C++20 Модули

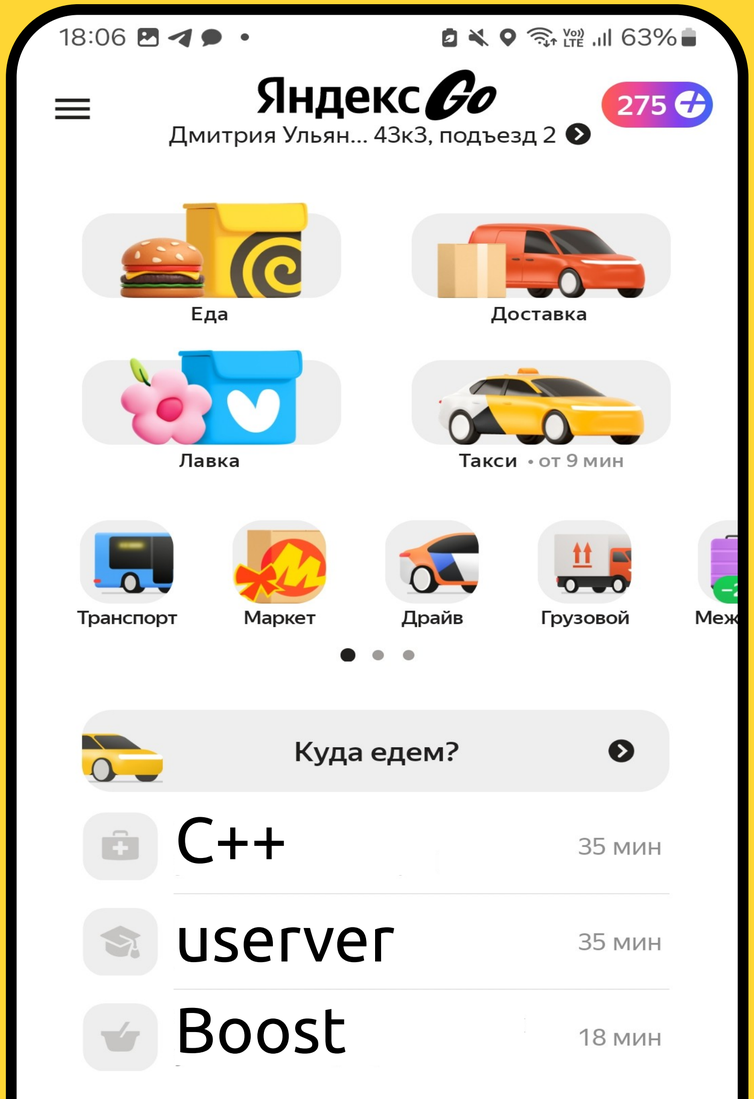
практическое внедрение

Антон Полухин
Эксперт разработчик C++

Техплатформа городских сервисов Яндекса

Содержание

- Что такое модули
- Как написать модуль для нового проекта
- Модуляризация имеющегося проекта
  - Boost.PFR
  - Libstdc++, libc++
  - Boost.PFR (вторая попытка)
  - Boost.Any, Boost.TypeIndex, ...



Какие именно модули?

Одноюнитные именованные модули

Какие именно модули?

Одноюнитные именованные модули

1. Не import <header/file.hpp>

Какие именно модули?

Одноюнитные именованные модули

1. Не import <header/file.hpp>
2. Не precompiled headers

Какие именно модули?

Одноюнитные именованные модули

1. Не import <header/file.hpp>
2. Не precompiled headers
3. Не мультиюнитные модули

Что такое модули?

