Обзор C++26

Обо мне

- Пишу на C++ больше 15 лет.
- Основал WG21 Russia в 2016 вместе с @apolukhin.
- В 2016-2019 представлял предложения от РФ в комитете.
- Руководил разработкой поискового движка в Яндексе.
- Руководил инфраструктурой, поиском и ML в Озоне.

Этот доклад

1. Немного о том, что уже есть в C++26.
2. И о том, что мы на самом деле ждем от C++26.
3. Кратко о Pattern Matching.
4. Кратко о Рефлексии.
5. Много кода.
6. Много английских терминов и англицизмов.

Я прочитал обсуждаемые предложения в стандарт и присутствовал на обсуждениях в комитете, чтобы вам не пришлось этого делать.

Новости со встречи в Токио

- Чтение неинициализированной памяти теперь не является "неопределенным поведением" (UB). Появилось понятие "ошибочного поведения" (erroneous behavior).
- Началось обсуждение политик комитета.
  - Одобрили политику о `[[nodiscard]]` – не использовать `[[nodiscard]]` в стандарте.
  - Начали обсуждение политики о `noexcept`.
- `std::string + std::string_view`.
- `= delete("should have a reason");`
- ...и еще немного по мелочи.

Что уже есть в C++26

- `std::function_ref` и `std::copyable_function`.
- `pack…[indexing]`.
- `_` для неиспользуемых переменных.
- Библиотека линейной алгебры на основе BLAS.
- Поддержка идиомы RCU.
- `is_debugger_present()` и `breakpoint()`.
- …и еще немного по мелочи.

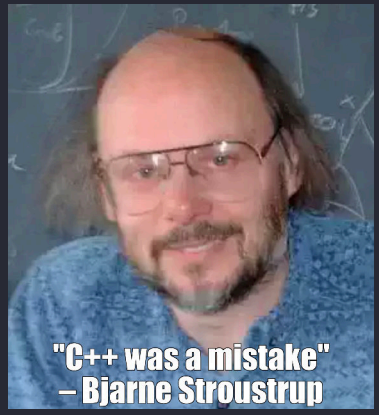
Что мы на самом деле ждем от C++26

	Статус	Консервативная оценка	Оптимистичная оценка
Senders	Рассмотрено в LWG	C++26	C++26
Networking	Зависит от Senders	C++29	C++26
Линейная алгебра	Замержено!	C++26	C++26
SIMD	Направлено в LWG	C++26	C++26
Контракты	Рассмотрено в SG21	C++29	C++26
Рефлексия	Направлено в EWG	C++26	C++26
Pattern Matching	Рассматривается в EWG	C++29	C++26

Что нужно обычным программистам

Наибольшую пользу для сообщества принесет pattern matching. Я проводил эксперименты с этим 15 лет назад. Мы не можем ожидать, что pattern matching просто появится в C++, люди должны работать над этим. И я хотел бы увидеть завершение работы над senders, это тянется уже три стандарта подряд. И после этого, возможно, статическую рефлексию.

— Бьёрн Страуструп,
последнее заседание комитета.



Pattern matching

Pattern matching: зачем?

```cpp
using Expression = std::variant<AddNode, MulNode, double>;

double evaluate(const Expression &e) {
    return std::visit<double>([]<class T>(const T &arg) {
        if constexpr (std::is_same_v<T, AddNode>)
            return evaluate(*arg.l) + evaluate(*arg.r);
        else if constexpr (std::is_same_v<T, MulNode>)
            return evaluate(*arg.l) * evaluate(*arg.r);
        else if constexpr (std::is_same_v<T, double>)
            return arg;
        else
            static_assert(always_false_v<T>, "Non-exhaustive visitor!");
    }, e);
}
```

```cpp
struct AddNode {
    std::unique_ptr<Expression> l, r;
};
struct MulNode {
    std::unique_ptr<Expression> l, r;
};
```

Pattern matching: зачем?

```cpp
using Expression = std::variant<AddNode, MulNode, double>;

double evaluate(const Expression &e) {
    double value = 0.0;
    if (std::holds_alternative<AddNode>(e))
        value = evaluate(*std::get<AddNode>(e).l) + evaluate(*std::get<AddNode>(e).r);
    else if (std::holds_alternative<MulNode>(e))
        value = evaluate(*std::get<MulNode>(e).l) * evaluate(*std::get<MulNode>(e).r);
    else if (std::holds_alternative<double>(e))
        value = std::get<double>(e);
    else
        assert(false && "eeeh?");
}
```

C++ — ужасный язык.

— Линус Торвальдс, рассылка git.

