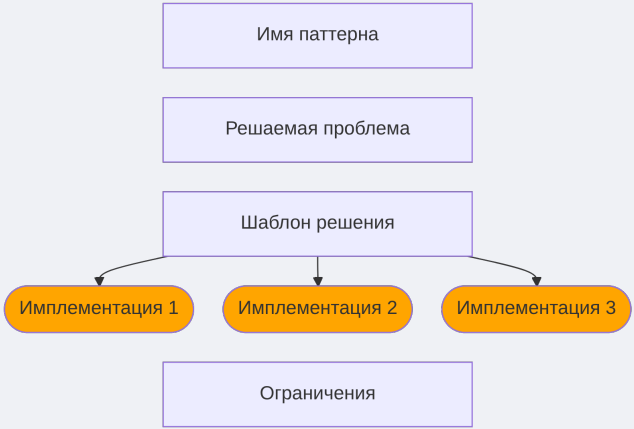
Шаблон решения

Ограничения



Архитектурная модель программы

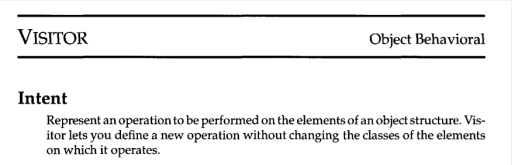
- Согласно GOF "Design Patterns"



Паттерная угадайка

Что такое визитор?

Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.



VISITOR — Object Behavioral

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

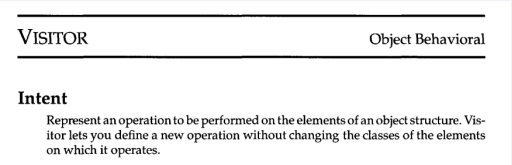
Рукописная иерархия

```
struct VisitorInterface;

struct NodeInterface {
  virtual ~NodeInterface() = 0;
  virtual void accept(VisitorInterface&);
};

struct IntNode : public NodeInterface {
  void accept(VisitorInterface& v) override {
    v.visit(*this);
  }
};

struct VisitorInterface {
  virtual ~VisitorInterface() = 0;
  virtual void visit(IntNode&) = 0;
  virtual void visit(FloatNode&) = 0;
};
```

Паттерная угадайка

Что такое визитор?

Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.



VISITOR — Object Behavioral

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

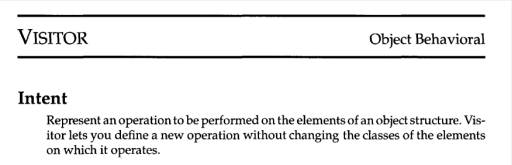
Используем **std::visit**

```
struct DrawVisitor {
  void operator()(Circle &);
  void operator()(Triangle &);
};

std::variant<Circle, Triangle> Shape = f();
DrawVisitor Draw;
std::visit(Draw, Shape);
```

Паттерная угадайка

Что такое визитор?

Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.



VISITOR — Object Behavioral

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Используем **std::visit**

```
struct DrawVisitor {
  void operator()(Circle &);
  void operator()(Triangle &);
};

std::variant<Circle, Triangle> Shape = f();
DrawVisitor Draw;
std::visit(Draw, Shape);
```

Оба удовлетворяют определению Visitor

Проблема Переиспользования Паттернов

factory для иерархии **Shapes**

```
struct ShapeFactory {
 std::unique_ptr<Shape>
             create(std::string_view name) {
    if (name == "Circle") {
      do_something();
      return new Circle();
    }
    if (name == "Triangle") {
      do_something();
      return new Triangle();
    }
    return nullptr;
  }
};
```

Проблема Переиспользования Паттернов

factory для иерархии **Shapes**

```
struct ShapeFactory {
 std::unique_ptr<Shape>
             create(std::string_view name) {
    if (name == "Circle") {
      do_something();
      return new Circle();
    }
    if (name == "Triangle") {
      do_something();
      return new Triangle();
    }
    return nullptr;
  }
};
```

factory для иерархии **ASTNode**

```
struct NodeFactory {
  std::unique_ptr<BaseNode>
              create(std::string_view name) {
    if (name == "IntNode") {
      do_something();
      return new IntNode();
    }
    if (name == "AddNode") {
      do_something();
      return new AddNode();
    }
    return nullptr;
  }
};
```

