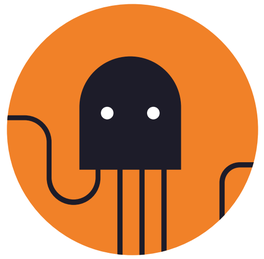
C++ трюки из Такси

Полухин Антон

Antony Polukhin

Яндекс Go

C++ трюки из userver

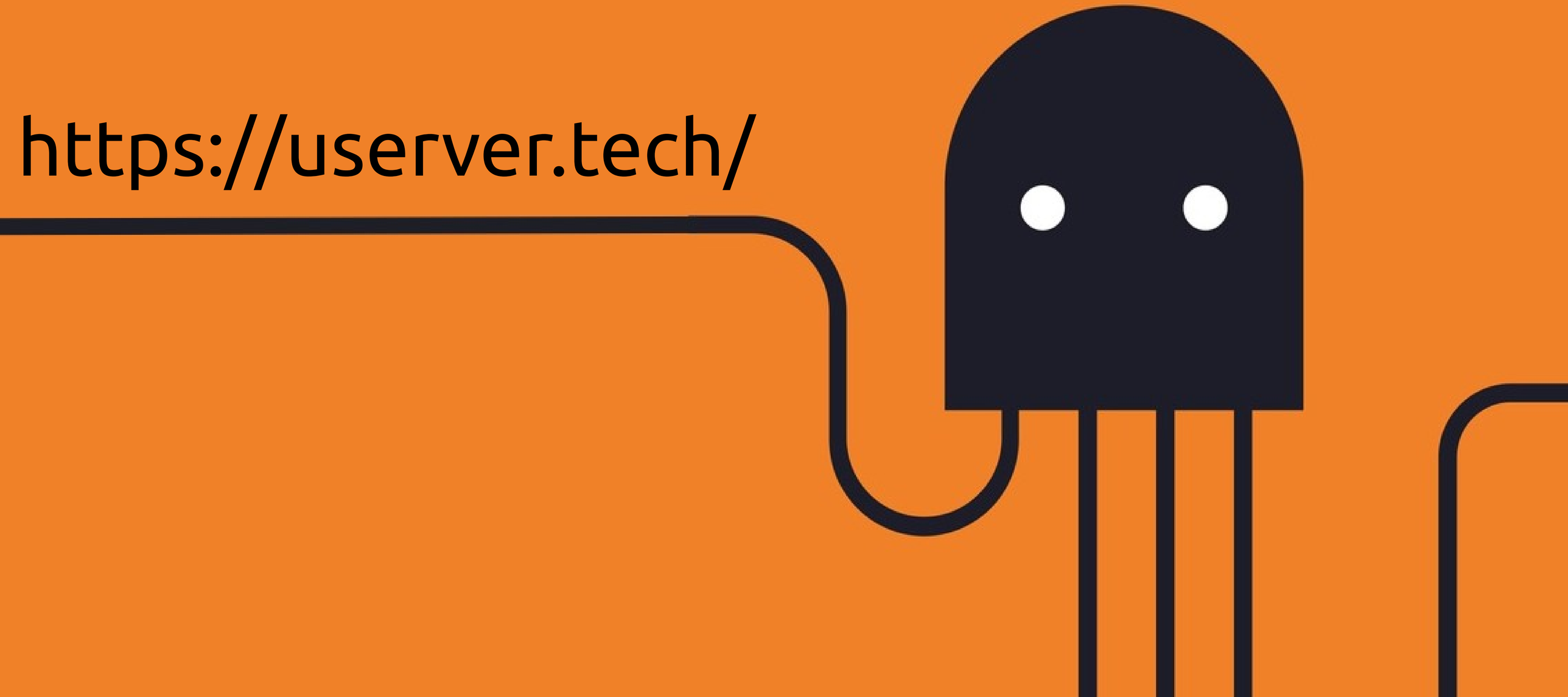


Полухин Антон

Antony Polukhin

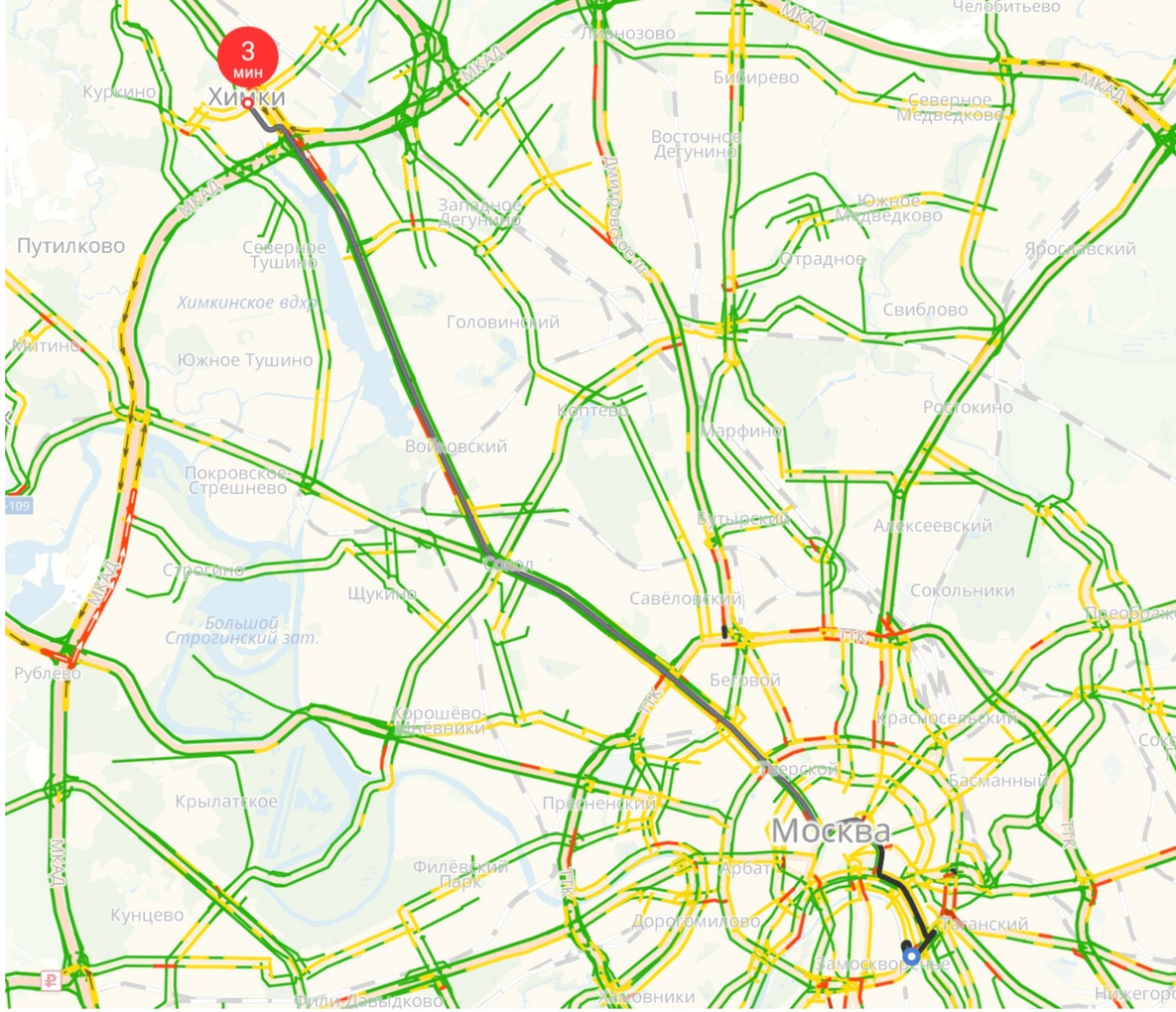
Яндекс Go

https://userver.tech/



Содержание

- utils::LazyPrvalue
- utils::FixedArray
- ODR-violation самодиагностика
- самая чудная bi-map
- Загадка про std::shared_ptr



Когда чужой код «специфичен»

Когда чужой код «специфичен»

Когда чужой код «специфичен»

```cpp
#include <mutex>
#include <vector>

namespace third_party_lib {

class Something {
 public:
    static Something Create(std::size_t value);

    // ...

 private:
    std::mutex mutex_;
};

}
```

Когда чужой код «специфичен»

```
#include <mutex>
#include <vector>

namespace third_party_lib {

class Something {
 public:
    static Something Create(std::size_t value);

    // ...

 private:
    std::mutex mutex_;
};

}
```

Когда чужой код «специфичен»

```cpp
#include <mutex>
#include <vector>

namespace third_party_lib {

class Something {
 public:
    static Something Create(std::size_t value);

    // ...

 private:
    std::mutex mutex_;
};

}
```

Когда чужой код «специфичен»

```
#include <mutex>
#include <vector>

namespace third_party_lib {

class Something {
 public:
    static Something Create(std::size_t value);

    // ...

 private:
    std::mutex mutex_;
};

}
```

Когда чужой код «специфичен»

```cpp
#include <mutex>
#include <vector>

namespace third_party_lib {

class Something {
 public:
    static Something Create(std::size_t value);

    // ...

 private:
    std::mutex mutex_;
};

}
```

Когда чужой код «специфичен»

```
class Component {
 public:
    // ...
 private:
    std::list<third_party_lib::Something> clients_;
};
```