Обычные #include

```
#include <iostream>


int main() { std::cout << "Hello" << std::endl;  }
```


Обычные #include

```
$ clang++-19 -E main.cpp -o preprocessed.txt
$ clang++-19 -M main.cpp -o includes.txt

$ wc -l includes.txt
180 includes.txt

$ ls -lh preprocessed.txt
-rw-rw-r-- 1 antoshkka antoshkka 839K Jun 11 14:11 preprocessed.txt
```


Обычные #include

```
$ clang++-19 -E main.cpp -o preprocessed.txt
$ clang++-19 -M main.cpp -o includes.txt

$ wc -l includes.txt
180 includes.txt

$ ls -lh preprocessed.txt
-rw-rw-r-- 1 antoshkka antoshkka 839K Jun 11 14:11 preprocessed.txt
```


Обычные #include

```
$ clang++-19 -E main.cpp -o preprocessed.txt
$ clang++-19 -M main.cpp -o includes.txt

$ wc -l includes.txt
180 includes.txt

$ ls -lh preprocessed.txt
-rw-rw-r-- 1 antoshkka antoshkka 839K Jun 11 14:11 preprocessed.txt
```


Обычные #include

```
import std;

int main() { std::cout << "Hello" << std::endl;  }
```


Обычные #include

```
$ clang++-19 -M main.cpp -o includes.txt
$ clang++-19 -E main.cpp -o preprocessed.txt

$ wc -l includes.txt
1 includes.txt

$ ls -lh preprocessed.txt
-rw-rw-r-- 1 antoshkka antoshkka 196 Jun 11 14:12 preprocessed.txt
```


Обычные #include

```
$ clang++-19 -M main.cpp -o includes.txt
$ clang++-19 -E main.cpp -o preprocessed.txt

$ wc -l includes.txt
1 includes.txt

$ ls -lh preprocessed.txt
-rw-rw-r-- 1 antoshkka antoshkka 196 Jun 11 14:12 preprocessed.txt
```


Обычные #include

```
$ clang++-19 -M main.cpp -o includes.txt
$ clang++-19 -E main.cpp -o preprocessed.txt

$ wc -l includes.txt
1 includes.txt

$ ls -lh preprocessed.txt
-rw-rw-r-- 1 antoshkka antoshkka 196 Jun 11 14:12 preprocessed.txt
```


И какое же ускорение?

Ускорение
(в теории)

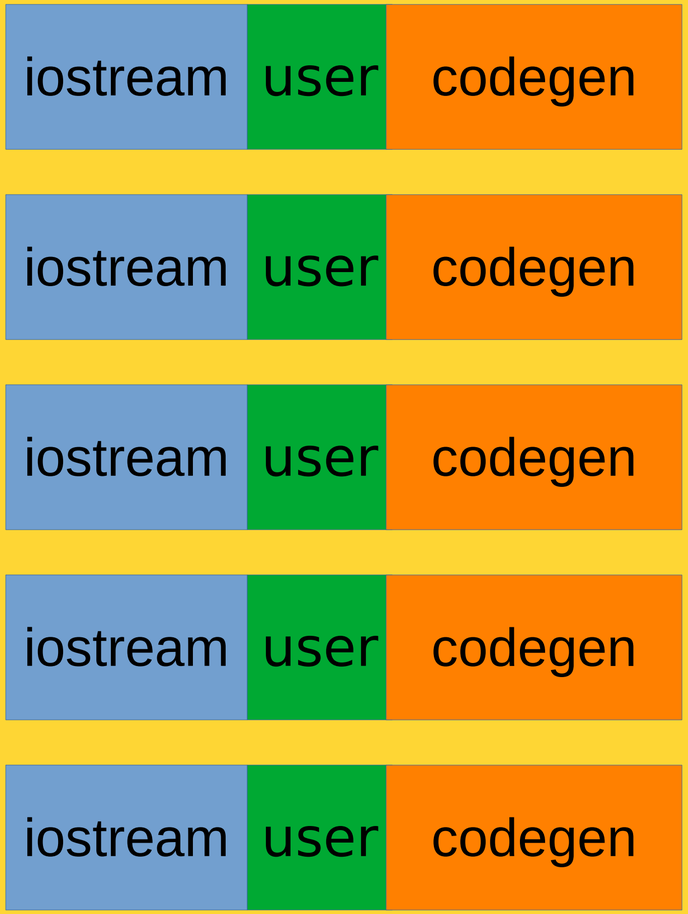
1. На каждый #include нужно открыть файл
2. Прочесть/подмапить его в память
3. Разбить на токены
4. Препроцессировать
5. Парсинг, построение AST
* ...