Проблема Переиспользования Паттернов

▉ factory для иерархии **Shapes**

```
struct ShapeFactory {
 std::unique_ptr<Shape>
             create(std::string_view name) {
    if (name == "Circle") {
      do_something();
      return new Circle();
    }
    if (name == "Triangle") {
      do_something();
      return new Triangle();
    }
    return nullptr;
  }
};
```

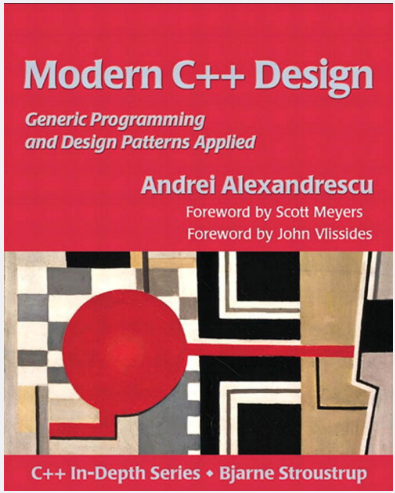
▉ factory для иерархии **ASTNode**

```
struct NodeFactory {
  std::unique_ptr<BaseNode>
             create(std::string_view name) {
    if (name == "IntNode") {
      do_something();
      return new IntNode();
    }
    if (name == "AddNode") {
      do_something();
      return new AddNode();
    }
    return nullptr;
  }
};
```

▓ Дуплицируем один и тот же код для разных иерархий

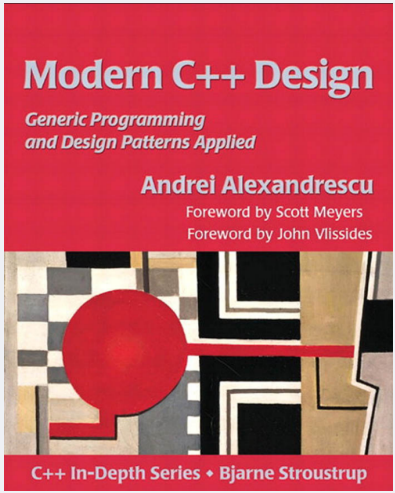
Решения

- "Modern C++ Design" A. Alexandrescu (2001)
- Boost
  - Boost.Signals2
  - Boost.TypeErasure
  - Boost.Factory
  - Boost.Flyweight



Решения

- "Modern C++ Design" A. Alexandrescu (2001)
- Boost
  - Boost.Signals2
  - Boost.TypeErasure
  - Boost.Factory
  - Boost.Flyweight
- Рефлексия C++26



Factory

Factory

Типичная реализация

```
struct NodeFactory {
  std::unique_ptr<BaseNode> create(std::string_view name) {
    if (name == "IntNode") {
      do_something();
      return std::make_unique<IntNode>();
    }
    if (name == "FloatNode") {
      do_something();
      return std::make_unique<FloatNode>();
    }
    ....
    return nullptr;
  }
};

NodeFactory f;
auto node1 = f.create("IntNode");
auto node2 = f.create("FloatNode");
```

Factory

Обобщаем решение

Хотим переиспользования между иерархиями. Для этого сделаем **Factory** шаблоном, параметризуем базовым классом и пачкой наследников.

```
1 template <typename BaseT, typename ...Derived>
2 struct Factory {
3   std::unique_ptr<BaseT> create (str::string_view name) {
4     // Как будем искать нужный тип?
5   }
6 };
```

Factory

Обобщаем решение

Хотим переиспользования между иерархиями. Для этого сделаем **Factory** шаблоном, параметризуем базовым классом и пачкой наследников.

```
1 template <typename BaseT, typename ...Derived>
2 struct Factory {
3   std::unique_ptr<BaseT> create (str::string_view name) {
4     // Как будем искать нужный тип?
5   }
6 };
```

Получим имя типа при помощи рефлексии

```
struct MyClass;
static_assert(meta::identifier_of(^^MyClass) == "MyClass");
static_assert(meta::identifier_of(^^std::string) == "string");
```

https://godbolt.org/z/dzqYacKoT

Factory

Итерируемся: expansion

Пройдёмся по всем наследникам из пачки при помощи **template for**

```
1 template <typename BaseT, typename ...Derived>
2 struct Factory {
3   std::unique_ptr<BaseT> create (str::string_view name) {
4     template for(constexpr auto d: {^^Derived...}) {
5
6     }
7   }
8 };
```