Когда чужой код «специфичен»

```
class Component {
 public:
    // ...
 private:
    std::list<third_party_lib::Something> clients_;
};
```

Когда чужой код «специфичен»

```
class Component {
 public:
    // ...
 private:
    std::list<third_party_lib::Something> clients_;
};
```

Когда чужой код «специфичен»

```
class Component {
 public:

    Component() {
        constexpr std::size_t kClientsCount = 42;
        for (std::size_t i = 0; i < kClientsCount; ++i) {
            clients_.emplace_back(
                third_party_lib::Something::Create(i)
            );
        }
    }

    // ...
 private:
    std::list<third_party_lib::Something> clients_;
};
```

Когда чужой код «специфичен»

```
class Component {
 public:

    Component() {
        constexpr std::size_t kClientsCount = 42;
        for (std::size_t i = 0; i < kClientsCount; ++i) {
            clients_.emplace_back(
                third_party_lib::Something::Create(i)
            );
        }
    }

    // ...
 private:
    std::list<third_party_lib::Something> clients_;
};
```

Когда чужой код «специфичен»

```
error: no matching function for call to
'construct_at(third_party_lib::Something*&, third_party_lib::Something)'
  518 |           std::construct_at(__p, std::forward<_Args>(__args)...);
      |           ~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

prvalue

prvalue

```
Something Create();

auto list::emplace_back(Something&& v);
```

prvalue

```
Something Create();  // prvalue

auto list::emplace_back(Something&& v);
```

prvalue

```
Something Create();  // prvalue

auto list::emplace_back(Something&& v);  // не prvalue
```

prvalue

```
Something Create();  // prvalue

auto list::emplace_back(Something&& v);  // не prvalue, есть имя
```

prvalue

```
auto list::emplace_back(Something&& v) {    // не prvalue
    // ...
    new (&node.value) Something(v);         // v - не prvalue
    // ...
}
```

prvalue

```
template <typename Arg>
auto list::emplace_back(Arg&& v) {   // не prvalue
    // ...
    new (&node.value) Something(v);  // v - не prvalue
    // ...
}
```

prvalue

```
template <typename Arg>
auto list::emplace_back(Arg&& v) {   // не prvalue
    // ...
    new (&node.value) Something(v);  // v - не prvalue, Something(v) - prvalue
    // ...
}
```

Когда чужой код «специфичен»

```cpp
namespace utils {

template <typename Func>
class LazyPrvalue final {
 public:
  constexpr explicit LazyPrvalue(Func&& func) : func_(std::move(func)) {}

  LazyPrvalue(LazyPrvalue&&) = delete;
  LazyPrvalue& operator=(LazyPrvalue&&) = delete;

  constexpr /* implicit */ operator std::invoke_result_t<Func&&>() && {
    return std::move(func_)();
  }

 private:
  Func func_;
};

}
```

Когда чужой код «специфичен»

```
namespace utils {

template <typename Func>
class LazyPrvalue final {
 public:
  constexpr explicit LazyPrvalue(Func&& func) : func_(std::move(func)) {}

  LazyPrvalue(LazyPrvalue&&) = delete;
  LazyPrvalue& operator=(LazyPrvalue&&) = delete;

  constexpr /* implicit */ operator std::invoke_result_t<Func&&>() && {
    return std::move(func_)();
  }

 private:
  Func func_;
};

}
```

Когда чужой код «специфичен»

```
namespace utils {

template <typename Func>
class LazyPrvalue final {
 public:
  constexpr explicit LazyPrvalue(Func&& func) : func_(std::move(func)) {}

  LazyPrvalue(LazyPrvalue&&) = delete;
  LazyPrvalue& operator=(LazyPrvalue&&) = delete;

  constexpr /* implicit */ operator std::invoke_result_t<Func&&>() && {
    return std::move(func_)();
  }

 private:
  Func func_;
};

}
```

Когда чужой код «специфичен»

```
namespace utils {

template <typename Func>
class LazyPrvalue final {
 public:
  constexpr explicit LazyPrvalue(Func&& func) : func_(std::move(func)) {}

  LazyPrvalue(LazyPrvalue&&) = delete;
  LazyPrvalue& operator=(LazyPrvalue&&) = delete;

  constexpr /* implicit */ operator std::invoke_result_t<Func&&>() && {
    return std::move(func_)();
  }

 private:
  Func func_;
};

}
```

prvalue

```
template <typename Arg>
auto list::emplace_back(Arg&& v) {   // не prvalue
    // ...
    new (&node.value) Something(v);  // v - не prvalue, Something(v) - prvalue
    // ...
}
```

prvalue

```
auto list::emplace_back(LazyPrvalue&& v) {   // не prvalue
    // ...
    new (&node.value) Something(v);  // v - не prvalue, Something(v) - prvalue
    // …
}
```

Когда чужой код «специфичен»

```
class Component {
 public:

    Component() {
        constexpr std::size_t kClientsCount = 42;
        for (std::size_t i = 0; i < kClientsCount; ++i) {
            clients_.emplace_back(

                third_party_lib::Something::Create(i)

            );
        }
    }

    // ...
 private:
    std::list<third_party_lib::Something> clients_;
};
```

Когда чужой код «специфичен»

```
class Component {
 public:

    Component() {
        constexpr std::size_t kClientsCount = 42;
        for (std::size_t i = 0; i < kClientsCount; ++i) {
            clients_.emplace_back(

                third_party_lib::Something::Create(i)

            );
        }
    }

    // ...
 private:
    std::list<third_party_lib::Something> clients_;
};
```

Когда чужой код «специфичен»

```
class Component {
 public:

    Component() {
        constexpr std::size_t kClientsCount = 42;
        for (std::size_t i = 0; i < kClientsCount; ++i) {
            clients_.emplace_back(
                utils::LazyPrvalue([i]() {
                    return third_party_lib::Something::Create(i);
                })
            );
        }
    }

    // ...
 private:
    std::list<third_party_lib::Something> clients_;
};
```

Когда чужой код «специфичен»

```
class Component {
 public:

    Component() {
        constexpr std::size_t kClientsCount = 42;
        for (std::size_t i = 0; i < kClientsCount; ++i) {
            clients_.emplace_back(
                utils::LazyPrvalue([i]() {
                    return third_party_lib::Something::Create(i);
                })
            );
        }
    }

    // ...
 private:
    std::list<third_party_lib::Something> clients_;
};
```

Когда std::vector пасует

Когда стандартные контейнеры не подходят

Когда стандартные контейнеры не подходят

```cpp
#include <mutex>
#include <vector>

namespace third_party_lib {

class Some {
 public:
    explicit Some(std::size_t value);

    // ...

 private:
    std::mutex mutex_;
};

}
```

Когда стандартные контейнеры не подходят

```
#include <mutex>
#include <vector>

namespace third_party_lib {

class Some {
 public:
    explicit Some(std::size_t value);

    // ...

 private:
    std::mutex mutex_;
};

}
```

Когда стандартные контейнеры не подходят

```cpp
#include <mutex>
#include <vector>

namespace third_party_lib {

class Some {
 public:
    explicit Some(std::size_t value);

    // ...

 private:
    std::mutex mutex_;
};

}
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count) {
        for (std::size_t i = 0; i < count; ++i) {
            clients_.emplace_back(100500);
        }
    }

    // ...
 private:
    std::list<third_party_lib::Some> clients_;
};
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count) {
        for (std::size_t i = 0; i < count; ++i) {
            clients_.emplace_back(100500);
        }
    }

    // ...
 private:
    std::list<third_party_lib::Some> clients_;
};
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count) {
        for (std::size_t i = 0; i < count; ++i) {
            clients_.emplace_back(100500);
        }
    }

    // ...
 private:
    std::list<third_party_lib::Some> clients_;
};
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count) {
        for (std::size_t i = 0; i < count; ++i) {
            clients_.emplace_back(100500);
        }
    }

    // ...
 private:
    std::list<third_party_lib::Some> clients_;
};
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count) {
        for (std::size_t i = 0; i < count; ++i) {
            clients_.emplace_back(100500);
        }
    }

    // ...
 private:
    std::list<third_party_lib::Some> clients_;
};
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count) {
        for (std::size_t i = 0; i < count; ++i) {
            clients_.emplace_back(100500);
        }
    }

    // ...
 private:
    std::list<third_party_lib::Some> clients_;
};
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count) {
        for (std::size_t i = 0; i < count; ++i) {
            clients_.emplace_back(100500);
        }
    }

    // ...
 private:
    std::vector<third_party_lib::Some> clients_;
};
```

Когда стандартные контейнеры не подходят