Сборка проекта без модулей

100 *.cpp файлов собирается на 5 ядрах

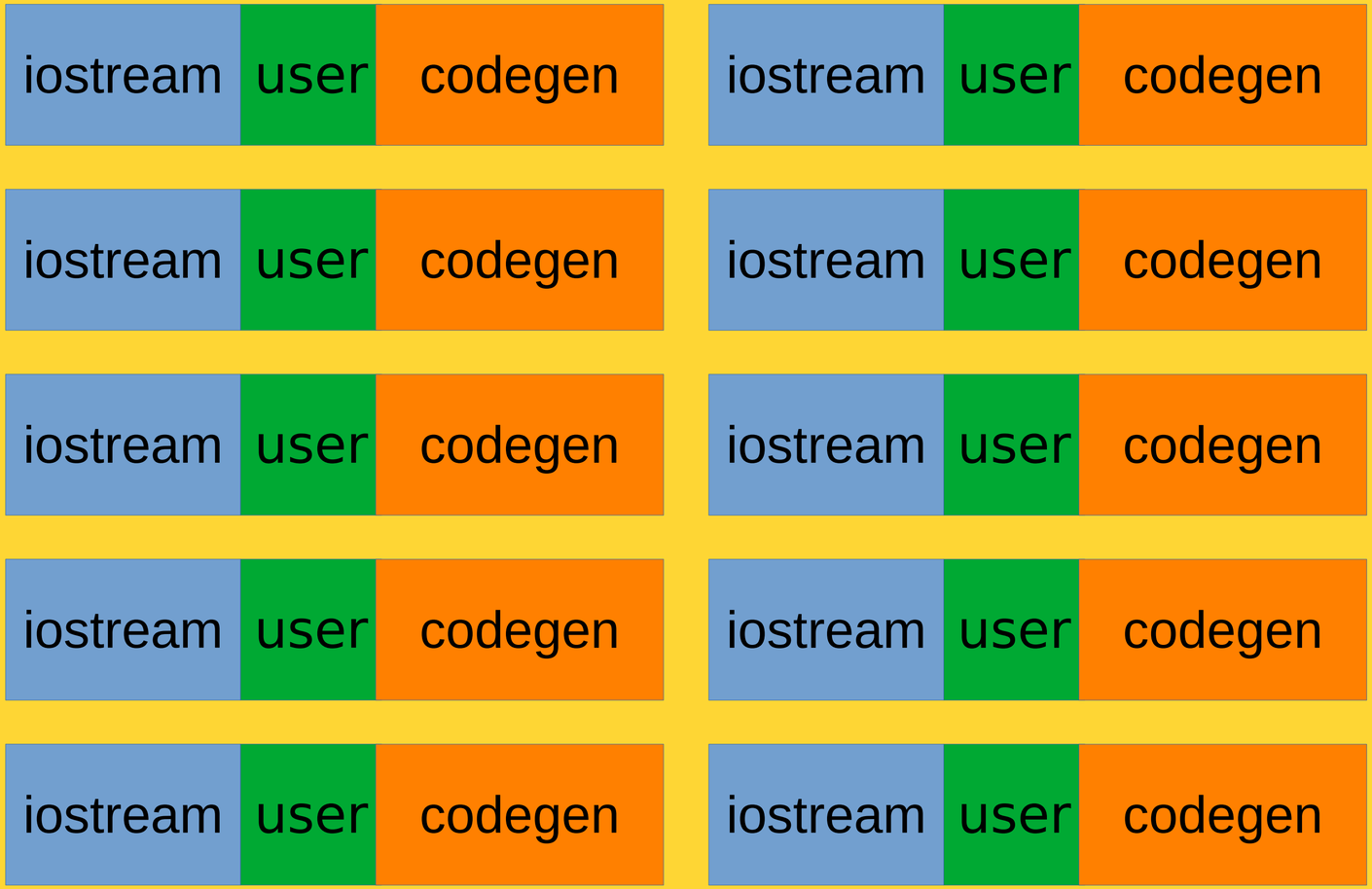
Сборка проекта без модулей

100 *.cpp файлов собирается на 5 ядрах