Factory

- Поиск нужного типа по его имени

Снова используем **meta::identifier_of**

```
template <typename BaseT, typename ...Derived>
struct Factory {
  std::unique_ptr<BaseT> create (str::string_view name) {
    template for(constexpr auto d: {^^Derived...}) {
      if (name == meta::identifier_of(d)) {

      }
    }
  }
};
```

Factory

Конструирование

Создаём объект по найденному типу. Используем splice operator.

```
template <typename BaseT, typename ...Derived>
struct Factory {
  std::unique_ptr<BaseT> create (str::string_view name) {
    template for(constexpr auto d: {^^Derived...}) {
      if (name == meta::identifier_of(d)) {
        return std::make_unique<[:d:]>();
      }
    }
  }
};
```

Factory

Конструирование

Создаём объект по найденному типу. Используем splice operator.

```
 1 template <typename BaseT, typename ...Derived>
 2 struct Factory {
 3   std::unique_ptr<BaseT> create (str::string_view name) {
 4     template for(constexpr auto d: {^^Derived...}) {
 5       if (name == meta::identifier_of(d)) {
 6         return std::make_unique<[:d:]>();
 7       }
 8     }
 9   }
10 };
```

```
<source>:8:38: error: unparenthesized splice expression cannot be used as a template argument
    8 |         return std::make_unique<[:d:]>();
      |                                      ^
```

https://godbolt.org/z/9cMzvonzr

Factory

Исправляем ошибки

Попробуем обернуть в скобки, как и просил компилятор

```
 1 template <typename BaseT, typename ...Derived>
 2 struct Factory {
 3   std::unique_ptr<BaseT> create (str::string_view name) {
 4     template for(constexpr auto d: {^^Derived...}) {
 5       if (name == meta::identifier_of(d)) {
 6         return std::make_unique<([:d:])>();
 7       }
 8     }
 9   }
10 };
```

Factory

Исправляем ошибки

Попробуем обернуть в скобки, как и просил компилятор

```
 1 template <typename BaseT, typename ...Derived>
 2 struct Factory {
 3   std::unique_ptr<BaseT> create (str::string_view name) {
 4     template for(constexpr auto d: {^^Derived...}) {
 5       if (name == meta::identifier_of(d)) {
 6         return std::make_unique<([:d:])>();
 7       }
 8     }
 9   }
10 };
```

```
<source>:8:34: error: reflection not usable in a splice expression
    8 |         return std::make_unique<([:d:])>();
      |                                  ^~~~~
```

https://godbolt.org/z/x6oz445Kh

Factory

Исправляем ошибки

На самом деле проблема заключалась в том, что компилятор воспринял наш сплайс как объект, а не тип. Решаем как и в любом шаблонном коде - вставляем слово **typename**.

```
 1 template <typename BaseT, typename ...Derived>
 2 struct Factory {
 3   std::unique_ptr<BaseT> create (str::string_view name) {
 4     template for(constexpr auto d: {^^Derived...}) {
 5       if (name == meta::identifier_of(d)) {
 6         return std::make_unique<typename[:d:]>();
 7       }
 8     }
 9   }
10 };
```

Работает! 👍

Factory

Полезная работа

Скорее всего мы делали factory не просто так. Добавляем полезную нагрузку.

```
template <typename BaseT, typename ...Derived>
struct Factory {
  std::unique_ptr<BaseT> create (str::string_view name) {
    template for(constexpr auto d: {^^Derived...}) {
      if (name == meta::identifier_of(d)) {
        auto res = std::make_unique<typename[:d:]>();
        log_created(res.get());
        return std::move(res);
      }
    }
    return nullptr;
  }
};
```