Pattern matching: `std::variant`

```
1   using Expression = std::variant<AddNode, MulNode, double>;
2
3   double evaluate(const Expression &e) {
4       return e match {
5           AddNode: let arg ⇒ evaluate(*arg.l) + evaluate(*arg.r);
6           MulNode: let arg ⇒ evaluate(*arg.l) * evaluate(*arg.r);
7           double:  let arg ⇒ arg;
8           _                ⇒ throw EvaluationException(); // e.valueless_by_exception().
9       };
10  }
```

- Матчи вычисляются по порядку.
- `let` используется для объявления binding'ов.
- После ⇒ можно использовать только выражения.
- Если матч не был найден, то вызывается `std::terminate()`.

Pattern matching: `std::any`

```
1   using Expression = std::any;
2
3   double evaluate(const Expression &e) {
4       return e match {
5           AddNode: let arg ⇒ evaluate(arg.l) + evaluate(arg.r);
6           MulNode: let arg ⇒ evaluate(arg.l) * evaluate(arg.r);
7           double:  let arg ⇒ arg;
8           _                ⇒ throw EvaluationException(); // All other types.
9       };
10  }
```

Pattern matching: `std::any`

```
1   using Expression = std::any;
2
3   double evaluate(const Expression &e) {
4       return e match {
5           AddNode: let arg ⇒ evaluate(arg.l) + evaluate(arg.r);
6           MulNode: let arg ⇒ evaluate(arg.l) * evaluate(arg.r);
7           double:  let arg ⇒ arg;
8           _                ⇒ throw EvaluationException(); // All other types.
9       };
10  }
```

Вам не нужно беспокоиться о том, что не важно. В этом красота C++. Он позволяет вам сосредоточиться на том, что вы считаете важным.

— Бьёрн Страуструп, как нагаллюцинировал ChatGPT.

Pattern matching: как?

```
1   struct Expression : std::variant<AddNode, MulNode, double> {};
2
3   double evaluate(const Expression &e) {
4       // return e match {
5       switch (e.index()) {
6       //      AddNode: let arg ⇒ evaluate(*arg.l) + evaluate(*arg.r);
7           case 0: {
8               auto &&arg = get<0>(e);
9               return evaluate(*arg.l) + evaluate(*arg.r);
10          }
11      //      ...
12      //      double:  let arg ⇒ arg;
13          case 2: {
14              auto &&arg = get<2>(e);
15              return arg;
16          }
17      //      _                ⇒ throw EvaluationException(); // e.valueless_by_exception().
18          default:
19              throw EvaluationException(); // e.valueless_by_exception().
20      }
21  }
```

Pattern matching: как?

```cpp
using Expression = std::any;

double evaluate(const Expression &e) {
    // return e match {
    //     AddNode: let arg => evaluate(*arg.l) + evaluate(*arg.r);
    if (auto *p = std::cast<Expression>::operator()<AddNode>(e)) {
        auto &&arg = *p;
        return evaluate(*arg.l) + evaluate(*arg.r);
    //     ...
    //     double:  let arg => arg;
    } else if (auto *p = std::cast<Expression>::operator()<double>(e)) {
        auto &&arg = *p;
        return arg;
    //     _                => throw EvaluationException(); // e.valueless_by_exception().
    } else {
        throw EvaluationException(); // e.valueless_by_exception().
    }
}
```

Pattern matching: полиморфные классы

```
1   struct Expression { virtual ~Expression() = default; };
2
3   double evaluate(const Expression &e) {
4       return e match {
5           AddNode: let arg ⇒ evaluate(*arg.l) + evaluate(*arg.r);
6           MulNode: let arg ⇒ evaluate(*arg.l) * evaluate(*arg.r);
7           ValNode: let arg ⇒ arg.v;
8           _                ⇒ throw EvaluationException();
9       };
10  }
```

- Вы не должны этого хотеть, но вдруг...
- Внутри те же самые вызовы `std::cast`, который уже вызывает `dynamic_cast`.
- Есть техники, позволяющие выполнить этот код быстрее, чем цепочку вызовов `dynamic_cast`.

Pattern matching: `switch`

```
1  double evaluate(Operation op, double l, double r) {
2      return op match {
3          Operation::Add ⇒ l + r;
4          Operation::Mul ⇒ l * r;
5          Operation::Val ⇒ l;
6          _              ⇒ throw EvaluationException();
7      };
8  }
```

- Метки являются константными выражениями.
- Матчи проверяются по порядку с помощью `operator==`.
- На деле компилируется в `switch`.