```
error: static assertion failed: result type must be constructible from
input type
   90 |       static_assert(is_constructible<_ValueType, _Tp>::value,
      |                                                    ^~~~~
```

Проблемы контейнеров

Проблемы контейнеров

std::vector

Проблемы контейнеров

std::vector

- Часто требует перемещаемость типа

Проблемы контейнеров

std::vector

- Часто требует перемещаемость типа
- Хранит capacity

Проблемы контейнеров

std::vector

- Часто требует перемещаемость типа
- Хранит capacity

std::unique_ptr<T[N]>

Проблемы контейнеров

std::vector

- Часто требует перемещаемость типа
- Хранит capacity

std::unique_ptr<T[N]>

- Требует знать размер на этапе компиляции

Проблемы контейнеров

std::vector

- Часто требует перемещаемость типа
- Хранит capacity

std::unique_ptr<T[N]>

- Требует знать размер на этапе компиляции
- Нет поэлементной инициализации

Проблемы контейнеров

std::vector

- Часто требует перемещаемость типа
- Хранит capacity

std::unique_ptr<T[N]>

- Требует знать размер на этапе компиляции
- Нет поэлементной инициализации

std::unique_ptr<T[]>

Проблемы контейнеров

std::vector

- Часто требует перемещаемость типа
- Хранит capacity

std::unique_ptr<T[N]>

- Требует знать размер на этапе компиляции
- Нет поэлементной инициализации

std::unique_ptr<T[]>

- Не помнит свой размер

Проблемы контейнеров

std::vector

- Часто требует перемещаемость типа
- Хранит capacity

std::unique_ptr<T[N]>

- Требует знать размер на этапе компиляции
- Нет поэлементной инициализации

std::unique_ptr<T[]>

- Не помнит свой размер
- Нет поэлементной инициализации

FixedArray

FixedArray

- Не требует перемещаемости от типа

FixedArray

- Не требует перемещаемости от типа
- Размер задаётся на runtime

FixedArray

- Не требует перемещаемости от типа
- Размер задаётся на runtime
- Хранит size

FixedArray

- Не требует перемещаемости от типа
- Размер задаётся на runtime
- Хранит size
- Может инициализировать все элементы одними значениями

FixedArray

- Не требует перемещаемости от типа
- Размер задаётся на runtime
- Хранит size
- Может инициализировать все элементы одними значениями
- Может инициализировать все элементы из функции

FixedArray

- Не требует перемещаемости от типа
- Размер задаётся на runtime
- Хранит size
- Может инициализировать все элементы одними значениями
- Может инициализировать все элементы из функции
- Continious layout

FixedArray

```
template <class T>
class FixedArray final {
 public:
  using iterator = T*;
  using const_iterator = const T*;

  FixedArray() = default;

  /// Make an array and initialize each element with "args"
  template <class... Args>
  explicit FixedArray(std::size_t size, Args&&... args);

  FixedArray(FixedArray&& other) noexcept;
  FixedArray& operator=(FixedArray&& other) noexcept;

  FixedArray(const FixedArray&) = delete;
  FixedArray& operator=(const FixedArray&) = delete;
```

FixedArray

```
  ~FixedArray();

  std::size_t size() const noexcept { return size_; }
  bool empty() const noexcept { return size_ == 0; }
  const T& operator[](std::size_t i) const noexcept;

  // ...
 private:
  T* storage_{nullptr};
  std::size_t size_{0};
};

}
```

FixedArray

```cpp
template <class T>
template <class... Args>
FixedArray<T>::FixedArray(std::size_t size, Args&&... args) : size_(size)
{
  if (size_ == 0) return;
  storage_ = std::allocator<T>{}.allocate(size_);
  auto* begin = data();
  try {
    for (auto* end = begin + size - 1; begin != end; ++begin) {
      new (begin) T(args...);
    }
    new (begin) T(std::forward<Args>(args)...);
  } catch (...) {
    std::destroy(data(), begin);
    std::allocator<T>{}.deallocate(storage_, size);
    throw;
  }
}
```

FixedArray

```
template <class T>
template <class... Args>
FixedArray<T>::FixedArray(std::size_t size, Args&&... args) : size_(size)
{
  if (size_ == 0) return;
  storage_ = std::allocator<T>{}.allocate(size_);
  auto* begin = data();
  try {
    for (auto* end = begin + size - 1; begin != end; ++begin) {
      new (begin) T(args...);
    }
    new (begin) T(std::forward<Args>(args)...);
  } catch (...) {
    std::destroy(data(), begin);
    std::allocator<T>{}.deallocate(storage_, size);
    throw;
  }
}
```

FixedArray

```
template <class T>
template <class... Args>
FixedArray<T>::FixedArray(std::size_t size, Args&&... args) : size_(size)
{
  if (size_ == 0) return;
  storage_ = std::allocator<T>{}.allocate(size_);
  auto* begin = data();
  try {
    for (auto* end = begin + size - 1; begin != end; ++begin) {
      new (begin) T(args...);
    }
    new (begin) T(std::forward<Args>(args)...);
  } catch (...) {
    std::destroy(data(), begin);
    std::allocator<T>{}.deallocate(storage_, size);
    throw;
  }
}
```

FixedArray

```
template <class T>
template <class... Args>
FixedArray<T>::FixedArray(std::size_t size, Args&&... args) : size_(size)
{
  if (size_ == 0) return;
  storage_ = std::allocator<T>{}.allocate(size_);
  auto* begin = data();
  try {
    for (auto* end = begin + size - 1; begin != end; ++begin) {
      new (begin) T(args...);
    }
    new (begin) T(std::forward<Args>(args)...);
  } catch (...) {
    std::destroy(data(), begin);
    std::allocator<T>{}.deallocate(storage_, size);
    throw;
  }
}
```

FixedArray

```
template <class T>
template <class... Args>
FixedArray<T>::FixedArray(std::size_t size, Args&&... args) : size_(size)
{
  if (size_ == 0) return;
  storage_ = std::allocator<T>{}.allocate(size_);
  auto* begin = data();
  try {
    for (auto* end = begin + size - 1; begin != end; ++begin) {
      new (begin) T(args...);
    }
    new (begin) T(std::forward<Args>(args)...);
  } catch (...) {
    std::destroy(data(), begin);
    std::allocator<T>{}.deallocate(storage_, size);
    throw;
  }
}
```

FixedArray

```cpp
template <class T>
template <class... Args>
FixedArray<T>::FixedArray(std::size_t size, Args&&... args) : size_(size)
{
  if (size_ == 0) return;
  storage_ = std::allocator<T>{}.allocate(size_);
  auto* begin = data();
  try {
    for (auto* end = begin + size - 1; begin != end; ++begin) {
      new (begin) T(args...);
    }
    new (begin) T(std::forward<Args>(args)...);
  } catch (...) {
    std::destroy(data(), begin);
    std::allocator<T>{}.deallocate(storage_, size);
    throw;
  }
}
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count) {
        for (std::size_t i = 0; i < count; ++i) {
            clients_.emplace_back(100500);
        }
    }

    // ...
 private:
    std::list<third_party_lib::Some> clients_;
};
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count)
      : clients_(count, 100500)
    {}

    // ...
 private:
    utils::FixedArray<third_party_lib::Some> clients_;
};
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count)
      : clients_(utils::GenerateFixedArray(count, [](std::size_t i) {
          return third_party_lib::Something::Create(i);
      }))
    {}

    // ...
 private:
    utils::FixedArray<third_party_lib::Something> clients_;
};
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count)
      : clients_(utils::GenerateFixedArray(count, [](std::size_t i) {
          return third_party_lib::Something::Create(i);
      }))
    {}

    // ...
 private:
    utils::FixedArray<third_party_lib::Something> clients_;
};
```

Когда стандартные контейнеры не подходят

```
class Component {
 public:

    Component(std::size_t count)
      : clients_(utils::GenerateFixedArray(count, [](std::size_t i) {
          return third_party_lib::Something::Create(i);
      }))
    {}

    // ...
 private:
    utils::FixedArray<third_party_lib::Something> clients_;
};
```

ODR-violation

Когда случается боль

Когда случается боль

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
  return Formatter(value);
}
```

Когда случается боль

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
  return Formatter(value);
}
```

Когда случается боль

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
  return Formatter(value);
}
```

Когда случается боль

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
  return Formatter(value);
}
```

Когда случается боль

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
  return Formatter(value);
}
```

Когда случается боль

Когда случается боль

```
// generated.hpp
struct MyCodegeneratedStructure { /*...*/ };
```

Когда случается боль

```cpp
// generated.hpp
struct MyCodegeneratedStructure { /*...*/ };

// my_writer.hpp
struct MyCodegeneratedStructure;
template <>
struct storages::postgres::io::CppToUserPg<MyCodegeneratedStructure> {
  // ...
};
```

Когда случается боль

```cpp
// generated.hpp
struct MyCodegeneratedStructure { /*...*/ };

// my_writer.hpp
struct MyCodegeneratedStructure;
template <>
struct storages::postgres::io::CppToUserPg<MyCodegeneratedStructure> {
  // ...
};

// a.cpp
#include <generated.hpp>
#include <my_writer.hpp>
```

Когда случается боль

```cpp
// generated.hpp
struct MyCodegeneratedStructure { /*...*/ };

// my_writer.hpp
struct MyCodegeneratedStructure;
template <>
struct storages::postgres::io::CppToUserPg<MyCodegeneratedStructure> {
  // ...
};

// a.cpp
#include <generated.hpp>
#include <my_writer.hpp>

// b.cpp
#include <generated.hpp>
```

Когда случается боль

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
  return Formatter(value);
}
```

Разный Formatter в зависимости от include

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
  return Formatter(value);
}
```

