

Яндекс Такси

# **C++ трюки из Такси**

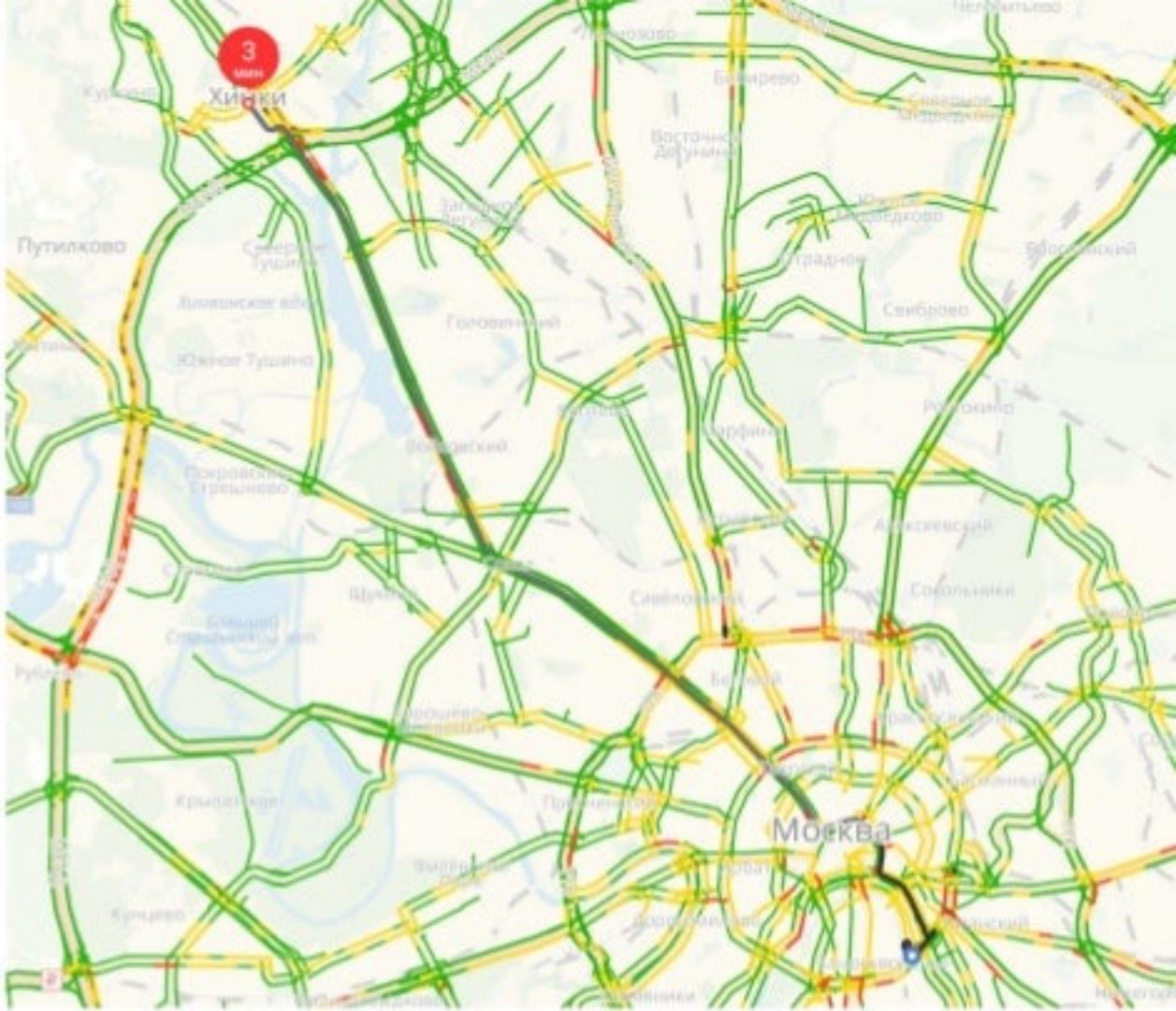
**Полухин Антон**

Antony Polukhin

**Яндекс Такси**

# Содержание

- Зачем нам это
- Pimpl
- Logging
- Parse



- C++:-)
- C++:-)



Эконом  
4Р



КОМФОРТ  
8Р



КОМФОРТ+  
**9Р**



БИЗНЕС  
34Р



МИНИВЕН  
15Р



ДЕТСКИЙ  
2Р

Комментарий, пожелания

Способ оплаты  
Команда Яндекс.Такси

# userver

# userver

– framework для написания  
высокопроизводительных приложений

# userver

– framework для написания  
высокопроизводительных приложений

Прост и удобен в использовании

# userver

– framework для написания  
высокопроизводительных приложений

Прост и удобен в использовании:

– Работа с форматами JSON/BSON/Yaml/...