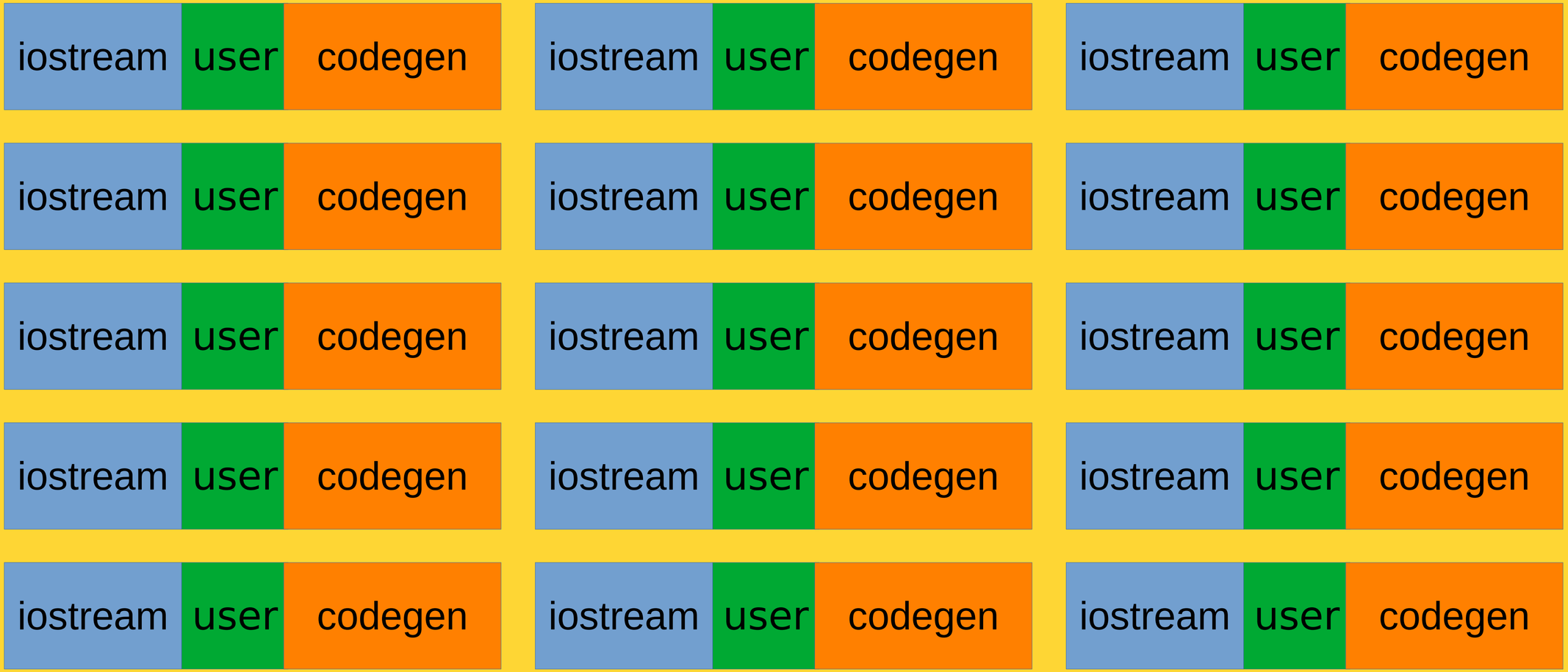
Сборка проекта без модулей

100 *.cpp файлов собирается на 5 ядрах

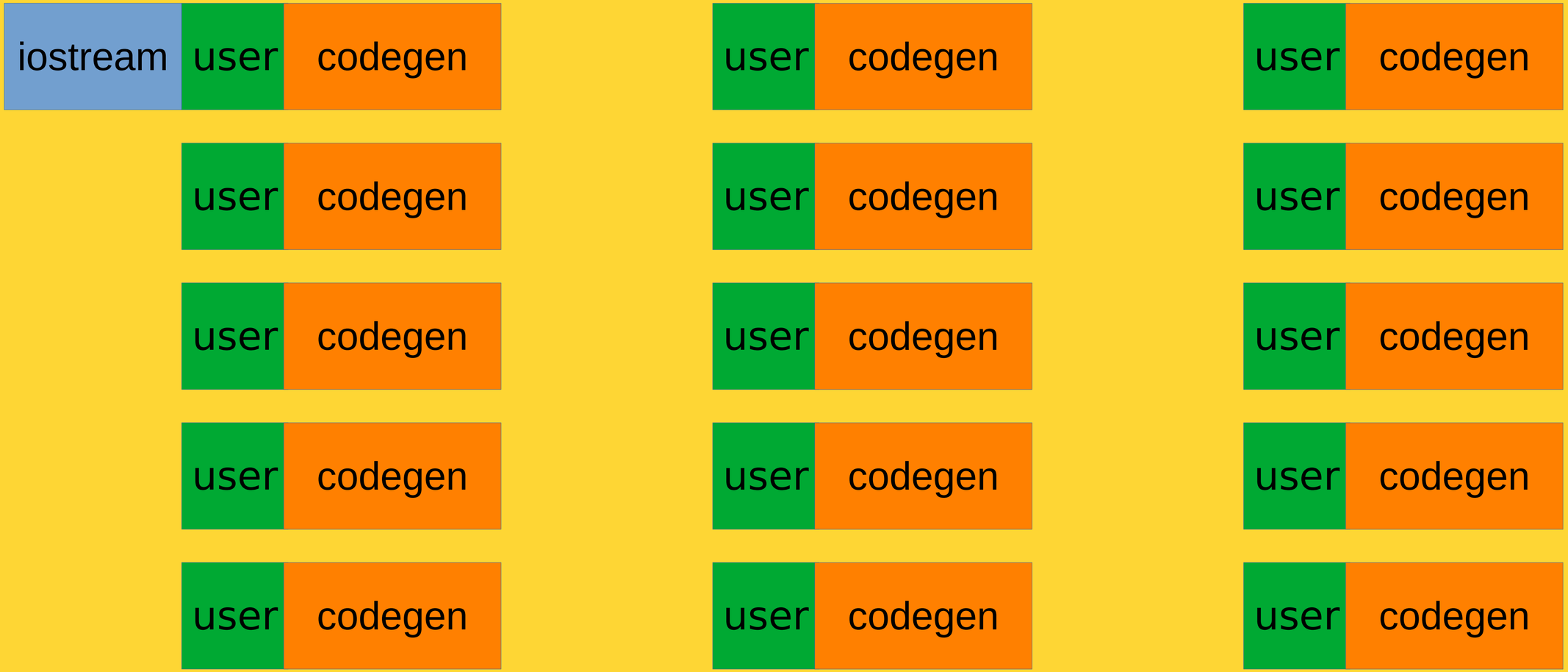


Сборка проекта без модулей