Разный Formatter в зависимости от include

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
  return Formatter(value);
}
```

Итак, проблема

Итак, проблема

- Имя у функции одно, а тела разные

Итак, проблема

- Имя у функции одно, а тела разные
- Линкер выбирает «случайное» тело функции

Итак, проблема

- Имя у функции одно, а тела разные
- Линкер выбирает «случайное» тело функции
- Ошибка проявляется неожиданно

Итак, проблема

- Имя у функции одно, а тела разные
- Линкер выбирает «случайное» тело функции
- Ошибка проявляется неожиданно:
  - Добавили новый cpp файл с правильными инклюдами

Итак, проблема

- Имя у функции одно, а тела разные
- Линкер выбирает «случайное» тело функции
- Ошибка проявляется неожиданно:
  - Добавили новый cpp файл с правильными инклюдами
  - Совершенно сторонний функционал развалило

Итак, проблема

- Имя у функции одно, а тела разные
- Линкер выбирает «случайное» тело функции
- Ошибка проявляется неожиданно:
  - Добавили новый cpp файл с правильными инклюдами
  - Совершенно сторонний функционал развалило
- Ошибку невероятно сложно диагностировать

Итак, проблема

- Имя у функции одно, а тела разные
- Линкер выбирает «случайное» тело функции
- Ошибка проявляется неожиданно:
  - Добавили новый cpp файл с правильными инклюдами
  - Совершенно сторонний функционал развалило
- Ошибку невероятно сложно диагностировать
  - Транзитивные инклюды

Итак, проблема

- Имя у функции одно, а тела разные
- Линкер выбирает «случайное» тело функции
- Ошибка проявляется неожиданно:
  - Добавили новый cpp файл с правильными инклюдами
  - Совершенно сторонний функционал развалило
- Ошибку невероятно сложно диагностировать
  - Транзитивные инклюды
  - Поменяв порядок инклюдов можно сломать сервис

Итак, проблема

- Имя у функции одно, а тела разные
- Линкер выбирает «случайное» тело функции
- Ошибка проявляется неожиданно:
  - Добавили новый cpp файл с правильными инклюдами
  - Совершенно сторонний функционал развалило
- Ошибку невероятно сложно диагностировать
  - Транзитивные инклюды
  - Поменяв порядок инклюдов можно сломать сервис
  - Нет подсказок компилятора

Итак, проблема

- Имя у функции одно, а тела разные
- Линкер выбирает «случайное» тело функции
- Ошибка проявляется неожиданно:
  - Добавили новый cpp файл с правильными инклюдами
  - Совершенно сторонний функционал развалило
- Ошибку невероятно сложно диагностировать
  - Транзитивные инклюды
  - Поменяв порядок инклюдов можно сломать сервис
  - Нет подсказок компилятора
  - Сторонние тулзы не ловят проблему

Когда случается боль, надо её диагностировать

Когда случается боль, надо её диагностировать

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
#ifndef NDEBUG
  detail::CheckForBufferWriterODR<T, Formatter>::content.RequireInstance();
#endif
  return Formatter(value);
}
```

Когда случается боль, надо её диагностировать

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
#ifndef NDEBUG
  detail::CheckForBufferWriterODR<T, Formatter>::content.RequireInstance();
#endif
  return Formatter(value);
}
```

Когда случается боль, надо её диагностировать

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
#ifndef NDEBUG
  detail::CheckForBufferWriterODR<T, Formatter>::content.RequireInstance();
#endif
  return Formatter(value);
}
```

Диагностика

```
#ifndef NDEBUG
class WritersRegistrator final {
 public:
  WritersRegistrator(std::type_index type, std::type_index formatter_type,
                     const char* base_file);
  void RequireInstance() const;
};

namespace {
template <class Type, class Writer>
struct CheckForBufferWriterODR final {
  static inline WritersRegistrator content{typeid(Type), typeid(Writer),
                                           __BASE_FILE__};
};
}  // namespace

#endif
```

Диагностика

```
#ifndef NDEBUG
class WritersRegistrator final {
 public:
  WritersRegistrator(std::type_index type, std::type_index formatter_type,
                     const char* base_file);
  void RequireInstance() const;
};

namespace {
template <class Type, class Writer>
struct CheckForBufferWriterODR final {
  static inline WritersRegistrator content{typeid(Type), typeid(Writer),
                                           __BASE_FILE__};
};
}  // namespace

#endif
```

Диагностика

```
#ifndef NDEBUG
class WritersRegistrator final {
 public:
  WritersRegistrator(std::type_index type, std::type_index formatter_type,
                     const char* base_file);
  void RequireInstance() const;
};

namespace {
template <class Type, class Writer>
struct CheckForBufferWriterODR final {
  static inline WritersRegistrator content{typeid(Type), typeid(Writer),
                                           __BASE_FILE__};
};
}  // namespace

#endif
```

Диагностика

```
#ifndef NDEBUG
class WritersRegistrator final {
 public:
  WritersRegistrator(std::type_index type, std::type_index formatter_type,
                     const char* base_file);
  void RequireInstance() const;
};

namespace {
template <class Type, class Writer>
struct CheckForBufferWriterODR final {
  static inline WritersRegistrator content{typeid(Type), typeid(Writer),
                                           __BASE_FILE__};
};
}  // namespace

#endif
```

Диагностика

```
#ifndef NDEBUG
class WritersRegistrator final {
 public:
  WritersRegistrator(std::type_index type, std::type_index formatter_type,
                     const char* base_file);
  void RequireInstance() const;
};

namespace {
template <class Type, class Writer>
struct CheckForBufferWriterODR final {
  static inline WritersRegistrator content{typeid(Type), typeid(Writer),
                                           __BASE_FILE__};
};
}  // namespace

#endif
```

Диагностика

```
#ifndef NDEBUG
class WritersRegistrator final {
 public:
  WritersRegistrator(std::type_index type, std::type_index formatter_type,
                     const char* base_file);
  void RequireInstance() const;
};

namespace {
template <class Type, class Writer>
struct CheckForBufferWriterODR final {
  static inline WritersRegistrator content{typeid(Type), typeid(Writer),
                                           __BASE_FILE__};
};
}  // namespace

#endif
```

Диагностика

```
#ifndef NDEBUG
class WritersRegistrator final {
 public:
  WritersRegistrator(std::type_index type, std::type_index formatter_type,
                     const char* base_file);
  void RequireInstance() const;
};

namespace {
template <class Type, class Writer>
struct CheckForBufferWriterODR final {
  static inline WritersRegistrator content{typeid(Type), typeid(Writer),
                                           __BASE_FILE__};
};
}  // namespace

#endif
```

Диагностика

```
#ifndef NDEBUG
class WritersRegistrator final {
 public:
  WritersRegistrator(std::type_index type, std::type_index formatter_type,
                     const char* base_file);
  void RequireInstance() const;
};

namespace {
template <class Type, class Writer>
struct CheckForBufferWriterODR final {
  static inline WritersRegistrator content{typeid(Type), typeid(Writer),
                                           __BASE_FILE__};
};
}  // namespace

#endif
```

Диагностика

```
#ifndef NDEBUG
class WritersRegistrator final {
 public:
  WritersRegistrator(std::type_index type, std::type_index formatter_type,
                     const char* base_file);
  void RequireInstance() const;
};

namespace {
template <class Type, class Writer>
struct CheckForBufferWriterODR final {
  static inline WritersRegistrator content{typeid(Type), typeid(Writer),
                                           __BASE_FILE__};
};
}  // namespace

#endif
```

Когда случается боль, надо её диагностировать

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
#ifndef NDEBUG
  detail::CheckForBufferWriterODR<T, Formatter>::content.RequireInstance();
#endif
  return Formatter(value);
}
```

Когда случается боль, надо её диагностировать

```
template <typename T>
typename traits::IO<T>::FormatterType BufferWriter(const T& value) {
  using Formatter = typename traits::IO<T>::FormatterType;
#ifndef NDEBUG
  detail::CheckForBufferWriterODR<T, Formatter>::content.RequireInstance();
#endif
  return Formatter(value);
}
```

Теперь проблема сразу видна:

```
Type 'MyCodegeneratedStructure' has conflicting instantiation
of formatters: 'pg::DefaultFormatter' vs
'storages::postgres::io::CppToUserPg<MyCodegeneratedStructure>' in base
files [b.cpp] vs [a.cpp]
```

Теперь проблема сразу видна:

```
Type 'MyCodegeneratedStructure' has conflicting instantiation
of formatters: 'pg::DefaultFormatter' vs
'storages::postgres::io::CppToUserPg<MyCodegeneratedStructure>' in base
files [b.cpp] vs [a.cpp]
```

Теперь проблема сразу видна:

```
Type 'MyCodegeneratedStructure' has conflicting instantiation
of formatters: 'pg::DefaultFormatter' vs
'storages::postgres::io::CppToUserPg<MyCodegeneratedStructure>' in base
files [b.cpp] vs [a.cpp]
```

Теперь проблема сразу видна:

```
Type 'MyCodegeneratedStructure' has conflicting instantiation
of formatters: 'pg::DefaultFormatter' vs
'storages::postgres::io::CppToUserPg<MyCodegeneratedStructure>' in base
files [b.cpp] vs [a.cpp]
```

Теперь проблема сразу видна:

```
Type 'MyCodegeneratedStructure' has conflicting instantiation
of formatters: 'pg::DefaultFormatter' vs
'storages::postgres::io::CppToUserPg<MyCodegeneratedStructure>' in base
files [b.cpp] vs [a.cpp]
```

Теперь проблема сразу видна:

```
Type 'MyCodegeneratedStructure' has conflicting instantiation
of formatters: 'pg::DefaultFormatter' vs
'storages::postgres::io::CppToUserPg<MyCodegeneratedStructure>' in base
files [b.cpp] vs [a.cpp]
```

Чудная BiMap

Задача

Задача

Нужен очень быстрый преобразователь string → enum

Задача

Нужен очень быстрый преобразователь string → enum

- std::unordered_map — медленный

Задача

Нужен очень быстрый преобразователь string → enum

- std::unordered_map — медленный

Дополнительные хотелки:

Задача

Нужен очень быстрый преобразователь string → enum

- std::unordered_map — медленный

Дополнительные хотелки:

- Преобразование enum → string

Задача

Нужен очень быстрый преобразователь string → enum

- std::unordered_map — медленный

Дополнительные хотелки:

- Преобразование enum → string
- Возможно получить все занчения enum

Задача

Нужен очень быстрый преобразователь string → enum

- std::unordered_map — медленный

Дополнительные хотелки:

- Преобразование enum → string
- Возможно получить все занчения enum
- Возможность получить все string

Первый подход

Первый подход

```
template <class Key, class Value, std::size_t N>
class ConsinitMap {
 public:
  constinit explicit ConsinitMap(std::pair<Key, Value>(&&map)[N]) {
    CompileTimeSlowSort(map);
    for (std::size_t i = 0; i < N; ++i) {
      keys_[i] = map[i].first;
      values_[i] = map[i].second;
    }
    CompileTimeAssertUnique(keys_);
  }
  constexpr bool Contains(const Key& key) const noexcept;
  constexpr const Value* FindOrNullptr(const Key& key) const noexcept;
  // ...
 private:
  Key keys_[N] = {};
  Value values_[N] = {};
};
```

Первый подход

```
template <class Key, class Value, std::size_t N>
class ConsinitMap {
 public:
  constinit explicit ConsinitMap(std::pair<Key, Value>(&&map)[N]) {
    CompileTimeSlowSort(map);
    for (std::size_t i = 0; i < N; ++i) {
      keys_[i] = map[i].first;
      values_[i] = map[i].second;
    }
    CompileTimeAssertUnique(keys_);
  }
  constexpr bool Contains(const Key& key) const noexcept;
  constexpr const Value* FindOrNullptr(const Key& key) const noexcept;
  // ...
 private:
  Key keys_[N] = {};
  Value values_[N] = {};
};
```

Первый подход

```
template <class Key, class Value, std::size_t N>
class ConsinitMap {
 public:
  constinit explicit ConsinitMap(std::pair<Key, Value>(&&map)[N]) {
    CompileTimeSlowSort(map);
    for (std::size_t i = 0; i < N; ++i) {
      keys_[i] = map[i].first;
      values_[i] = map[i].second;
    }
    CompileTimeAssertUnique(keys_);
  }
  constexpr bool Contains(const Key& key) const noexcept;
  constexpr const Value* FindOrNullptr(const Key& key) const noexcept;
  // ...
 private:
  Key keys_[N] = {};
  Value values_[N] = {};
};
```

Первый подход

```
template <class Key, class Value, std::size_t N>
class ConsinitMap {
 public:
  constinit explicit ConsinitMap(std::pair<Key, Value>(&&map)[N]) {
    CompileTimeSlowSort(map);
    for (std::size_t i = 0; i < N; ++i) {
      keys_[i] = map[i].first;
      values_[i] = map[i].second;
    }
    CompileTimeAssertUnique(keys_);
  }
  constexpr bool Contains(const Key& key) const noexcept;
  constexpr const Value* FindOrNullptr(const Key& key) const noexcept;
  // ...
 private:
  Key keys_[N] = {};
  Value values_[N] = {};
};
```

Первый подход

```
template <class Key, class Value, std::size_t N>
class ConsinitMap {
 public:
  constinit explicit ConsinitMap(std::pair<Key, Value>(&&map)[N]) {
    CompileTimeSlowSort(map);
    for (std::size_t i = 0; i < N; ++i) {
      keys_[i] = map[i].first;
      values_[i] = map[i].second;
    }
    CompileTimeAssertUnique(keys_);
  }
  constexpr bool Contains(const Key& key) const noexcept;
  constexpr const Value* FindOrNullptr(const Key& key) const noexcept;
  // ...
 private:
  Key keys_[N] = {};
  Value values_[N] = {};
};
```

Первый подход

```
template <class Key, class Value, std::size_t N>
class ConsinitMap {
 public:
  constinit explicit ConsinitMap(std::pair<Key, Value>(&&map)[N]) {
    CompileTimeSlowSort(map);
    for (std::size_t i = 0; i < N; ++i) {
      keys_[i] = map[i].first;
      values_[i] = map[i].second;
    }
    CompileTimeAssertUnique(keys_);
  }
  constexpr bool Contains(const Key& key) const noexcept;
  constexpr const Value* FindOrNullptr(const Key& key) const noexcept;
  // ...
 private:
  Key keys_[N] = {};
  Value values_[N] = {};
};
```

Альтернативное решение

Альтернативное решение

```
enum Color {
    Red, Orange, Yellow, UnknownColor,
};

Color color = llvm::StringSwitch<Color>(x)
  .Case("red", Red)
  .Case("orange", Orange)
  .Case("yellow", Yellow)
  .Default(UnknownColor);
```

Альтернативное решение

```
enum Color {
    Red, Orange, Yellow, UnknownColor,
};

Color color = llvm::StringSwitch<Color>(x)
  .Case("red", Red)
  .Case("orange", Orange)
  .Case("yellow", Yellow)
  .Default(UnknownColor);
```

Альтернативное решение

```
enum Color {
    Red, Orange, Yellow, UnknownColor,
};

Color color = llvm::StringSwitch<Color>(x)
  .Case("red", Red)
  .Case("orange", Orange)
  .Case("yellow", Yellow)
  .Default(UnknownColor);
```

Альтернативное решение под капотом

```
std::optional<Color> color;
if (!color && x == "red") {
    color = Red;
}
if (!color && x == "orange") {
    color = Orange;
}
if (!color && x == "yellow") {
    color = Yellow;
}

return color.value_or(UnknownColor);
```

Альтернативное решение под капотом

```
std::optional<Color> color;
if (!color && x == "red") {
    color = Red;
}
if (!color && x == "orange") {
    color = Orange;
}
if (!color && x == "yellow") {
    color = Yellow;
}

return color.value_or(UnknownColor);
```

Альтернативное решение под капотом

```
std::optional<Color> color;
if (!color && x == "red") {
    color = Red;
}
if (!color && x == "orange") {
    color = Orange;
}
if (!color && x == "yellow") {
    color = Yellow;
}

return color.value_or(UnknownColor);
```

Альтернативное решение под капотом

```
std::optional<Color> color;
if (!color && x == "red") {
    color = Red;
}
if (!color && x == "orange") {
    color = Orange;
}
if (!color && x == "yellow") {
    color = Yellow;
}

return color.value_or(UnknownColor);
```

Альтернативное решение под капотом

```
std::optional<Color> color;
if (!color && x == "red") {
    color = Red;
}
if (!color && x == "orange") {
    color = Orange;
}
if (!color && x == "yellow") {
    color = Yellow;
}

return color.value_or(UnknownColor);
```

Альтернативное решение под капотом

```
std::optional<Color> color;
if (!color && x == "red") {
    color = Red;
}
if (!color && x == "orange") {
    color = Orange;
}
if (!color && x == "yellow") {
    color = Yellow;
}

return color.value_or(UnknownColor);
```

Второй подход

Второй подход

```
constexpr utils::TrivialBiMap kMyEnumDescription3 = [](auto selector) {
    return selector()
    .Case("a", 9)
    .Case("ab", 10)
    .Case("abc", 11)
    .Case("abcd", 12)
    .Case("abcde", 13)
    .Case("abcdef", 14)
    .Case("abcdefg", 15)
    .Case("abcdefgz", 16)
    .Case("abcdefgzx", 17)
    .Case("abcdefgzxz", 18);
};

int StringCase3(std::string_view param) {
  return *kMyEnumDescription3.TryFind(param);
}
```

Второй подход

```
constexpr utils::TrivialBiMap kMyEnumDescription3 = [](auto selector) {
    return selector()
    .Case("a", 9)
    .Case("ab", 10)
    .Case("abc", 11)
    .Case("abcd", 12)
    .Case("abcde", 13)
    .Case("abcdef", 14)
    .Case("abcdefg", 15)
    .Case("abcdefgz", 16)
    .Case("abcdefgzx", 17)
    .Case("abcdefgzxz", 18);
};

int StringCase3(std::string_view param) {
  return *kMyEnumDescription3.TryFind(param);
}
```

Бенчмарк

```
-----------------------------------------------------------------------
Benchmark                             Time             CPU   Iterations
-----------------------------------------------------------------------
MappingSmallTrivialBiMap           12.2 ns         12.0 ns     57766546
MappingSmallUnordered               167 ns          164 ns      3844217

MappingMediumTrivialBiMap          19.5 ns         19.1 ns     33153861
MappingMediumUnordered              210 ns          207 ns      3130815

MappingHugeTrivialBiMap            72.2 ns         71.0 ns      9917775
MappingHugeUnordered                264 ns          264 ns      2584118

MappingHugeTrivialBiMapLast        19.1 ns         19.0 ns     37326680
MappingHugeUnorderedLast           23.0 ns         22.6 ns     30076471
```

Ассемблер

x86-64 gcc 12.1 (Editor #1) | C++ source #1

x86-64 gcc 12.1 | -std=c++17 -O2