# userver

- framework для написания высокопроизводительных приложений

Прост и удобен в использовании:

- Работа с форматами JSON/BSON/Yaml/...
- Логирование

# userver

- framework для написания высокопроизводительных приложений

Прост и удобен в использовании:

- Работа с форматами JSON/BSON/Yaml/...
- Логирование
- Сеть

# userver

- framework для написания высокопроизводительных приложений

Прост и удобен в использовании:

- Работа с форматами JSON/BSON/Yaml/...
- Логирование
- Сеть
- Драйвера Postgres/Mongo/...

# userver

- framework для написания высокопроизводительных приложений

Прост и удобен в использовании:

- Работа с форматами JSON/BSON/Yaml/...
- Логирование
- Сеть
- Драйвера Postgres/Mongo/...
- Корутины

- framework для написания высокопроизводительных приложений

Прост и удобен в использовании:

- Работа с форматами JSON/BSON/Yaml/...
- Логирование
- Сеть
- Драйвера Postgres/Mongo/...
- Корутины
- Кодогенерация

# userver

- framework для написания высокопроизводительных приложений

Прост и удобен в использовании:

- Работа с форматами JSON/BSON/Yaml/...
- Логирование
- Сеть
- Драйвера Postgres/Mongo/...
- Корутины
- Кодогенерация
- ...

# userver

- framework для написания высокопроизводительных приложений

Прост и удобен в использовании:

- Работа с форматами JSON/BSON/Yaml/...
- Логирование
- Сеть
- Драйвера Postgres/Mongo/...
- Корутины
- Кодогенерация
- ...

Очень-очень быстрый!

# userver

- framework для написания высокопроизводительных приложений

Прост и удобен в использовании:

- Работа с форматами JSON/BSON/Yaml/...
- Логирование
- Сеть
- Драйвера Postgres/Mongo/...
- Корутины
- Кодогенерация
- ...

**Очень-очень** быстрый!

**userver**

**Очень-очень быстрый!**

## Очень-очень быстрый!

Зачастую привычные решения нам не подходят

# Pimpl

# Pimpl

```
#include <third_party/json.hpp>

namespace formats::json {
    class Exception;
    struct Value;
} // namespace formats::json
```

# Pimpl

```
#include <third_party/json.hpp>

struct Value {
    Value() = default;
    Value(Value&& other) = default;
    Value& operator=(Value&& other) = default;
    ~Value() = default;

    std::size_t Size() const { return data_.size(); }

private:
    third_party::Json data_;
};


```

# Pimpl

```
#include <third_party/json.hpp>

struct Value {
    Value() = default;
    Value(Value&& other) = default;
    Value& operator=(Value&& other) = default;
    ~Value() = default;

    std::size_t Size() const { return data_.size(); }

private:
    third_party::Json data_;
};


```

# Pimpl

```
#include <third_party/json.hpp>

struct Value {
    Value() = default;
    Value(Value&& other) = default;
    Value& operator=(Value&& other) = default;
    ~Value() = default;

    std::size_t Size() const { return data_.size(); }

private:
    third_party::Json data_;
};

Taxi C++ tricks
```

# Pimpl

```
#include <third_party/json.hpp> // PROBLEMS!
```

```
struct Value {  
    Value() = default;  
    Value(Value&& other) = default;  
    Value& operator=(Value&& other) = default;  
    ~Value() = default;  
  
    std::size_t Size() const { return data_.size(); }  
  
private:  
    third_party::Json data_;  
};
```

# Pimpl

```
std::size_t Sample(const formats::json::Value& value) {
    try {
        return value.Size();
    } catch (const third_party::Exception& e) {
        LOG_ERROR() << e;
        return kFallback;
    }
}
```

# Pimpl

```
std::size_t Sample(const formats::json::Value& value) {
    try {
        return value.Size();
    } catch (const third_party::Exception& e) {
        LOG_ERROR() << e;
        return kFallback;
    }
}
```

# Pimpl

```
#include <third_party/json.hpp>

namespace formats::json {
    class Exception;
    struct Value;
} // namespace formats::json
```

# Pimpl

```
namespace third_party {  
    struct Json; // forward declaration  
} // namespace third_party  
  
namespace formats::json {  
    class Exception;  
    struct Value;  
} // namespace formats::json
```

# Pimpl

```
#include <impl/json_fwd.hpp>

struct Value {
    Value() = default;
    Value(Value&& other) = default;
    Value& operator=(Value&& other) = default;
    ~Value() = default;

    std::size_t Size() const { return data_.size(); }

private:
    third_party::Json data_;
};