Factory

Полный пример

```
template <typename BaseT, typename ...Derived>
struct Factory {
  Factory(logger l) : log(std::move(l)) {}
  std::unique_ptr<BaseT> create (str::string_view name) {
    template for(constexpr auto d: {^^Derived...}) {
      if (name == meta::identifier_of(d)) {
        auto res = std::make_unique<typename[:d:]>();
        log_created(res.get());
        return std::move(res);
      }
    }
    return nullptr;
  }
};

Factory<app::BaseNode, app::IntNode, app::FloatNode> f(logger{});
auto node1 = f.create("IntNode");
auto node2 = f.create("FloatNode");
```

https://godbolt.org/z/ddo5dezcn

Factory

Переиспользуемое решение

Теперь для написания factory для другой иерархии не нужно производить никакой работы. Мы успешно сделали обобщили factory.

```
std::vector<std::unique_ptr<BaseNode>> Nodes;
Factory<BaseNode, IntNode, FloatNode, AddNode> NodeFactory;
Nodes.emplace_back(NodeFactory.create("IntNode"));
Nodes.emplace_back(NodeFactory.create("FloatNode"));
Nodes.emplace_back(NodeFactory.create("AddNode"));

std::vector<std::unique_ptr<Shape>> Shapes;
Factory<Shape, Circle, Triangle, Rectangle> ShapeFactory;
Shapes.emplace_back(ShapeFactory.create("Circle"));
Shapes.emplace_back(ShapeFactory.create("Rectangle"));
Shapes.emplace_back(ShapeFactory.create("Triangle"));
```

Observer

Observer

Обычная реализация

Пишем простую обёртку, логирующую на присваивании своих полей.

```
 1 struct Widget {
 2   unsigned x = 0;
 3   unsigned y = 0;
 4   std::string message = "Hello";
 5 };
 6
 7 observable_widget ow (Widget{});
 8 ow.subscribe(
 9   [](auto event) {
10      std::println("'{}' changed", event.name);
11   });
12 ow.message = "Hello, friends!";
13 ow.x = 3;
```

Observer

Обычная реализация

Пишем простую обёртку, логирующую на присваивании своих полей.

```
 1 struct Widget {
 2   unsigned x = 0;
 3   unsigned y = 0;
 4   std::string message = "Hello";
 5 };
 6
 7 observable_widget ow (Widget{});
 8 ow.subscribe(
 9   [](auto event) {
10      std::println("'{}' changed", event.name);
11   });
12 ow.message = "Hello, friends!";
13 ow.x = 3;
```

```
'message' changed
'x' changed
```

Observer

Обычная реализация

```
using observer_func = std::function<void(std::string_view)>;
using observer_list = std::vector<observer_func>;
template <typename T> struct observable_field final {
  T impl{};
  std::string_view name{};
  observer_list *observers = nullptr;
  observable_field &operator=(const T &v) {
    impl = v;
    for (auto &o : *observers)
      o(event{name});
    return *this;
  }
  operator const T &() const { return impl; }
};
```

Observer

Обычная реализация

```
class observable_widget {
  observer_list observers;
public:
  observable_field<int> x;
  observable_field<int> y;
  observable_field<std::string> message;

  observable_widget(const Widget &w);

  template <typename F>
  void subscribe(F f) {
    observers.emplace_back(f);
  }
};
```

Observer

Обычная реализация

```
class observable_widget {
  observer_list observers;
public:
  observable_field<int> x;
  observable_field<int> y;
  observable_field<std::string> message;

  observable_widget(const Widget &w);

  template <typename F>
  void subscribe(F f) {
    observers.emplace_back(f);
  }
};
```

Используем

```
observable_widget ow (Widget{});
ow.subscribe(
  [](auto name) {
     std::println("'{}' changed", name);
  });
ow.message = "Hello, friends!";
```

Observer

Обычная реализация

```
class observable_widget {
  observer_list observers;
public:
  observable_field<int> x;
  observable_field<int> y;
  observable_field<std::string> message;

  observable_widget(const Widget &w);

  template <typename F>
  void subscribe(F f) {
    observers.emplace_back(f);
  }
};
```

Используем

```
observable_widget ow (Widget{});
ow.subscribe(
  [](auto name) {
     std::println("'{}' changed", name);
  });
ow.message = "Hello, friends!";
```

```
'message' changed
```

Observer

- Обычная реализация

```
observable_widget::observable_widget(const Widget &w) {
  x.impl = w.x;
  x.observers = &observers;
  x.name = "x";
  y.impl = w.y;
  y.observers = &observers;
  y.name = "y";
  message.impl = w.message;
  message.observers = &observers;
  message.name = "message";
}
```

- Много ручной работы
- Каждый тип приходится оборачивать заново

Observer

- Используем рефлексию для создания обобщённого **observable**

Генерируем специализацию **observable_impl** при помощи рефлексии

C++26 позволяет определять агрегаты.

```
1 template <typename T> struct observable_impl;
2
3 consteval meta::info make_class_observable(meta::info tp) {
4   return meta::define_aggregate(meta::substitute(^^observable_impl, {tp}),
5                                 get_members_spec(tp));
6 }
```

Этот код генерирует нам специализацию **observable_impl** для типа своего аргумента. У специализации будут такие поля, которые мы захотим в **get_members_spec**.