```
1  StringCase3(std::basic_string_view<cha
2    cmp rdi, 1
3    je .L65
4    cmp rdi, 2
5    je .L66
6    cmp rdi, 3
7    je .L67
8    cmp rdi, 4
9    jne .L68
10   xor eax, eax
11   mov edx, 12
12   cmp DWORD PTR [rsi], 1684234849
13   cmove eax, edx
14   ret
15 .L68:
16   cmp rdi, 5
```

x86-64 clang 12.0.1 (Editor #1)

x86-64 clang 12.0.1 | -std=c++17 -O2

```
1  StringCase3(std::basic_string_view<char, std
2    mov eax, 9
3    add rdi, -1
4    cmp rdi, 9
5    ja .LBB0_21
6    jmp qword ptr [8*rdi + .LJTI0_0]
7  .LBB0_2:
8    mov eax, 9
9    cmp byte ptr [rsi], 97
10   ret
11 .LBB0_3:
12   movzx ecx, word ptr [rsi]
13   cmp ecx, 25185
14   je .LBB0_4
15 .LBB0_21:
16   ret
```

Компилятор сотимизировал

```
std::optional<Color> color;

if (!color && x == "red") {
    color = Red;
}
if (!color && x == "orange") {
    color = Orange;
}
if (!color && x == "yellow") {
    color = Yellow;
}

return color.value_or(UnknownColor);
```

Компилятор сотимизировал

```
std::optional<Color> color;

if (!color && x == "red") {
    color = Red;
}
if (!color && x == "orange") {
    color = Orange;
}
if (!color && x == "yellow") {
    color = Yellow;
}

return color.value_or(UnknownColor);
```

Компилятор сотимизировал

```
std::optional<Color> color;

if (!color && x == "red") {
    color = Red;
}
if (!color && x == "orange") {
    color = Orange;
}
if (!color && x == "yellow") {
    color = Yellow;
}

return color.value_or(UnknownColor);
```

Компилятор сотимизировал

```
std::optional<Color> color;

if (!color && x == "red") {
    color = Red;
}
if (!color && x == "orange") {
    color = Orange;
}
if (!color && x == "yellow") {
    color = Yellow;
}

return color.value_or(UnknownColor);
```

Компилятор сотимизировал

```
if (x == "red") {
    return Red;
}
if (x == "orange") {
    return Orange;
}
if (x == "yellow") {
    return Yellow;
}

return UnknownColor;
```

Компилятор сотимизировал

```
if (x.size() == 3 && x[0] == 'r' && x[1] == 'e' && x[2] == 'd') {
    return Red;
}
if (x.size() == 6 && x[0] == 'o' && x[1] == 'r' && x[2] == 'a' && x[3] ...) {
    return Orange;
}
if (x.size() == 6 && x[0] == 'y' && x[1] == 'e' && x[2] == 'l' && x[3] ...) {
    return Yellow;
}

return UnknownColor;
```

Компилятор сотимизировал

```
if (x.size() == 3 && x[0] == 'r' && x[1] == 'e' && x[2] == 'd') {
    return Red;
}
if (x.size() == 6 && x[0] == 'o' && x[1] == 'r' && x[2] == 'a' && x[3] ...) {
    return Orange;
}
if (x.size() == 6 && x[0] == 'y' && x[1] == 'e' && x[2] == 'l' && x[3] ...) {
    return Yellow;
}

return UnknownColor;
```

Компилятор сотимизировал

```
switch (x.size()) {
case 3:
  return (x[0] == 'r' && x[1] == 'e' && x[2] == 'd') ? Red : UnknownColor;

case 6:
  if (x[0] == 'o' && x[1] == 'r' && x[2] == 'a' && x[3] ...) {
    return Orange;
  }
  if (x[0] == 'y' && x[1] == 'e' && x[2] == 'l' && x[3] ...) {
    return Yellow;
  }
}
return UnknownColor;
```

Ассемблер

x86-64 gcc 12.1 (Editor #1) | C++ source #1

x86-64 gcc 12.1 | -std=c++17 -O2

```
1   StringCase3(std::basic_string_view<cha
2     cmp rdi, 1
3     je .L65
4     cmp rdi, 2
5     je .L66
6     cmp rdi, 3
7     je .L67
8     cmp rdi, 4
9     jne .L68
10    xor eax, eax
11    mov edx, 12
12    cmp DWORD PTR [rsi], 1684234849
13    cmove eax, edx
14    ret
15  .L68:
16    cmp rdi, 5
```

x86-64 clang 12.0.1 (Editor #1)

x86-64 clang 12.0.1 | -std=c++17 -O2

```
1   StringCase3(std::basic_string_view<char, std
2     mov eax, 9
3     add rdi, -1
4     cmp rdi, 9
5     ja .LBB0_21
6     jmp qword ptr [8*rdi + .LJTI0_0]
7   .LBB0_2:
8     mov eax, 9
9     cmp byte ptr [rsi], 97
10    ret
11  .LBB0_3:
12    movzx ecx, word ptr [rsi]
13    cmp ecx, 25185
14    je .LBB0_4
15  .LBB0_21:
16    ret
```

Ассемблер

x86-64 gcc 12.1 (Editor #1) | C++ source #1

x86-64 gcc 12.1 | -std=c++17 -O2

```
40      cmove eax, edx
41      ret
42    .L72:
43      cmp rdi, 8
44      jne .L53
45      movabs rax, 8820130980890370657
46      mov edx, 16
47      cmp QWORD PTR [rsi], rax
48      mov eax, 0
49      cmove eax, edx
50      ret
51    .L67:
52      cmp WORD PTR [rsi], 25185
53      je .L73
54    .L54:
55      cmp rdi, 10
```

x86-64 clang 12.0.1 (Editor #1)

x86-64 clang 12.0.1 | -std=c++17 -O2

```
56      mov eax, 15
57      ret
58    .LBB0_15:
59      movabs rcx, 8820130980890370657
60      cmp qword ptr [rsi], rcx
61      jne .LBB0_21
62      mov eax, 16
63      ret
64    .LBB0_19:
65      movabs rcx, 8820130980890370657
66      xor rcx, qword ptr [rsi]
67      movzx edx, byte ptr [rsi + 8]
68      xor rdx, 120
69      or rdx, rcx
70      jne .LBB0_21
71      mov eax, 17
```

Компилятор сотимизировал

```
switch (x.size()) {
case 3:
  return (x[0] == 'r' && x[1] == 'e' && x[2] == 'd') ? Red : UnknownColor;

case 6:
  if (x[0] == 'o' && x[1] == 'r' && x[2] == 'a' && x[3] ...) {
    return Orange;
  }
  if (x[0] == 'y' && x[1] == 'e' && x[2] == 'l' && x[3] ...) {
    return Yellow;
  }
}
return UnknownColor;
```

Компилятор сотимизировал

```
switch (x.size()) {
case 3:
  return (x[0] == 'r' && x[1] == 'e' && x[2] == 'd') ? Red : UnknownColor;

case 6:
  if (x[0] == 'o' && x[1] == 'r' && x[2] == 'a' && x[3] ...) {
    return Orange;
  }
  if (x[0] == 'y' && x[1] == 'e' && x[2] == 'l' && x[3] ...) {
    return Yellow;
  }
}
return UnknownColor;
```

Компилятор сотимизировал

```
switch (x.size()) {
case 3:
  return (x == 12731231283123) ? Red : UnknownColor;

case 6:
  if (x == 12312312312312312 3) {
    return Orange;
  }
  if (x == 78798798798798789) {
    return Yellow;
  }
}
return UnknownColor;
```

Ассемблер

x86-64 gcc 12.1 (Editor #1) | C++ source #1

x86-64 gcc 12.1 | -std=c++17 -O2

```
40      cmove eax, edx
41      ret
42    .L72:
43      cmp rdi, 8
44      jne .L53
45      movabs rax, 8820130980890370657
46      mov edx, 16
47      cmp QWORD PTR [rsi], rax
48      mov eax, 0
49      cmove eax, edx
50      ret
51    .L67:
52      cmp WORD PTR [rsi], 25185
53      je .L73
54    .L54:
55      cmp rdi, 10
```

x86-64 clang 12.0.1 (Editor #1)

x86-64 clang 12.0.1 | -std=c++17 -O2

```
56      mov eax, 15
57      ret
58    .LBB0_15:
59      movabs rcx, 8820130980890370657
60      cmp qword ptr [rsi], rcx
61      jne .LBB0_21
62      mov eax, 16
63      ret
64    .LBB0_19:
65      movabs rcx, 8820130980890370657
66      xor rcx, qword ptr [rsi]
67      movzx edx, byte ptr [rsi + 8]
68      xor rdx, 120
69      or rdx, rcx
70      jne .LBB0_21
71      mov eax, 17
```

Что мы получили

Нужен очень быстрый преобразователь string → enum

- std::unordered_map — медленный

Дополнительные хотелки:

- Преобразование enum → string
- Возможно получить все занчения enum
- Возможность получить все string

Что мы получили

Нужен очень быстрый преобразователь string → enum

- std::unordered_map — медленный

Дополнительные хотелки:

- Преобразование enum → string
- Возможно получить все занчения enum
- Возможность получить все string

Бонус:

Что мы получили

Нужен очень быстрый преобразователь string → enum

- std::unordered_map — медленный

Дополнительные хотелки:

- Преобразование enum → string
- Возможно получить все занчения enum
- Возможность получить все string

Бонус:

- constexpr

Что мы получили

Нужен очень быстрый преобразователь string → enum

- std::unordered_map — медленный

Дополнительные хотелки:

- Преобразование enum → string
- Возможно получить все занчения enum
- Возможность получить все string

Бонус:

- constexpr
- ICase

TrivialBiMap

TrivialBiMap