```

# Pimpl

```
#include <impl/json_fwd.hpp>

struct Value {
    Value() = default;
    Value(Value&& other) = default;
    Value& operator=(Value&& other) = default;
    ~Value() = default;

    std::size_t Size() const { return data_.size(); }

private:
    third_party::Json data_;
};


```

# Pimpl

```
#include <impl/json_fwd.hpp>

struct Value {
    Value() = default;
    Value(Value&& other) = default;
    Value& operator=(Value&& other) = default;
    ~Value() = default;

    std::size_t Size() const { return data_.size(); }

private:
    third_party::Json data_;
};
```

# Pimpl

```
error: field 'data_' has incomplete type 'third_party::Json'  
|   third_party::Json data_;
```

# Pimpl

```
#include <impl/json_fwd.hpp>

struct Value {
    Value() = default;
    Value(Value&& other) = default;
    Value& operator=(Value&& other) = default;
    ~Value() = default;

    std::size_t Size() const { return data_.size(); }

private:
    third_party::Json data_;
};
```

# Pimpl

```
#include <impl/json_fwd.hpp>

struct Value {
    Value();
    Value(Value&& other);
    Value& operator=(Value&& other);
    ~Value();

    std::size_t Size() const;

private:
    std::unique_ptr<third_party::Json> data_;
};


```

# Pimpl

```
#include <impl/json_fwd.hpp>

struct Value {

    Value();
    Value(Value&& other);
    Value& operator=(Value&& other);
    ~Value();

    std::size_t Size() const;

private:
    std::unique_ptr<third_party::Json> data_;
};


```

# Pimpl

```
#include <formats/json.hpp>
#include <third_party/json.hpp>

Value::Value()
: data_{std::make_unique<third_party::Json>()}
{}

Value::Value(Value&& other) = default;
Value::Value& operator=(Value&& other) = default;
Value::~Value() = default;

std::size_t Value::Size() const { return data_->size(); }
```

# Pimpl

```
#include <formats/json.hpp>
#include <third_party/json.hpp>

Value::Value()
: data_{std::make_unique<third_party::Json>()}

{}

Value::Value(Value&& other) = default;
Value::Value& operator=(Value&& other) = default;
Value::~Value() = default;

std::size_t Value::Size() const { return data_->size(); }
```

# Pimpl

```
#include <formats/json.hpp>
#include <third_party/json.hpp>

Value::Value()
: data_{std::make_unique<third_party::Json>()}

{}

Value::Value(Value&& other) = default;
Value::Value& operator=(Value&& other) = default;
Value::~Value() = default;

std::size_t Value::Size() const { return data_->size(); }
```

# Pimpl

```
#include <formats/json.hpp>
#include <third_party/json.hpp>

Value::Value()
: data_{std::make_unique<third_party::Json>()}
{}

Value::Value(Value&& other) = default;
Value::Value& operator=(Value&& other) = default;
Value::~Value() = default;

std::size_t Value::Size() const { return data_->size(); }
```

# Pimpl

```
#include <formats/json.hpp>
#include <third_party/json.hpp>

Value::Value()
: data_{std::make_unique<third_party::Json>()}
{}

Value::Value(Value&& other) = default;
Value::Value& operator=(Value&& other) = default;
Value::~Value() = default;

std::size_t Value::Size() const { return data_->size(); }
```

# Pimpl

```
#include <formats/json.hpp>
#include <third_party/json.hpp>

Value::Value()
: data_{std::make_unique<third_party::Json>()} // Медленно!
{}

Value::Value(Value&& other) = default;
Value::Value& operator=(Value&& other) = default;
Value::~Value() = default;

std::size_t Value::Size() const { return data_->size(); }
```

# Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

# Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

# Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

- Динамическая аллокация

# Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

- Динамическая аллокация
- Не кеш дружелюбно

# Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

- Динамическая аллокация
- Не кеш дружелюбно

Плюсы:

# Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

- Динамическая аллокация
- Не кеш дружелюбно

Плюсы:

- Не торчат наружу детали реализации

# Fast Pimpl - основы

# Fast Pimpl основы

```
struct Value {  
    // ...  
  
private:  
    using JsonNative = third_party::Json;  
  
    std::unique_ptr<JsonNative> data_;  
};
```

# Fast Pimpl основы

```
struct Value {  
    // ...  
  
private:  
    using JsonNative = third_party::Json;  
  
    std::unique_ptr<JsonNative> data_;  
};
```

# Fast Pimpl основы

```
struct Value {  
    // ...  
  
private:  
    using JsonNative = third_party::Json;  
  
    std::aligned_storage_t<sizeof(JsonNative), alignof(JsonNative)> data_;  
};
```

