Яндекс Такси

C++ трюки из Такси

version 1.2

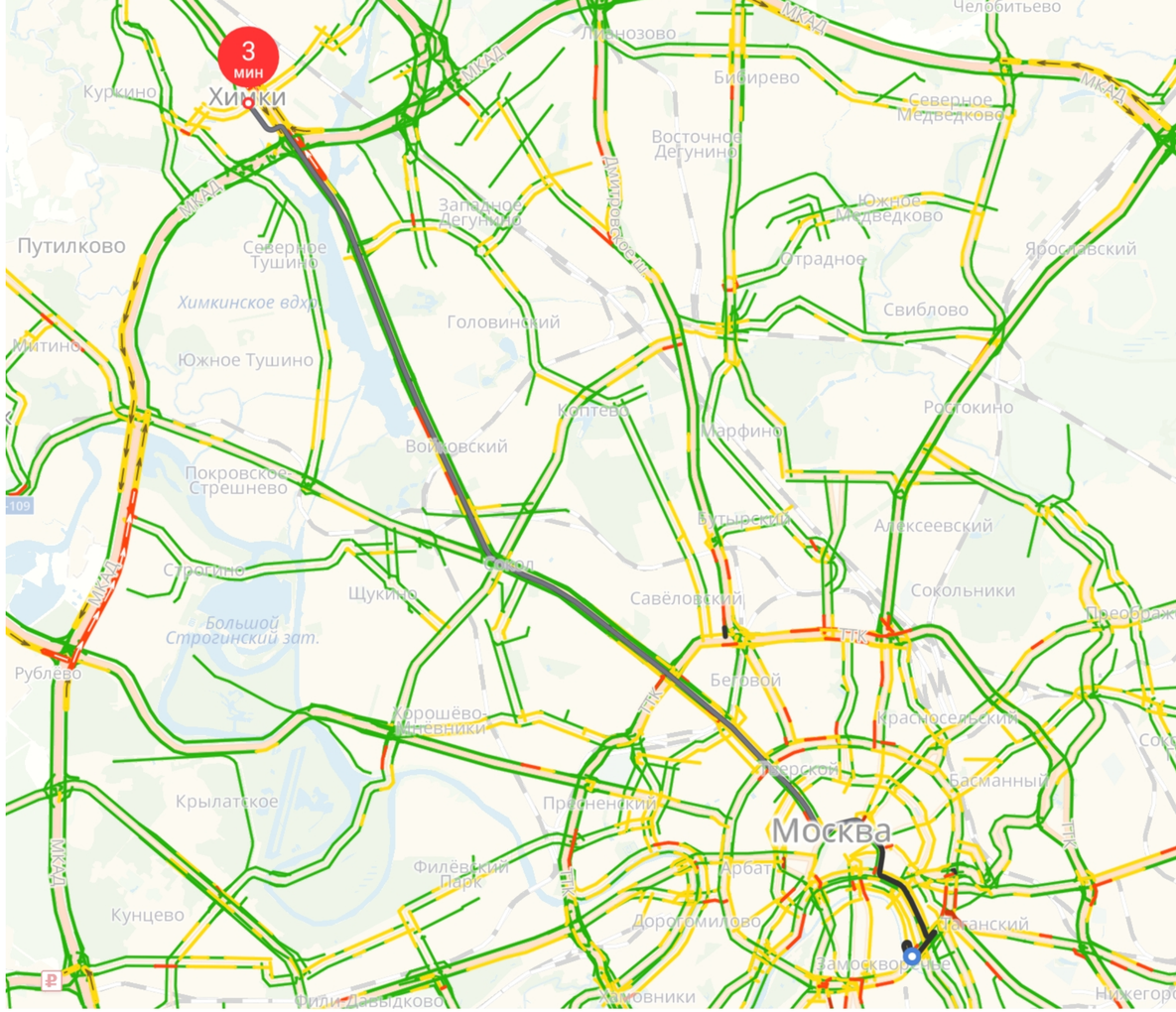
Полухин Антон

Antony Polukhin

Яндекс Такси

Содержание

- Зачем нам это
- Pimpl
- Logging
- Parse
- Typedef
- Perf or compile times?



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

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
- ...

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!

userver

Очень-очень быстрый!

Зачастую привычные решения нам не подходят

Pimpl

Pimpl

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


namespace formats::json {

class Exception;

struct Value;

}  // namespace formats::json
```

Pimpl

```cpp
#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

```cpp
#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

```cpp
#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>   // 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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#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.oss << value;
  return lh;
}
```

Логирование (стало)

```
template <typename T>
LogHelper& LogHelper::operator<<(const T& value) {
  if constexpr (std::is_constructible<utils::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<utils::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<utils::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<utils::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 LazyInitedStream {
    BufferStd sbuf;
    std::ostream ostr;
    explicit LazyInitedStream(FastStackBuffer& impl) : sbuf{impl}, ostr(&sbuf) {}
  };

  std::optional<LazyInitedStream> lazy_;
};
```

Логирование

```
private:
  FastStackBuffer buffer_;

  struct LazyInitedStream {
    BufferStd sbuf;
    std::ostream ostr;
    explicit LazyInitedStream(FastStackBuffer& impl) : sbuf{impl}, ostr(&sbuf) {}
  };

  std::optional<LazyInitedStream> 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 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

```
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<LazyInitedStream> 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
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:

- namespce T

Parse

– точка кастомизации, которая должна искать в:

- namespce T
- где-то, где описаны общие парсеры

Parse

– точка кастомизации, которая должна искать в:

- namespce 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