Pattern matching: лучше чем `switch`!

```
1  Color colorFromString(std::string_view s) {
2      return s match {
3          "red"    ⇒ Color::Red;
4          "green"  ⇒ Color::Green;
5          "blue"   ⇒ Color::Blue;
6          "orange" ⇒ Color::Orange;
7          _        ⇒ Color::Unknown;
8      };
9  }
```

- Помните, что матчи вычисляются по порядку.
- Есть возможности для оптимизации если позволить компилятору применить немного магии.

Pattern matching: `std::tuple`

```
1  double evaluate(const std::tuple<Operation, double, double> &op) {
2      return op match {
3          [Operation::Add, let l, let r] ⇒ l + r;
4          [Operation::Mul, let l, let r] ⇒ l * r;
5          [Operation::Val, let v, _]     ⇒ v;
6          _                              ⇒ throw EvaluationException();
7      };
8  }
```

- Код странный, исключительно для иллюстрации.
- Показывает, почему нам нужен `let`.

Pattern matching: structured bindings

```
1  double classify(Point point) {
2      return point match {
3          [0, 0] ⇒ PointClass::Origin;
4          [0, _] ⇒ PointClass::YAxis;
5          [_, 0] ⇒ PointClass::XAxis;
6          _      ⇒ PointClass::Other;
7      };
8  }
```

Паттерны можно компоновать!

```
1   using Expression = std::any;
2
3   double evaluate(const std::any &e) {
4       return e match {
5           AddNode: let [l, r] ⇒ evaluate(l) + evaluate(r);
6           MulNode: let [l, r] ⇒ evaluate(l) * evaluate(r);
7           double:  let v      ⇒ v;
8           _                   ⇒ throw EvaluationException();
9       };
10  }
```

- Также возможно:
  - `let [[a, b], c]`.
  - `[[0, 0], let c]`.
  - `[MyPair: let [a, b], 0]`.

Pattern matching: использование `if`

```
int fib(int n) {
    return n match {
        let x if (x < 0) ⇒ 0;
        1 ⇒ n;
        2 ⇒ n;
        let x ⇒ fib(x - 1) + fib(x - 2);
    };
}
```

```
if (expr match [let foo, 0]) {
    // `foo` is available here
}
```

Pattern matching: точки кастомизации

- `operator==` для матчинга константных выражений.
- `std::tuple_size<T>`, `std::tuple_element<I, T>`, и `get<I>(v)` для типов, подобных `std::tuple`.
- `std::variant_size<T>`, `std::variant_alternative<I, T>`, и `v.index()` и `get<I>(v)` для типов, подобных `std::variant`.
- `std::cast` для `std::any`, `std::exception_ptr`, и полиморфных типов.

Pattern matching: открытые вопросы

- Матчинг для `std::variant<T, T>`.
- Матчинг для `std::expected<T, T>`.
- Вызов `std::terminate` если матч не был найден — это как-то чересчур...
  - Можем ли мы проверить во время компиляции, что `std::terminate` точно не будет вызван?
- Протоколы кастомизации. Протокол для типов, подобных `std::variant`, сейчас вызывает вопросы (например, `std::visit` не реализован в терминах этого протокола).

Рефлексия

Рефлексия: немного истории

- Работа над предложеним началась до появления `constexpr`, когда у нас было только метапрограммирование на шаблонах.
- Reflection TS с `reflexpr` уходит корнями в 2010-е.
- Люди успели навелосипедить больше чем влезает на один слайд:
  - Boost.Describe.
  - Boost.PFR.
  - refl-cpp.
  - ...
- Дизайн последней итерации предложения основан на `consteval` функциях, которые появились только в C++20.

Рефлексия: базовые операции

```
1  constexpr std::meta::info r = ^int;
2  [:r:] value = 42;
3
4  std::vector<[:r:]> v = { value };
5
6  [:^vector<int>:]::iterator it;
```

- `std::meta::info` — это непрозрачный тип для рефлексии.
- Префиксный оператор ^ выполняет рефлексию.
- `[:` и `:]` создают грамматические конструкции языка из объектов типа `std::meta::info`.

Рефлексия: базовые операции

```cpp
constexpr std::meta::info v = ^std::vector;
constexpr std::meta::info a = ^int;
constexpr std::meta::info va = std::meta::substitute(v, {a});

[:va:] vec = {1, 2, 3}; // Got std::vector<int> back.
```

- Рефлексия работает для шаблонов.
- Есть ряд `consteval` функций, работающих с `std::meta::info`.