# Fast Pimpl основы

```
struct Value {  
    // ...  
  
private:  
    using JsonNative = third_party::Json;  
  
    constexpr std::size_t kImplSize = 32;  
    constexpr std::size_t kImplAlign = 8;  
    std::aligned_storage_t<kImplSize, kImplAlign> data_;  
};
```

# Fast Pimpl основы

```
struct Value {  
    // ...  
  
private:  
    using JsonNative = third_party::Json;  
  
    constexpr std::size_t kImplSize = 32;  
    constexpr std::size_t kImplAlign = 8;  
    std::aligned_storage_t<kImplSize, kImplAlign> data_;  
};
```

# Fast Pimpl основы

```
struct Value {  
    // ...  
  
private:  
  
    using JsonNative = third_party::Json;  
  
    const JsonNative* Ptr() const noexcept;  
    JsonNative* Ptr() noexcept;  
  
    constexpr std::size_t kImplSize = 32;  
    constexpr std::size_t kImplAlign = 8;  
    std::aligned_storage_t<kImplSize, kImplAlign> data_;  
};
```

# Fast Pimpl основы

```
Value::Value() {
    new(Ptr()) JsonNative();
}

Value::~Value() {
    Ptr()->~JsonNative();
}

Value::JsonNative* Value::Ptr() noexcept {
    return reinterpret_cast<JsonNative*>(&data_);
}
```

# Fast Pimpl основы

```
Value::Value() {
    new(Ptr()) JsonNative();
}

Value::~Value() {
    Ptr()->~JsonNative();
}

Value::JsonNative* Value::Ptr() noexcept {
    return reinterpret_cast<JsonNative*>(&data_);
}
```

# Fast Pimpl основы

```
Value::Value() {
    new(Ptr()) JsonNative();
}

Value::~Value() {
    Ptr()->~JsonNative();
}

Value::JsonNative* Value::Ptr() noexcept {
    return reinterpret_cast<JsonNative*>(&data_);
}
```

# Fast Pimpl (основы)

В header file остаются только те части кода, которые не требуют знания о полном типе.

# Fast Pimpl (основы)

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы - неудобно пользоваться!

# Fast Pimpl (основы)

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы - неудобно пользоваться:

- Нужно писать стороннюю программу

# Fast Pimpl (основы)

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы - неудобно пользоваться:

- Нужно писать стороннюю программу
- Следить за временем жизни

# Fast Pimpl (основы)

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы - неудобно пользоваться:

- Нужно писать стороннюю программу
- Следить за временем жизни
- `reinterpret_cast`

# Fast Pimpl (основы)

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы - неудобно пользоваться:

- Нужно писать стороннюю программу
- Следить за временем жизни
- reinterpret\_cast
- Приходится сильно менять cpp файл

# Fast Pimpl (основы)

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы - неудобно пользоваться:

- Нужно писать стороннюю программу
- Следить за временем жизни
- reinterpret\_cast
- Приходится сильно менять cpp файл

Плюсы:

# Fast Pimpl (основы)

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы - неудобно пользоваться:

- Нужно писать стороннюю программу
- Следить за временем жизни
- reinterpret\_cast
- Приходится сильно менять cpp файл

Плюсы:

- Не торчат наружу детали реализации

# Fast Pimpl (основы)

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы - неудобно пользоваться:

- Нужно писать стороннюю программу:
- Следить за временем жизни
- reinterpret\_cast
- Приходится сильно менять cpp файл

Плюсы:

- Не торчат наружу детали реализации
- Кеш дружелюбно

# Fast Pimpl

# Fast Pimpl

```
struct Value {  
    // ...  
  
private:  
    using JsonNative = third_party::Json;  
  
    constexpr std::size_t kImplSize = 32;  
    constexpr std::size_t kImplAlign = 8;  
    utils::FastPimpl<JsonNative, kImplSize, kImplAlign> data_;  
};
```

# Fast Pimpl

```
struct Value {  
    // ...  
  
private:  
    using JsonNative = third_party::Json;  
  
    constexpr std::size_t kImplSize = 32;  
    constexpr std::size_t kImplAlign = 8;  
  
    utils::FastPimpl<JsonNative, kImplSize, kImplAlign> data_;  
};
```

# Fast Pimpl

```
struct Value {  
    // ...  
  
private:  
    using JsonNative = third_party::Json;  
  
    constexpr std::size_t kImplSize = 32;  
    constexpr std::size_t kImplAlign = 8;  
    utils::FastPimpl<JsonNative, kImplSize, kImplAlign> data_;  
};
```

# До FastPimpl

```
#include <formats/json.hpp>
#include <third_party/json.hpp>

Value::Value()
: data_{std::make_unique<third_party::Json>()}
{}

Value::Value(Value&& other) = default;
Value::Value& operator=(Value&& other) = default;
Value::~Value() = default;

std::size_t Value::Size() const { return data_->size(); }
```