Observer

Определяем члены класса

```
consteval auto get_members_spec(meta::info type) { // type == ^^Widget
  std::vector<meta::info> members;
  auto ctx = meta::access_context::unchecked();
  for (auto m : meta::nonstatic_data_members_of(type, ctx)) {
    // Получили observable_field<int> для `int x`
    auto field = meta::substitute(^^observable_field, {meta::type_of(m)});
    auto name = meta::identifier_of(m);
    members.push_back(meta::data_member_spec(field, {.name = name}));
  }
  return members;
}
```

Observer

Определяем члены класса

```
consteval auto get_members_spec(meta::info type) { // type == ^^Widget
  std::vector<meta::info> members;
  auto ctx = meta::access_context::unchecked();
  for (auto m : meta::nonstatic_data_members_of(type, ctx)) {
    // Получили observable_field<int> для `int x`
    auto field = meta::substitute(^^observable_field, {meta::type_of(m)});
    auto name = meta::identifier_of(m);
    members.push_back(meta::data_member_spec(field, {.name = name}));
  }
  return members;
}
```

Было

```
struct Widget {
  int x = 0;
  int y = 0;
  std::string message = "Hello";
};
```

Стало

```
template<> struct observable_impl<Widget> {
  observable_field<int> x;
  observable_field<int> y;
  observable_field<std::string> message;
};
```

Observer

Пишем интерфейсный observable

```
1 template <is_class T>
2 struct observable final : public observable_impl<T> {
3 private:
4   observer_list observers;
5 public:
6   observable(T v) {
7   // ...
8   }
9 };
```

Observer

- Делаем observable шаблоном

```
template <is_class T>
struct observable final : public observable_impl<T> {
private:
  observer_list observers;
public:
  observable(T v) {
    constexpr auto ctx = meta::access_context::unchecked();
    template for (constexpr auto dm :
                  std::define_static_array(
                    meta::nonstatic_data_members_of(^^T, ctx))) {
      auto member_name = meta::identifier_of(dm);
      auto &member = this->[:member_named(^^observable_impl<T>, member_name):];
      member.impl = v.[:dm:];
      member.observers = &observers;
      member.name = meta::identifier_of(dm);
    }
  }
};
```

Observer

Поиск члена класса по имени

```
consteval meta::info member_named(meta::info obj, std::string_view name) {
  auto ctx = meta::access_context::unchecked();
  for (auto m : meta::nonstatic_data_members_of(obj, ctx)) {
    if (meta::identifier_of(m) == name) {
      return m;
    }
  }
}
```

Если не нашли, то будет ошибка компиляции, т.к. мы ничего не вернули из функции.

Observer

- Поиск члена класса по имени

```
consteval meta::info member_named(meta::info obj, std::string_view name) {
  auto ctx = meta::access_context::unchecked();
  for (auto m : meta::nonstatic_data_members_of(obj, ctx)) {
    if (meta::identifier_of(m) == name) {
      return m;
    }
  }
}
```

Если не нашли, то будет ошибка компиляции, т.к. мы ничего не вернули из функции.

- Что это за алгоритм?

Observer

Поиск члена класса по имени

```
consteval meta::info member_named(meta::info obj, std::string_view name) {
  auto ctx = meta::access_context::unchecked();
  for (auto m : meta::nonstatic_data_members_of(obj, ctx)) {
    if (meta::identifier_of(m) == name) {
      return m;
    }
  }
}
```

Если не нашли, то будет ошибка компиляции, т.к. мы ничего не вернули из функции.

Что это за алгоритм?