```
template <typename BuilderFunc>
class TrivialBiMap final {
  using TypesPair =
      std::invoke_result_t<const BuilderFunc&, impl::SwitchTypesDetector>;

 public:
  using First = typename TypesPair::first_type;
  using Second = typename TypesPair::second_type;

  constexpr TrivialBiMap(BuilderFunc func) noexcept;

  constexpr std::optional<Second> TryFindByFirst(First value) const noexcept {
    return func_(impl::SwitchByFirst<First, Second>{value}).Extract();
  }

 private:
  const BuilderFunc func_;
};
```

TrivialBiMap

```
template <typename BuilderFunc>
class TrivialBiMap final {
  using TypesPair =
      std::invoke_result_t<const BuilderFunc&, impl::SwitchTypesDetector>;

 public:
  using First = typename TypesPair::first_type;
  using Second = typename TypesPair::second_type;

  constexpr TrivialBiMap(BuilderFunc func) noexcept;

  constexpr std::optional<Second> TryFindByFirst(First value) const noexcept {
    return func_(impl::SwitchByFirst<First, Second>{value}).Extract();
  }

 private:
  const BuilderFunc func_;
};
```

TrivialBiMap

```
template <typename BuilderFunc>
class TrivialBiMap final {
  using TypesPair =
      std::invoke_result_t<const BuilderFunc&, impl::SwitchTypesDetector>;

 public:
  using First = typename TypesPair::first_type;
  using Second = typename TypesPair::second_type;

  constexpr TrivialBiMap(BuilderFunc func) noexcept;

  constexpr std::optional<Second> TryFindByFirst(First value) const noexcept {
    return func_(impl::SwitchByFirst<First, Second>{value}).Extract();
  }

 private:
  const BuilderFunc func_;
};
```

TrivialBiMap

```
template <typename First, typename Second>
class SwitchByFirst final {
 public:
  constexpr explicit SwitchByFirst(First search) noexcept : search_(search) {}

  constexpr SwitchByFirst& Case(First first, Second second) noexcept {
    if (!result_ && search_ == first) {
      result_.emplace(second);
    }
    return *this;
  }

  [[nodiscard]] constexpr std::optional<Second> Extract() noexcept {
    return result_;
  }
 private:
  const First search_;
  std::optional<Second> result_{};
};
```

TrivialBiMap

```
template <typename First, typename Second>
class SwitchByFirst final {
 public:
  constexpr explicit SwitchByFirst(First search) noexcept : search_(search) {}

  constexpr SwitchByFirst& Case(First first, Second second) noexcept {
    if (!result_ && search_ == first) {
      result_.emplace(second);
    }
    return *this;
  }

  [[nodiscard]] constexpr std::optional<Second> Extract() noexcept {
    return result_;
  }
 private:
  const First search_;
  std::optional<Second> result_{};
};
```

Второй подход

```
constexpr utils::TrivialBiMap kMyEnumDescription3 = [](auto selector) {
    return selector()
    .Case("a", 9)
    .Case("ab", 10)
    .Case("abc", 11)
    .Case("abcd", 12)
    .Case("abcde", 13)
    .Case("abcdef", 14)
    .Case("abcdefg", 15)
    .Case("abcdefgz", 16)
    .Case("abcdefgzx", 17)
    .Case("abcdefgzxz", 18);
};

int StringCase3(std::string_view param) {
  return *kMyEnumDescription3.TryFind(param);
}
```

TrivialBiMap

```
template <typename First, typename Second>
class SwitchByFirst final {
 public:
  constexpr explicit SwitchByFirst(First search) noexcept : search_(search) {}

  constexpr SwitchByFirst& Case(First first, Second second) noexcept {
    if (!result_ && search_ == first) {
      result_.emplace(second);
    }
    return *this;
  }

  [[nodiscard]] constexpr std::optional<Second> Extract() noexcept {
    return result_;
  }
 private:
  const First search_;
  std::optional<Second> result_{};
};
```

TrivialBiMap

```
template <typename First, typename Second>
class SwitchByFirst final {
 public:
  constexpr explicit SwitchByFirst(First search) noexcept : search_(search) {}

  constexpr SwitchByFirst& Case(First first, Second second) noexcept {
    if (!result_ && search_ == first) {
      result_.emplace(second);
    }
    return *this;
  }

  [[nodiscard]] constexpr std::optional<Second> Extract() noexcept {
    return result_;
  }
 private:
  const First search_;
  std::optional<Second> result_{};
};
```

TrivialBiMap

```
template <typename BuilderFunc>
class TrivialBiMap final {
  using TypesPair =
      std::invoke_result_t<const BuilderFunc&, impl::SwitchTypesDetector>;

 public:
  using First = typename TypesPair::first_type;
  using Second = typename TypesPair::second_type;

  constexpr TrivialBiMap(BuilderFunc func) noexcept;

  constexpr std::optional<Second> TryFindByFirst(First value) const noexcept {
    return func_(impl::SwitchByFirst<First, Second>{value}).Extract();
  }

 private:
  const BuilderFunc func_;
};
```

TrivialBiMap

```
template <typename BuilderFunc>
class TrivialBiMap final {
  using TypesPair =
      std::invoke_result_t<const BuilderFunc&, impl::SwitchTypesDetector>;

 public:
  using First = typename TypesPair::first_type;
  using Second = typename TypesPair::second_type;

  constexpr TrivialBiMap(BuilderFunc func) noexcept;

  constexpr std::optional<Second> TryFindByFirst(First value) const noexcept {
    return func_(impl::SwitchByFirst<First, Second>{value}).Extract();
  }

 private:
  const BuilderFunc func_;
};
```

TrivialBiMap

```
template <typename First, typename Second>
struct SwitchTypesDetected final {
  using first_type = First;
  using second_type = Second;
  constexpr SwitchTypesDetected& Case(First, Second) noexcept { return *this; }
};

struct SwitchTypesDetector final {
  template <typename First, typename Second>
  constexpr auto Case(First, Second) noexcept {
    using first_type =
        std::conditional_t<std::is_convertible_v<First, std::string_view>,
                           std::string_view, First>;
    using second_type =
        std::conditional_t<std::is_convertible_v<Second, std::string_view>,
                           std::string_view, Second>;
    return SwitchTypesDetected<first_type, second_type>{};
  }
};
```

TrivialBiMap

```
template <typename First, typename Second>
struct SwitchTypesDetected final {
  using first_type = First;
  using second_type = Second;
  constexpr SwitchTypesDetected& Case(First, Second) noexcept { return *this; }
};

struct SwitchTypesDetector final {
  template <typename First, typename Second>
  constexpr auto Case(First, Second) noexcept {
    using first_type =
        std::conditional_t<std::is_convertible_v<First, std::string_view>,
                           std::string_view, First>;
    using second_type =
        std::conditional_t<std::is_convertible_v<Second, std::string_view>,
                           std::string_view, Second>;
    return SwitchTypesDetected<first_type, second_type>{};
  }
};
```

TrivialBiMap

```
template <typename First, typename Second>
struct SwitchTypesDetected final {
  using first_type = First;
  using second_type = Second;
  constexpr SwitchTypesDetected& Case(First, Second) noexcept { return *this; }
};

struct SwitchTypesDetector final {
  template <typename First, typename Second>
  constexpr auto Case(First, Second) noexcept {
    using first_type =
        std::conditional_t<std::is_convertible_v<First, std::string_view>,
                           std::string_view, First>;
    using second_type =
        std::conditional_t<std::is_convertible_v<Second, std::string_view>,
                           std::string_view, Second>;
    return SwitchTypesDetected<first_type, second_type>{};
  }
};
```

TrivialBiMap

```
template <typename First, typename Second>
struct SwitchTypesDetected final {
  using first_type = First;
  using second_type = Second;
  constexpr SwitchTypesDetected& Case(First, Second) noexcept { return *this; }
};

struct SwitchTypesDetector final {
  template <typename First, typename Second>
  constexpr auto Case(First, Second) noexcept {
    using first_type =
        std::conditional_t<std::is_convertible_v<First, std::string_view>,
                           std::string_view, First>;
    using second_type =
        std::conditional_t<std::is_convertible_v<Second, std::string_view>,
                           std::string_view, Second>;
    return SwitchTypesDetected<first_type, second_type>{};
  }
};
```

TrivialBiMap

```
template <typename First, typename Second>
struct SwitchTypesDetected final {
  using first_type = First;
  using second_type = Second;
  constexpr SwitchTypesDetected& Case(First, Second) noexcept { return *this; }
};

struct SwitchTypesDetector final {
  template <typename First, typename Second>
  constexpr auto Case(First, Second) noexcept {
    using first_type =
        std::conditional_t<std::is_convertible_v<First, std::string_view>,
                           std::string_view, First>;
    using second_type =
        std::conditional_t<std::is_convertible_v<Second, std::string_view>,
                           std::string_view, Second>;
    return SwitchTypesDetected<first_type, second_type>{};
  }
};
```

TrivialBiMap

```
template <typename First, typename Second>
struct SwitchTypesDetected final {
  using first_type = First;
  using second_type = Second;
  constexpr SwitchTypesDetected& Case(First, Second) noexcept { return *this; }
};

struct SwitchTypesDetector final {
  template <typename First, typename Second>
  constexpr auto Case(First, Second) noexcept {
    using first_type =
        std::conditional_t<std::is_convertible_v<First, std::string_view>,
                           std::string_view, First>;
    using second_type =
        std::conditional_t<std::is_convertible_v<Second, std::string_view>,
                           std::string_view, Second>;
    return SwitchTypesDetected<first_type, second_type>{};
  }
};
```

Загадка про std::shared_ptr

Что это такое...