# C FastPimpl

```
#include <formats/json.hpp>
#include <third_party/json.hpp>

Value::Value() = default;

Value::Value(Value&& other) = default;
Value::Value& operator=(Value&& other) = default;
Value::~Value() = default;

std::size_t Value::Size() const { return data_->size(); }
```

# Fast Pimpl имплементация

# FastPimpl имплементация

```
template <class T, size_t Size, size_t Alignment>
class FastPimpl {
public:
    template <class... Args>
    explicit FastPimpl(Args&&... args) {
        new (Ptr()) T(std::forward<Args>(args)...);
    }

    FastPimpl& operator=(FastPimpl&& rhs) {
        *Ptr() = std::move(*rhs);
        return *this;
    }
}
```

# FastPimpl имплементация

```
template <class T, size_t Size, size_t Alignment>
class FastPimpl {
public:
    template <class... Args>
    explicit FastPimpl(Args&&... args) {
        new (Ptr()) T(std::forward<Args>(args)...);
    }

    FastPimpl& operator=(FastPimpl&& rhs) {
        *Ptr() = std::move(*rhs);
        return *this;
    }
}
```

# FastPimpl имплементация

```
template <class T, size_t Size, size_t Alignment>
class FastPimpl {
public:
    template <class... Args>
    explicit FastPimpl(Args&&... args) {
        new (Ptr()) T(std::forward<Args>(args)...);
    }
}
```

```
FastPimpl& operator=(FastPimpl&& rhs) {
    *Ptr() = std::move(*rhs);
    return *this;
}
```

}

# FastPimpl имплементация

```
template <class T, size_t Size, size_t Alignment>
class FastPimpl {
public:
    template <class... Args>
    explicit FastPimpl(Args&&... args) {
        new (Ptr()) T(std::forward<Args>(args)...);
    }
```

```
FastPimpl& operator=(FastPimpl&& rhs) {
    *Ptr() = std::move(*rhs);
    return *this;
}
```

# FastPimpl имплементация

```
T* operator->() noexcept { return Ptr(); }

const T* operator->() const noexcept { return Ptr(); }

T& operator*() noexcept { return *Ptr(); }

const T& operator*() const noexcept { return *Ptr(); }
```

# FastPimpl имплементация

```
~FastPimpl() noexcept {
    validate<sizeof(T), alignof(T)>();
    Ptr()->~T();
}
```

# FastPimpl имплементация

```
~FastPimpl() noexcept {  
    validate<sizeof(T), alignof(T)>();  
    Ptr() ->~T();  
}
```

# FastPimpl имплементация

```
template <class T, size_t Size, size_t Alignment>
class FastPimpl {
    // ...
private:
    template <std::size_t ActualSize, std::size_t ActualAlignment>
    static void validate() noexcept {
        static_assert(Size == ActualSize, "Size and sizeof(T) mismatch");
        static_assert(Alignment == ActualAlignment, "Alignment and alignof(T) mismatch");
    }
}
```

# FastPimpl имплементация

```
template <class T, size_t Size, size_t Alignment>
class FastPimpl {
    // ...
private:
    template <std::size_t ActualSize, std::size_t ActualAlignment>
    static void validate() noexcept {
        static_assert(Size == ActualSize, "Size and sizeof(T) mismatch");
        static_assert(Alignment == ActualAlignment, "Alignment and alignof(T) mismatch");
    }
}
```

# FastPimpl имплементация

```
template <class T, size_t Size, size_t Alignment>
class FastPimpl {
    // ...
private:
    template <std::size_t ActualSize, std::size_t ActualAlignment>
    static void validate() noexcept {
        static_assert(Size == ActualSize, "Size and sizeof(T) mismatch");
        static_assert(Alignment == ActualAlignment, "Alignment and alignof(T) mismatch");
    }
}
```

# FastPimpl имплементация

```
<source>: In instantiation of 'void FastPimpl<T, Size, Alignment>::validate() [with int  
ActualSize = 32; int ActualAlignment = 8; T = std::string; int Size = 8; int Alignment =  
8]'
```

# Fast Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

# Fast Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

# Fast Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

- Нужно написать 2 константы

# Fast Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

- Нужно написать 2 константы

Плюсы:

# Fast Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

- Нужно написать 2 константы

Плюсы:

- Не торчат наружу детали реализации

# Fast Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

- Нужно написать 2 константы

Плюсы:

- Не торчат наружу детали реализации
- Кеш дружелюбно

# Fast Pimpl

В header file остаются только те части кода, которые не требуют знания о полном типе.