```cpp
#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 namepsace

Parse

Плюсы:

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

Strong typedef

BAD

```
SomeResult DoSomething(int payment_id, int client_id);
```

BAD

```
SomeResult DoSomething(int payment_id, int client_id);

auto res = DoSomething(request.client, request.payment);
```

Strong typedef

```
enum class PaymentID : int {};

enum class ClientID : int {};


SomeResult DoSomething(PaymentID payment_id, ClientID client_id);
```

BAD

```
SomeResult DoSomething(std::string payment_id, std::string client_id);
```

BAD

```
SomeResult DoSomething(std::string payment_id, std::string client_id);

auto res = DoSomething(request.client, request.payment);
```

Strong typedef

```cpp
#include <utils/strong_typedef.hpp>
```

Strong typedef

```cpp
#include <utils/strong_typedef.hpp>

using PaymentId = utils::StrongTypedef<class PaymentIdTag, std::string>;
```

Strong typedef

```cpp
#include <utils/strong_typedef.hpp>

using PaymentId = utils::StrongTypedef<class PaymentIdTag, std::string>;

class ClientID : public utils::StrongTypedef<ClientID, std::string> {
    using StrongTypedef::StrongTypedef;
};
```

Strong typedef

```cpp
#include <utils/strong_typedef.hpp>

using PaymentId = utils::StrongTypedef<class PaymentIdTag, std::string>;
class ClientID : utils::StrongTypedef<ClientID, std::string> {
    using StrongTypedef::StrongTypedef;
};

SomeResult DoSomething(PaymentId payment_id, ClientID client_id);
```

Strong typedef

```
// Generic implementation for classes
template <class Tag, class T>
class StrongTypedef {
  static_assert(!std::is_reference<T>::value);
  static_assert(!std::is_pointer<T>::value);

  static_assert(!std::is_reference<Tag>::value);
  static_assert(!std::is_pointer<Tag>::value);

 public:
  using UnderlyingType = T;
```

Strong typedef

```
 private:
  T data_{};
};
```

Strong typedef

```
StrongTypedef() = default;

StrongTypedef(const StrongTypedef&) = default;

StrongTypedef(StrongTypedef&&) = default;

StrongTypedef& operator=(const StrongTypedef&) = default;

StrongTypedef& operator=(StrongTypedef&&) = default;


template < class... Args,

    std::enable_if_t<std::is_constructible<T, Args...>::value, bool> = true>

explicit constexpr StrongTypedef(Args&&... args) noexcept(

    noexcept(T(std::forward<Args>(args)...)))

    : data_(std::forward<Args>(args)...) {}
```

Strong typedef

```
explicit constexpr operator const T&() const& noexcept { return data_; }
explicit constexpr operator T() && noexcept { return std::move(data_); }
explicit constexpr operator T&() & noexcept { return data_; }

constexpr const T& GetUnderlying() const& noexcept { return data_; }
constexpr T GetUnderlying() && noexcept { return std::move(data_); }
constexpr T& GetUnderlying() & noexcept { return data_; }
```

Strong typedef

```
auto begin() { return data_.begin(); }
auto end() { return data_.end(); }
auto begin() const { return data_.begin(); }
auto end() const { return data_.end(); }
auto cbegin() const { return data_.cbegin(); }
auto cend() const { return data_.cend(); }
auto size() const { return data_.size(); }
auto empty() const { return data_.empty(); }
auto clear() { return data_.clear(); }


template <class Arg>
decltype(auto) operator[](Arg&& i) { return data_[std::forward<Arg>(i)]; }
```

Strong typedef

Плюсы

Strong typedef

Плюсы:

- Type safety

Strong typedef

Плюсы:

- Type safety
- Быстрое создание новых типов в 1-3 строки

Perf or compile time?

Header files

Header files

```
#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_;
};
```

Header files

```
#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_;
};
```

Header files

```
#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;


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

Header files

Плюсы:

- Чем больше описано в header файле — тем лучше компилятор оптимизирует

Header files

Плюсы:

- Чем больше описано в header файле — тем лучше компилятор оптимизирует

Минусы:

- Чем больше описано в header файле — тем медленнее компилятор компилирует

LTO

LTO

Плюсы:

- Оптимизирует весь проект как если бы все исходники были в одном файле => отличный perf

LTO

Плюсы:

- Оптимизирует весь проект как если бы все исходники были в одном файле => отличный perf
- Нет ощутимого замедления компиляции

LTO

Плюсы:

- Оптимизирует весь проект как если бы все исходники были в одном файле => отличный perf
- Нет ощутимого замедления компиляции
- Умеет кешировать линковку

LTO

Плюсы:

- Оптимизирует весь проект как если бы все исходники были в одном файле => отличный perf
- Нет ощутимого замедления компиляции
- Умеет кешировать линковку
- Умеет многопоточно линковать

LTO

Плюсы:

- Оптимизирует весь проект как если бы все исходники были в одном файле => отличный perf
- Нет ощутимого замедления компиляции
- Умеет кешировать линковку
- Умеет многопоточно линковать

Минусы:

- Дольше линкует

LTO

Плюсы:

- Оптимизирует весь проект как если бы все исходники были в одном файле => отличный perf
- Нет ощутимого замедления компиляции
- Умеет кешировать линковку
- Умеет многопоточно линковать

Минусы:

- Дольше линкует?

Спасибо

Полухин Антон

Эксперт-разработчик C++

antoshkka@gmail.com

antoshkka@yandex-team.ru

https://github.com/apolukhin

https://stdcpp.ru/

РГ21 C++ РОССИЯ