```
return std::shared_ptr<Logger>(
    std::shared_ptr<void>{},
    &GetNullLogger()
);
```

Что это такое...

```
return std::shared_ptr<Logger>(
    std::shared_ptr<void>{},
    &GetNullLogger()
);
```

Что это такое...

```
return std::shared_ptr<Logger>(
    std::shared_ptr<void>{},
    &GetNullLogger()
);
```

Что это такое...

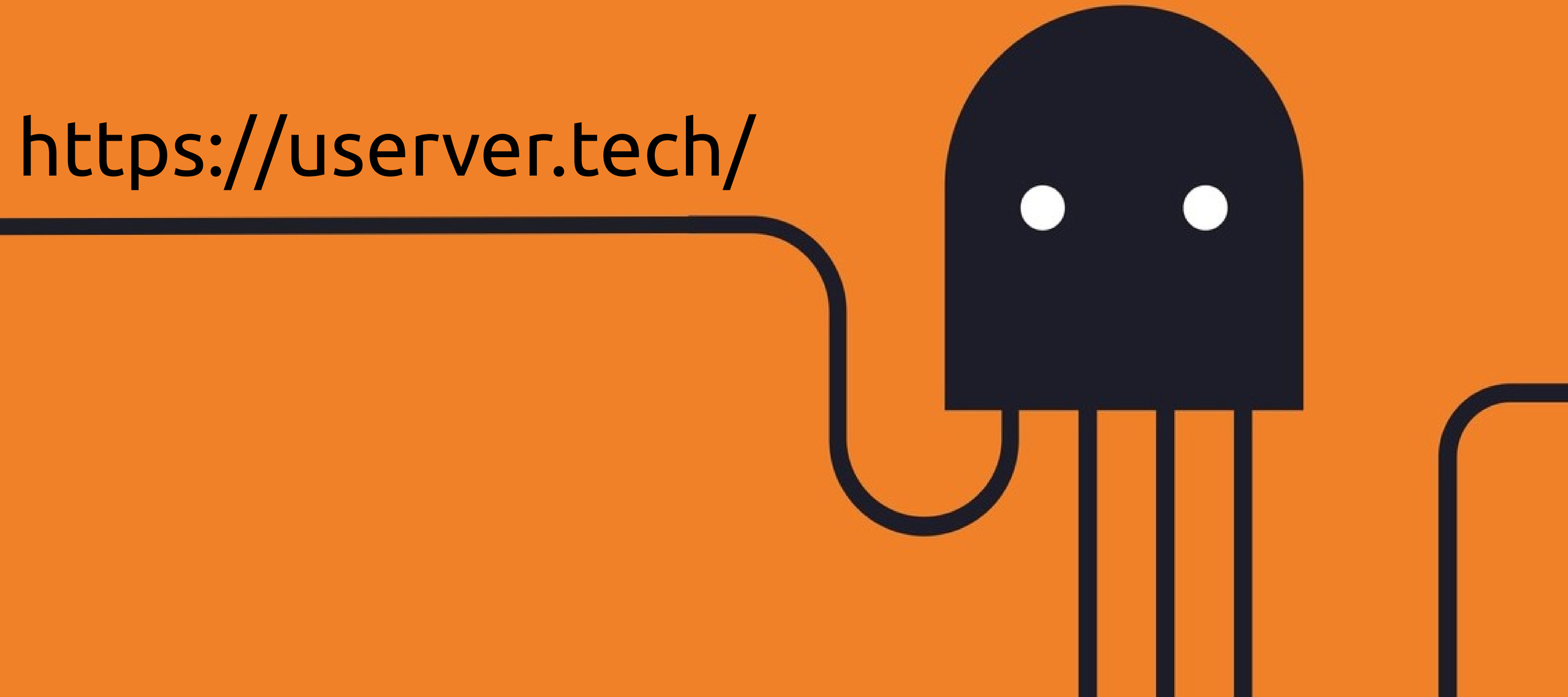
```
return std::shared_ptr<Logger>(
    std::shared_ptr<void>{},
    &GetNullLogger()
);
```

Что это такое...

```
Logger& GetNullLogger() noexcept {
  static NullLogger null_logger{};
  return null_logger;
}

std::shared_ptr<Logger> MakeNullLogger() {
  return std::shared_ptr<Logger>(
      std::shared_ptr<void>{},
      &GetNullLogger()
  );
}
```

https://userver.tech/



Спасибо

Полухин Антон

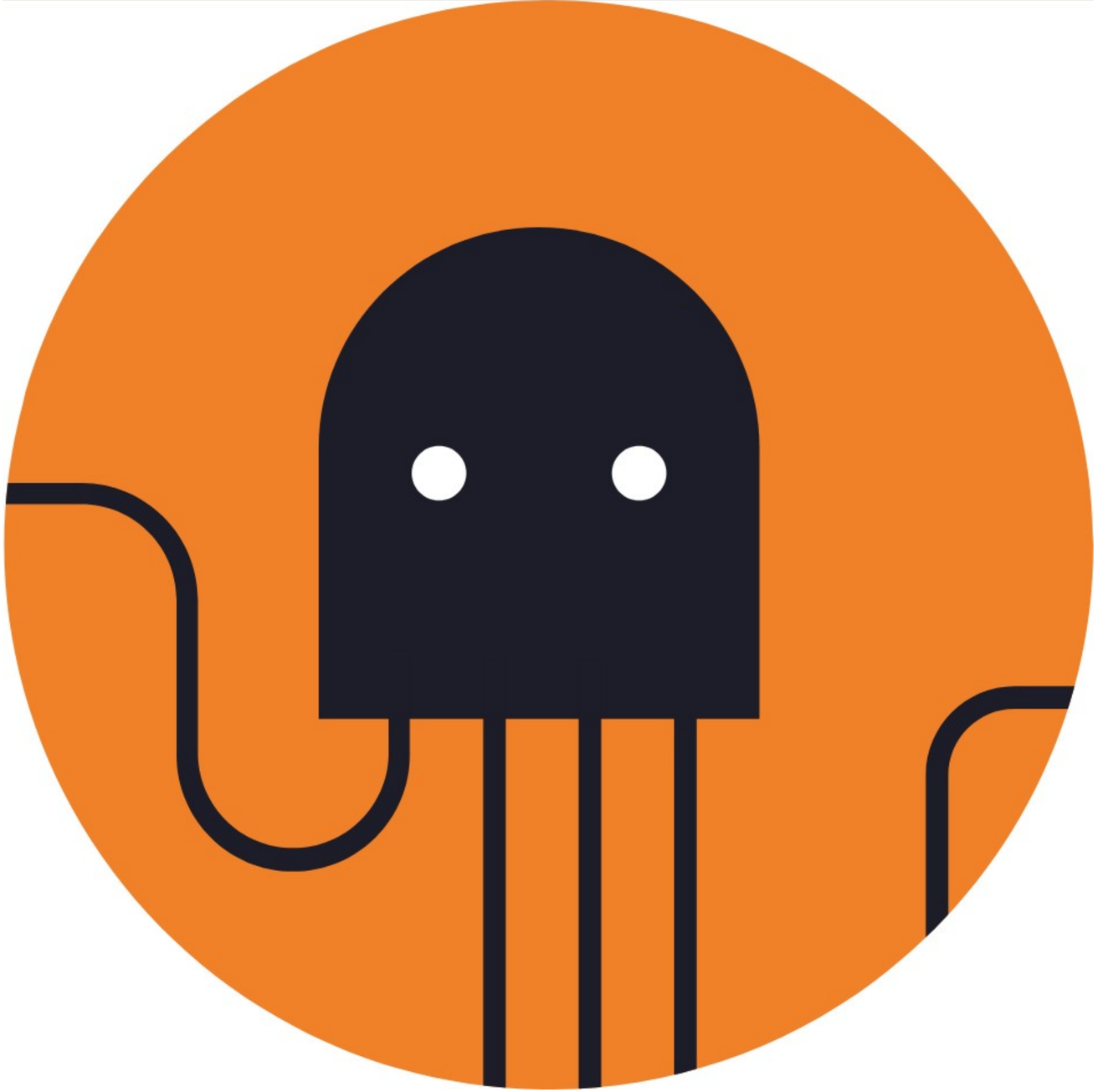
Эксперт-разработчик C++

antoshkka@gmail.com

antoshkka@yandex-team.ru

https://github.com/apolukhin

C++ РГ21 C++ РОССИЯ https://stdcpp.ru/



https://github.com/userver-framework

https://userver.tech/