Конечно же это **find_if**

Observer

Поиск члена класса по имени

Используем привычные алгоритмы

```
1 consteval meta::info member_named(meta::info obj, std::string_view name) {
2   auto ctx = meta::access_context::unchecked();
3   auto members = meta::nonstatic_data_members_of(obj, ctx);
4   auto found = ranges::find_if(members, [&](auto m) {
5     return meta::identifier_of(m) == name;
6   });
7   if (found != members.end()) return *found;
8 }
```

https://godbolt.org/z/o63jvbWMY

Observer

- Переиспользуемое решение

```
auto w = observable<Widget>(Widget{});
w.subscribe(
    [](auto event) {
        std::println("field \"{}\" changed", event.field_name);
    });
w.x = 42; // prints 'field "x" changed'
w.message = "hello"; // prints 'field "message" changed'

auto p = observable<std::pair<int, int>>({0, 0});
p.subscribe(
    [](auto event) {
        std::println("field \"{}\" changed", event.field_name);
    });

p.first = -1; // prints 'field "first" changed'
p.second = 3; // prints 'field "second" changed'
```

https://godbolt.org/z/Y1TMehErh

Стирание типов

Стирание типов

Типичная реализация

```
struct Circle {
  Point center;
  unsigned radius;
  void print() const;
};

struct Triangle {
  Point a, b, c;
  void print() const;
};


std::vector<Widget> Document;
Document.emplace_back(Circle{});
Document.emplace_back(Triangle{});
for (auto &w: Document)
  w.print();
```

Задача

Хотим сложить несвязные классы в один контейнер, если для них существует **print**

Стирание типов

Типичная реализация

https://godbolt.org/z/hePo459vq

```
class Widget {
  struct Concept {
    virtual ~Concept() = default;
    virtual void print() const = 0;
  };

  template <typename T>
  struct Model : public Concept {
    T impl;
    void print() const override { impl.print(); }
  };

  // или просто std::unique_ptr<Concept>
  std::polymorphic<Concept> Impl;
public:
  template<typename T>
  Widget(T val) : Impl(Model<T>(val)) {}

  void print() const { Impl.print(); }
};
```

Стирание типов

- Нельзя переиспользовать

Если хотим обобщить не по **print**, а по другому признаку, то всё придётся писать заново

Стирание типов

- Нельзя переиспользовать

Если хотим обобщить не по **print**, а по другому признаку, то всё придётся писать заново

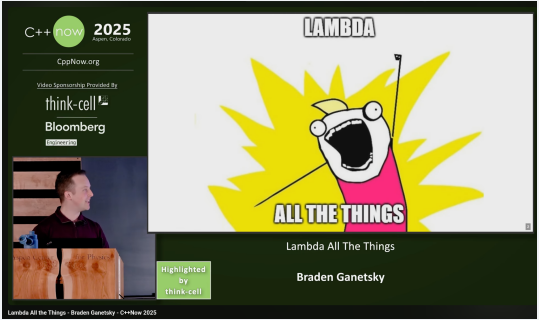
- Используем рефлексию для создания **Concept** и **Model<T>**
  - Проблема: невозможно создавать методы

```
template<typename T>
struct Model : public Concept {
  T val;
  void print() const override {....} // C++ 26 не позволяет добавлять методы
};
```

Стирание типов

Lambda All The Things!

```
1 struct Model {
2   T val;
3   std::function<void(void)> print = [&val]() { ns::print(val); };
4 };
```



Стирание типов

Снова **meta::define_aggregate**

```
template <typename T> struct erased_impl;

consteval meta::info make_class_erasable(meta::info tp) {
  return meta::define_aggregate(meta::substitute(^^erased_impl, {tp}),
                                get_members_spec(tp));
}

consteval auto get_members_spec(meta::info type) {
  std::vector<meta::info> members;
  for (auto m : get_member_functions(type)) {
    auto name = meta::identifier_of(m);
    auto inst = meta::substitute(^^std::function, {meta::type_of(m)});
    // `void print()` => `std::function<void(void)>`
    members.push_back(meta::data_member_spec(inst, {.name = name}));
  }
  return members;
}
```

Стирание типов

- Как хранить само значение?