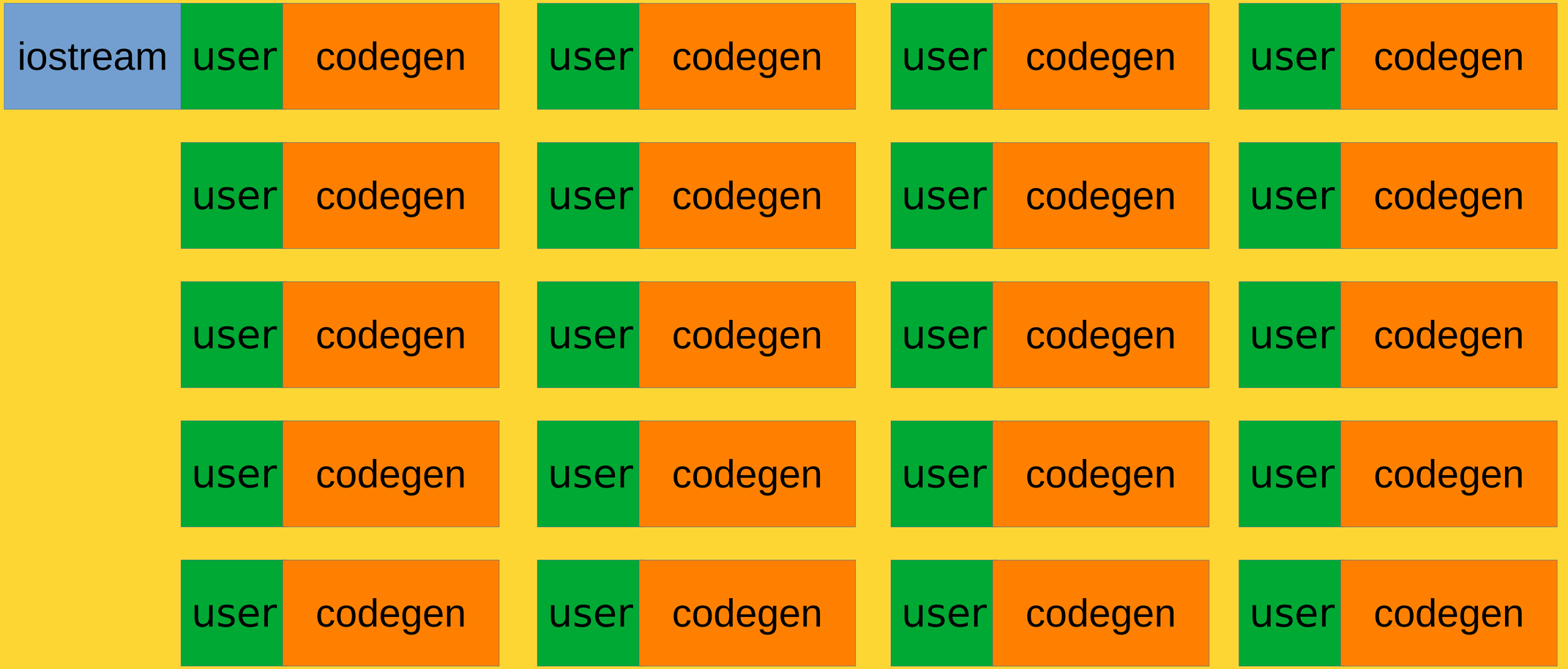
100 *.cpp файлов собирается на 5 ядрах



Сборка проекта с модулем

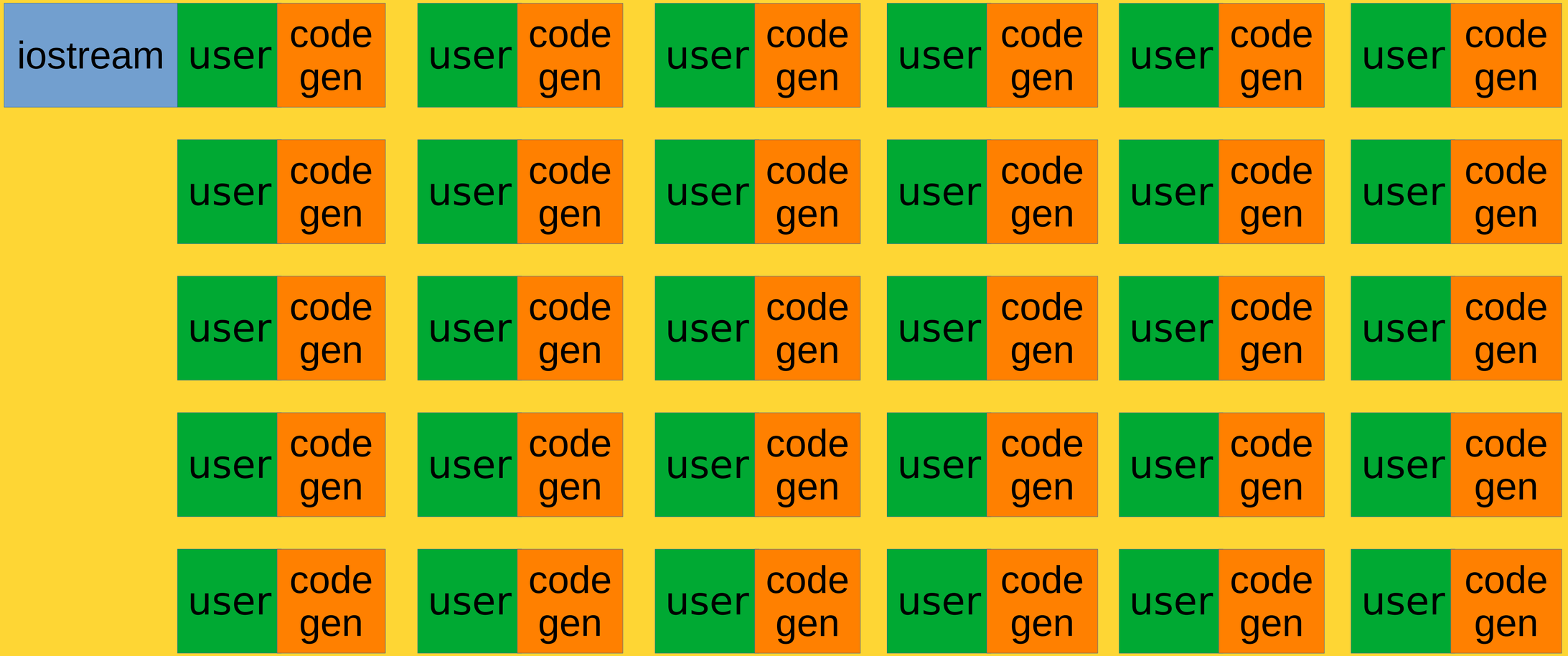


Сборка проекта с модулем



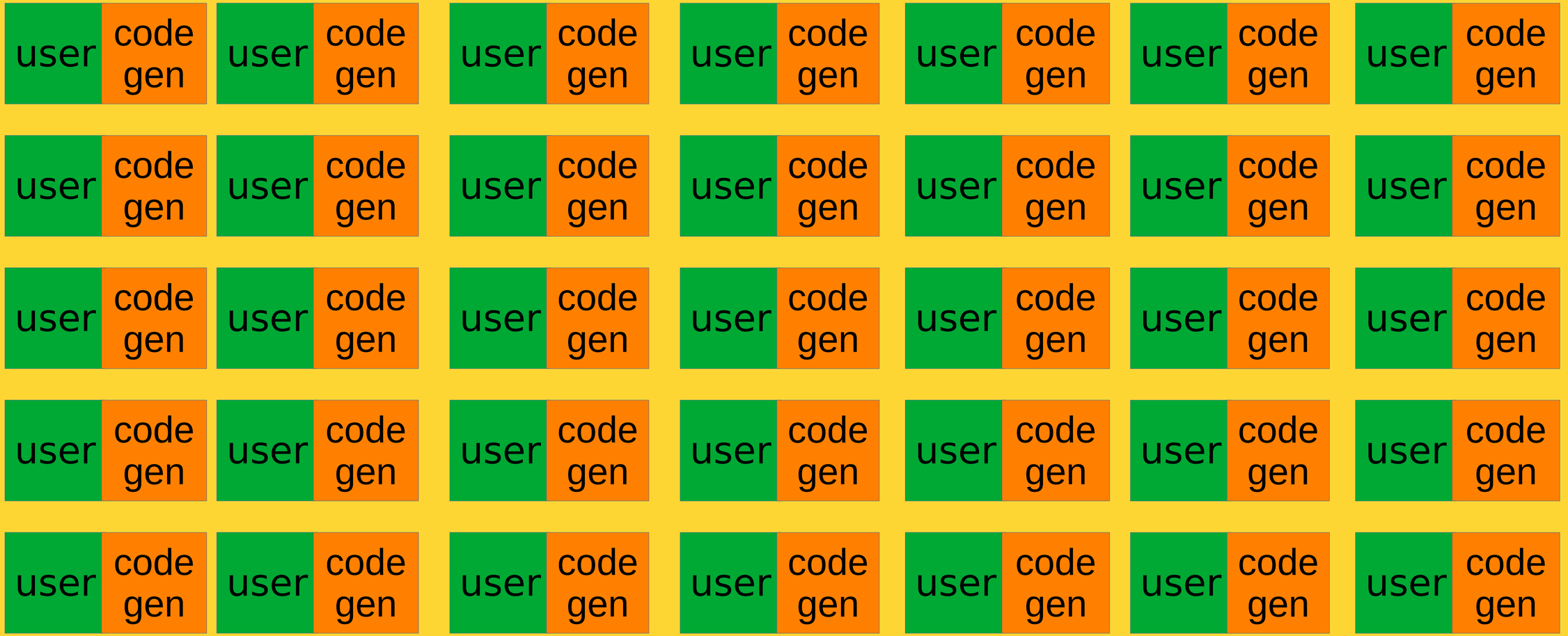