Рефлексия: доступ к полям

```cpp
struct Person {
    std::string name;
    std::string surname;
};

consteval auto member_named(std::string_view name) {
    for (std::meta::info field : nonstatic_data_members_of(^Person)) {
        if (name_of(field) == name) return field;
    }
}

int main() {
  Person p;
  p.[:member_named("name"):] = "John";
  p.[:member_named("surname"):] = "Doe";
  p.[:member_named("nickname"):] = "Coder"; // Error.
}
```

Рефлексия: кодогенерация с помощью `template for`

```cpp
using StringHandler = std::string(*)(const void *);

template<class T>
static std::string stringize(const void *ptr) {
    return std::to_string(*static_cast<const T *>(ptr));
}

constexpr std::array types = {^int, ^float, ^double}; // And maybe more.

constexpr std::array handlers = [] {
    std::array<StringHandler, types.size()> result;
    template for (std::size_t i = 0; constexpr auto e : types)
        result[i++] = &stringize<typename[:e:]>;
    return result;
}();
```

- Сегодня вы можете сделать то же самое с помощью макросов или списков типов.

Рефлексия: enum → std::string

```cpp
template <typename E>
    requires std::is_enum_v<E>
constexpr std::string enum_to_string(E value) {
    template for (constexpr auto e : std::meta::enumerators_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>");
```

Рефлексия: std::string → enum

```
template <typename E>
    requires std::is_enum_v<E>
constexpr std::optional<E> string_to_enum(std::string_view name) {
    template for (constexpr auto e : std::meta::enumerators_of(^E)) {
        if (name == std::meta::name_of(e)) {
            return [:e:];
        }
    }

    return std::nullopt;
}
```

Рефлексия: `std::string` → `enum`, но лучше

```cpp
template<class E>
    requires std::is_enum_v<E>
consteval make_enum_pairs() {
    constexpr auto size = std::meta::enumerators_of(^E).size();
    std::array<std::pair<std::string_view, E>, size> result;
    for (std::size_t i = 0; auto e : std::meta::enumerators_of(^E)) {
        result[i++] = {name_of(e), value_of<E>(e)};
    }
    return result;
}

enum Color { red, green, blue };
constexpr auto mapping = frozen::make_unordered_map(make_enum_pairs<Color>());
```

Рефлексия: std::tuple

```cpp
template<typename... Ts> struct Tuple {
  struct storage;
  [:define_class(^storage, { data_member_spec(^Ts)... }):] data;
  Tuple(): data{} {}
  Tuple(Ts const& ...vs): data{ vs... } {}
};

template<std::size_t I, typename... Ts>
struct std::tuple_element<I, Tuple<Ts...>> {
    static constexpr std::array types = {^Ts...};
    using type = [: types[I] :]; // Or you can do pack...[indexing].
};

consteval std::meta::info get_nth_nsdm(std::meta::info r, std::size_t n) {
    return nonstatic_data_members_of(r)[n];
}

template<std::size_t I, typename... Ts>
constexpr auto get(Tuple<Ts...> &t) noexcept → std::tuple_element_t<I, Tuple<Ts...>>& {
    return t.data.[:get_nth_nsdm(^decltype(t.data), I):];
}
```

Рефлексия: интересности

- Рефлексия читает текущее состояние компиляции.
- Рефлексия изменяет текущее состояние компиляции.

```
1  constexpr auto type = ^std::vector<int>;
2
3  // This requires instantiation!
4  constexpr auto members = nonstatic_data_members_of(type);
5
6  // And we can check whether the type was instantiated...
7  constexpr bool fun = is_incomplete_type(type);
```

Вы можете создать счетчик (как макрос `__COUNTER__`) с помощью рефлексии. Ранее это можно было сделать через объявления, теперь это вызов `consteval` функции.

Рефлексия: открытые вопросы

Что делает этот код?

```
1   constexpr auto tmpl = template_of(^int);
```

- Бросает исключение?
- Возвращает `std::expected`?
- Возвращает пустой `std::meta::info`?
- Какой-то другой механизм? Ошибка компиляции с "not-a-constant-expression"?

Было бы хорошо сделать обработку ошибок через исключения, но нужна поддержка исключений в `constexpr` контексте.

Рефлексия: открытые вопросы

- Текущее предложение не решает вопрос генерации кода.
  - Как сгенерировать `switch`? Актуальный вопрос для многих!
- API `define_class` в его текущей форме вызывает вопросы.
  - Как добавлять методы?
  - Как добавить `[[no_unique_address]]` к полю?
  - ...

Хотите узнать больше?

- P2688R1: Pattern Matching: `match` Expression.
- P2996R2: Reflection for C++26.
- P2300R9: `std::execution`.
- P1673R13: A free function linear algebra interface based on the BLAS.
- P1928R8: `std::simd` — merge data-parallel types from the Parallelism TS 2.
- P2900R6: Contracts for C++.

END