Минусы:

- Нужно написать 2 константы

Плюсы:

- Не торчат наружу детали реализации
- Кеш дружелюбно
- Минимум изменений в cpp файле

# Логирование

# Логирование

# Логирование

```
#include <iostream>

std::cerr << "Log message";
```

# Логирование

```
#include <iostream>

std::cerr << timestamp() << "Log message";
```

# Логирование

```
#include <iostream>

std::osyncstream{std::cerr} << timestamp() << "Log message";
```

# Логирование

```
struct LogHelper {  
    LogHelper() { oss << timestamp(); }  
    ~LogHelper() { std::cerr << oss.str(); }  
private:  
    std::ostringstream oss;  
};  
  
template <class T>  
LogHelper& operator<<(LogHelper& lh, const T& value) {  
    lh.oss << value;  
    return lh;  
}
```

# Логирование

```
struct LogHelper {  
    LogHelper() { oss << timestamp(); }  
    ~LogHelper() { std::cerr << oss.str(); }  
    private:  
    std::ostringstream oss;  
};  
  
template <class T>  
LogHelper& operator<<(LogHelper& lh, const T& value) {  
    lh.oss << value;  
    return lh;  
}
```

# Логирование

```
struct LogHelper {  
    LogHelper() { oss << timestamp(); }  
    ~LogHelper() { std::cerr << oss.str(); }  
private:  
    std::ostringstream oss;  
};  
  
template <class T>  
LogHelper& operator<<(LogHelper& lh, const T& value) {  
    lh.oss << value;  
    return lh;  
}
```

# Логирование

```
struct LogHelper {  
    LogHelper() { oss << timestamp(); }  
    ~LogHelper() { std::cerr << oss.str(); }  
private:  
    std::ostringstream oss;  
};  
  
template <class T>  
LogHelper& operator<<(LogHelper& lh, const T& value) {  
    lh.oss << value;  
    return lh;  
}
```

# Логирование

```
struct LogHelper {  
    LogHelper() { oss << timestamp(); }  
    ~LogHelper() { std::cerr << oss.str(); }  
private:  
    std::ostringstream oss; // Очень-очень медленно!  
};  
  
template <class T>  
LogHelper& operator<<(LogHelper& lh, const T& value) {  
    lh.oss << value;  
    return lh;  
}
```

# Логирование (было)

```
private:  
    std::ostringstream oss;  
  
};
```

# Логирование (стало)

```
private:  
    FastStackBuffer buffer_;  
  
    struct LazyInitedStream;  
    std::optional<LazyInitedStream> lazy_;  
};
```

# Логирование (стало)

```
private:  
    FastStackBuffer buffer_;// буффер использующий стек через std::array<char, 1024>  
  
    struct LazyInitedStream;  
    std::optional<LazyInitedStream> lazy_;  
};
```

# Логирование (стало)

```
private:  
    FastStackBuffer buffer_;// буффер использующий стек через std::array<char, 1024>  
  
    struct LazyInitedStream; // содержит std::ostream  
    std::optional<LazyInitedStream> lazy_;  
};
```

# Логирование (стало)

```
private:  
    FastStackBuffer buffer_;// буффер использующий стек через std::array<char, 1024>  
  
    struct LazyInitedStream; // содержит std::ostream  
    std::optional<LazyInitedStream> lazy_;// по умолчанию ничего не конструирует  
};
```

# Логирование (было)

```
template <class T>

LogHelper& operator<<(LogHelper& lh, const T& value) {

    lh.osss << value;

    return lh;

}
```

# Логирование (стало)

```
template <typename T>

LogHelper& LogHelper::operator<<(const T& value) {
    if constexpr (std::is_constructible<std::string_view, T>::value) {
        buffer_.Put(value);
    } else if constexpr (std::is_base_of<std::exception, T>::value) {
        buffer_.PutException(value);
    } else {
        Stream() << value;
    }
    return *this;
}
```

# Логирование (стало)

```
template <typename T>

LogHelper& LogHelper::operator<<(const T& value) {

    if constexpr (std::is_constructible<std::string_view, T>::value) {
        buffer_.Put(value);
    } else if constexpr (std::is_base_of<std::exception, T>::value) {
        buffer_.PutException(value);
    } else {
        Stream() << value;
    }

    return *this;
}
```

# Логирование (стало)

```
template <typename T>

LogHelper& LogHelper::operator<<(const T& value) {
    if constexpr (std::is_constructible<std::string_view, T>::value) {
        buffer_.Put(value);
    } else if constexpr (std::is_base_of<std::exception, T>::value) {
        buffer_.PutException(value);
    } else {
        Stream() << value;
    }
    return *this;
}
```