  - **std::any**

```
struct erased_impl {
  std::any val;
  std::function<void(void)> print = [&val]() { val.print(); }; // Псевдокод
  ....
};
```

Стираем тип при помощи **std::any** и делегируем правильным методам

Стирание типов

- Как хранить само значение?

std::any

```
consteval auto get_members_spec(meta::info type) {
  std::vector<meta::info> members;
  members.push_back(meta::data_member_spec(^^std::any, {.name = "impl"}));
  for (auto m : get_member_functions(type)) {
    auto tp = meta::type_of(m);
    auto name = meta::identifier_of(m);
    auto inst = meta::substitute(^^std::function, {tp});
    members.push_back(meta::data_member_spec(inst, {.name = name}));
  }
  return members;
}
```

Стирание типов

Как хранить само значение?

std::any

```
consteval auto get_members_spec(meta::info type) {
  std::vector<meta::info> members;
  members.push_back(meta::data_member_spec(^^std::any, {.name = "impl"}));
  for (auto m : get_member_functions(type)) {
    auto tp = meta::type_of(m);
    auto name = meta::identifier_of(m);
    auto inst = meta::substitute(^^std::function, {tp});
    members.push_back(meta::data_member_spec(inst, {.name = name}));
  }
  return members;
}
```

Было

```
struct Circle {
  void print() const;
};
```

Стало

```
template<> struct erased_impl<Printable> {
  std::any impl;
  std::function<void(void)> print;
};
```

Стирание типов

Инициализация

```
template <typename ConceptType>
struct type_erased : public erased_impl<ConceptType> {
  using base_t = erased_impl<ConceptType>;

  template <typename T>
  type_erased(T t) : base_t{t} {
    // ...
  }
};
```

Наш концепт

```
struct PrintableConcept {
  void print() const; // Определение не нужно
};
```

Стирание типов

Инициализация

Присваиваем **std::function** делегирующие лямбды

```
using base_t = erased_impl<ConceptType>;

template <typename T>
type_erased(T t) : base_t{t} {
    T *value = std::any_cast<T>(&this->impl);
    constexpr auto methods = get_member_functions(^^ConceptType);
    template for (constexpr auto m : methods) {
        constexpr auto name = meta::identifier_of(m); // например "print"
        // Поиск std::function с именем "print"
        constexpr auto member = member_named(^^base_t, name);
        this->[:member:] = [value](auto&...ts) {
            // Поиск настоящего метода в конкретном классе
            constexpr auto func = member_named(^^T, name);
            return value->[:func:](ts...);
        };
    }
}
```

Стирание типов

Используем

```
1  struct Circle {
2    void print() const { std::println("Circle");}
3  };
4  struct Triangle {
5    void print() const { std::println("Triangle"); }
6  };
7  struct PrintableInterface {
8    void print() const; // no definition required
9  };
10
11 std::vector<type_erased<PrintableInterface>> document;
12 document.push_back(Circle{});
13 document.push_back(Triangle{});
14 for (auto &w: document)
15   w.print()
```

Стирание типов

Используем

```
1  struct Circle {
2    void print() const { std::println("Circle");}
3  };
4  struct Triangle {
5    void print() const { std::println("Triangle"); }
6  };
7  struct PrintableInterface {
8    void print() const; // no definition required
9  };
10
11 std::vector<type_erased<PrintableInterface>> document;
12 document.push_back(Circle{});
13 document.push_back(Triangle{});
14 for (auto &w: document)
15   w.print()
```

```
Circle
Triangle
```

https://godbolt.org/z/6W8TjxbKs

Стирание типов

Переиспользуемое решение

```
struct logger_interface {
  bool log(std::string_view);
};

struct stdout_logger {
  bool log(std::string_view) { ... }
};
struct file_logger {
  bool log(std::string_view) { ... }
};

std::vector<type_erased<logger_interface>> loggers;
loggers.push_back(stdout_logger{});
loggers.push_back(file_logger{"file1.log"});
loggers.push_back(somelib::file_logger{"file2.log"});

for (auto &e: Vec)
  e.empty();
```

Выводы

- Паттерны проектирования тяжело переиспользовать
- Переиспользование - вечная проблема
  - Андрей Александреску, boost и другие пытались её решить
- Рефлексия C++ позволяет написать обобщённую имплементацию
  - Я показал малую долю возможностей

Выводы

Статус

bloomberg/clang-p2996

- Ведётся работа по вливу в основной clang

gcc-16

- Уже включает в себя поддержку рефлексии

Выводы

Статус

bloomberg/clang-p2996

- Ведётся работа по вливу в основной clang

gcc-16

- Уже включает в себя поддержку рефлексии

Опции

- **-std=c++26 -freflection**

Конец