# Логирование (стало)

```
template <typename T>

LogHelper& LogHelper::operator<<(const T& value) {
    if constexpr (std::is_constructible<std::string_view, T>::value) {
        buffer_.Put(value);
    } else if constexpr (std::is_base_of<std::exception, T>::value) {
        buffer_.PutException(value);
    } else {
        Stream() << value;
    }
    return *this;
}
```

# Логирование (стало)

```
private:  
    FastStackBuffer buffer_;// буффер использующий стек через std::array<char, 1024>  
  
    struct LazyInitedStream; // содержит std::ostream  
    std::optional<LazyInitedStream> lazy_;// по умолчанию ничего не конструирует  
};
```

# Логирование

```
private:  
    FastStackBuffer buffer_;  
  
    struct LazyInitStream {  
        BufferStd sbuf;  
        std::ostream ostr;  
        explicit LazyInitStream(FastStackBuffer& impl) : sbuf{impl}, ostr(&sbuf) {}  
    };  
  
    std::optional<LazyInitStream> lazy_;  
};
```

# Логирование

```
private:  
    FastStackBuffer buffer_;  
  
    struct LazyInitStream {  
        BufferStd sbuf;  
        std::ostream ostr;  
        explicit LazyInitStream(FastStackBuffer& impl) : sbuf{impl}, ostr(&sbuf) {}  
    };  
  
    std::optional<LazyInitStream> lazy_;  
};
```

# Логирование

```
struct BufferStd final : public std::streambuf {
    explicit BufferStd(FastStackBuffer& impl) : impl_{impl} {}

private:
    int_type overflow(int_type c) override;
    std::streamsize xsputn(const char_type* s, std::streamsize n) override;
    FastStackBuffer& impl_;
};

};
```

# Логирование

```
struct BufferStd final : public std::streambuf {  
    explicit BufferStd(FastStackBuffer& impl) : impl_{impl} {}  
  
    private:  
        int_type overflow(int_type c) override;  
        std::streamsize xputn(const char_type* s, std::streamsize n) override;  
        FastStackBuffer& impl_;  
};
```

# Логирование

```
struct BufferStd final : public std::streambuf {  
    explicit BufferStd(FastStackBuffer& impl) : impl_{impl} {}  
  
private:  
    int_type overflow(int_type c) override;  
    std::streamsize xsputn(const char_type* s, std::streamsize n) override;  
    FastStackBuffer& impl_;  
};
```

# Логирование

```
struct BufferStd final : public std::streambuf {  
    explicit BufferStd(FastStackBuffer& impl) : impl_{impl} {}  
  
private:  
    int_type overflow(int_type c) override;  
    std::streamsize xsputn(const char_type* s, std::streamsize n) override;  
    FastStackBuffer& impl_;  
};
```

# Логирование

```
private:  
    FastStackBuffer buffer_;  
  
    std::optional<LazyInitStream> lazy_;  
  
    std::ostream& Stream() {  
        if (!lazy_) {  
            lazy_.emplace(buffer_);  
        }  
        return lazy_->ostr;  
    }  
};
```

# Логирование

```
private:  
    FastStackBuffer buffer_;  
  
    std::optional<LazyInitedStream> lazy_;  
  
    std::ostream& Stream() {  
        if (!lazy_) {  
            lazy_.emplace(buffer_);  
        }  
        return lazy_->ostr;  
    }  
};
```

# Логирование

```
std::ostream& operator<<(std::ostream& os, const models::DriverInfo& info) {  
    return os << "Id = " << info.id << ", name = " << info.name;  
}
```

# Логирование

```
std::ostream& operator<<(std::ostream& os, const models::DriverInfo& info) {  
    return os << "Id = " << info.id << ", name = " << info.name;  
}
```

```
LogHelper& operator<<(LogHelper& lh, const models::DriverInfo& info) {  
    return lh << "Id = " << info.id << ", name = " << info.name;  
}
```

# Логирование

```
LogHelper& operator<<(LogHelper& lh, const models::DriverInfo& info) {
    return lh << "Id = " << info.id << ", name = " << info.name;
}
```

# Логирование (бенчмарки)

	Было		Стало
<hr/>			
LogNumber<int>	682 ns		321 ns
LogNumber<long>	680 ns		318 ns
LogChar/8	690 ns		334 ns
LogChar/16	718 ns		419 ns
LogChar/512	2066 ns		1928 ns
LogStruct	812 ns		685 ns

Note: в бенчмарках LogHelper() так же пишет уровень логирования, время, id корутины, id потока, id соединения; ~LogHelper() честно отрабатывает и отсылает данные в аналог std::cerr.

Несмотря на эти «неоптимизируемые расходы», мы получаем прирост производительности в ~2 раза.

# Логирование (маленькая хитрость)

```
Id id{42};  
  
LogHelper lh{kDebug};  
  
lh << staktrace() << id << ':' << db.FetchUserById(id);
```

# Логирование (маленькая хитрость)

```
Id id{42};  
  
LOG_DEBUG() << staktrace() << id << ':' << db.FetchUserById(id);
```

# Логирование (маленькая хитрость)

```
#define LOG_DEBUG() \
if (kDebug >= CurrentLogLevel()) LogHelper{}.AsLValue()
```

# Логирование

# Логирование

Плюсы:

# Логирование

Плюсы:

- Нет динамических аллокаций

# Логирование

Плюсы:

- Нет динамических аллокаций
- Быстрое форматирование без `std::locale`

# Логирование

Плюсы:

- Нет динамических аллокаций
- Быстрое форматирование без `std::locale`
- Переключение на рантайме уровня логирования

# Логирование

Плюсы:

- Нет динамических аллокаций
- Быстрое форматирование без `std::locale`
- Переключение на рантайме уровня логирования
- Не вычисляем лишнего

# Parse

# Parse

```
auto inf = json.As<models::DriverInfo>();
```

# Parse

```
auto inf = json.As<models::DriverInfo>();  
auto inf = yaml.As<models::DriverInfo>();
```

# Parse

```
auto inf = json.As<models::DriverInfo>();  
auto inf = yaml.As<models::DriverInfo>();  
auto inf = bson.As<models::DriverInfo>();
```

# Parse

– точка кастомизации, которая должна искать в:

# Parse

- точка кастомизации, которая должна искать в:
  - namespace T

# Parse

- точка кастомизации, которая должна искать в:
  - namespace T
  - где-то, где описаны общие парсеры

# Parse

- точка кастомизации, которая должна искать в:
  - namespace T
  - где-то, где описаны общие парсеры
  - где-то, где описаны формато-специфичные парсеры

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return Parse(*this); // ???  
        }  
  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return Parse(*this); // ???  
        }  
  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return T::Parse(*this);  
        }  
  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return T::Parse(*this); // std::vector<DriverInfo>, boost::optional<DriverInfo> :(  
        }  
  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return Parse<T>(*this);  
        }  
  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return Parse<T>(*this); // std::vector<DriverInfo>, boost::optional<DriverInfo> :(  
        }  
  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            T value;  
            Parse(*this, value);  
            return value;  
        }  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            T value; // не default constructible типы?  
            Parse(*this, value);  
            return value;  
        }  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            T* value = nullptr;  
            return Parse(*this, value);  
        }  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            T* value = nullptr; // Отстрел ноги :(  
            return Parse(*this, value);  
        }  
    };  
} // namespace formats::json
```

# Parse — что делать?

# Parse

```
#pragma once

namespace formats::parse {

template <class T>
struct To {};

} // namespace formats::parse
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return Parse(*this, parse::To<T>{});  
        }  
  
    };  
} // namespace formats::json
```

# ADL

# ADL

- Ищет функции с заданными именем в
  - namespace аргументов функции

# ADL

Ищет функции с заданными именем в

- namespace аргументов функции
- и namespace шаблонов аргументов функций

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return Parse(*this, parse::To<T>{});  
        }  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return Parse(*this, parse::To<T>{}); // namespace formats::<json>  
        }  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return Parse(*this, parse::To<T>{}); // namespace formats::parse  
        }  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::json {  
    struct Value {  
  
        template <class T>  
        T As() {  
            return Parse(*this, parse::To<T>{}); // namespace of T  
        }  
    };  
} // namespace formats::json
```

# Parse

```
namespace formats::parse {  
  
template <class Value, typename T>  
auto Parse(const Value& value, To<std::unordered_map<std::string, T>>);  
  
} // namespace formats::parse
```

# Parse

```
namespace drivers::models {  
  
template <class Value>  
auto Parse(const Value& v, formats::parse::To<DriverInfo>);  
  
} // namespace drivers::models
```

# Parse

```
namespace formats::bson {  
  
auto Parse(const bson::Value& v, parse::To<std::chrono::system_clock::time_point>);  
  
} // namespace formats::bson
```

# Parse

# Parse

## Плюсы

# Parse

Плюсы:

- Одна функция на все форматы

# Parse

Плюсы:

- Одна функция на все форматы
- Можно писать парсеры в своём namespace

# Parse

Плюсы:

- Одна функция на все форматы
- Можно писать парсеры в своём namespace
- Нет мороки с linker или порядком заголовков

**Спасибо**

# Полухин Антон

Эксперт-разработчик C++



antoshkka@gmail.com



antoshkka@yandex-team.ru



<https://github.com/apolukhin>



<https://stdcpp.ru/>