Сборка проекта с модулем

Дебаг



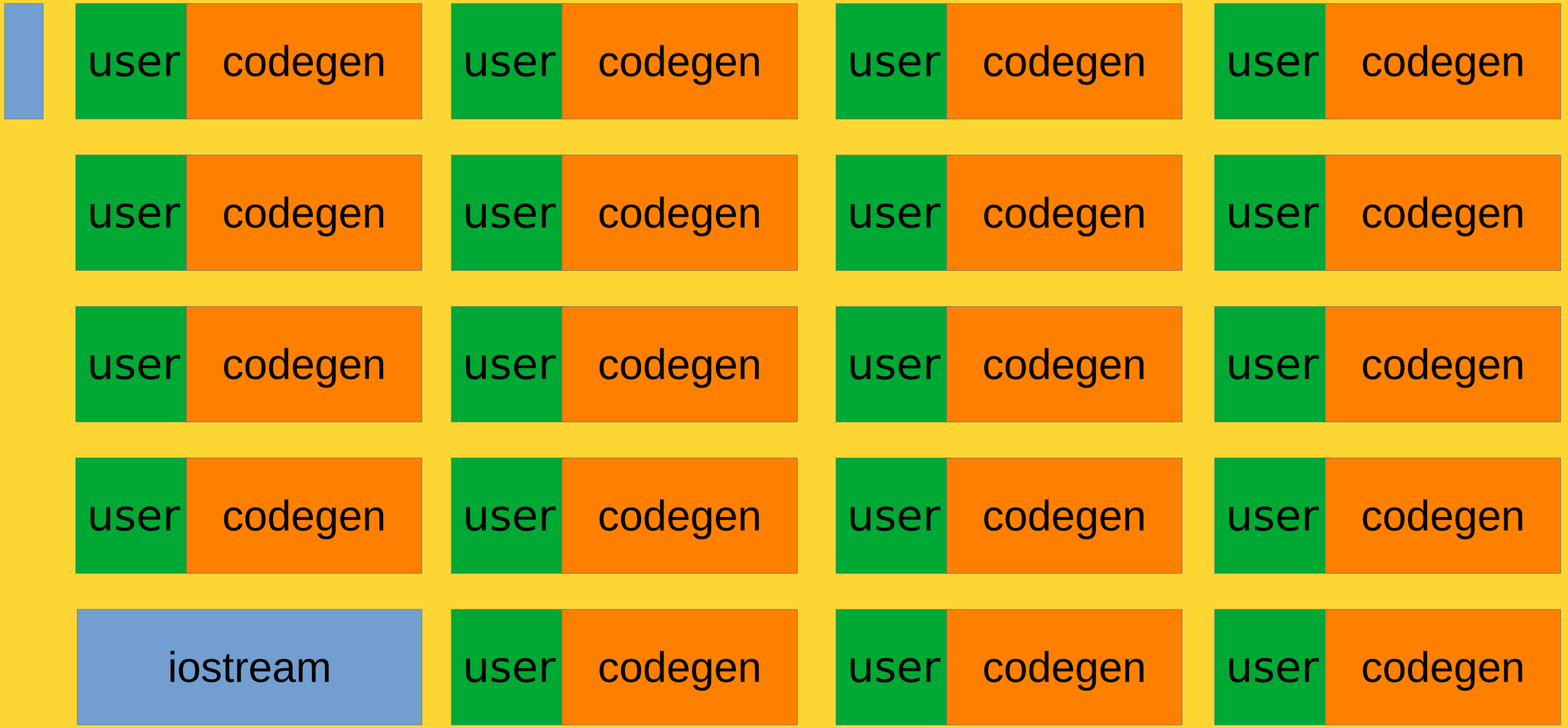
Сборка проекта с модулем

Дебаг + повторная сборка



Сборка проекта с модулем

Двухфазная сборка модуля (оч экспериментально)



Ускорение
(на практике)

	#include needed headers	**Import** needed headers	**import std**	**#include** all headers	**Import** all headers
“Hello world” (<iostream>)	0.87s	0.32s	0.08s	3.43s	0.62s
“Mix” (9 headers)	2.20s	0.77s	0.44s	3.53s	0.99s

Не на синтетике — ускорение x2-x3

Модуль для нового проекта

purview

```
export module my_module;

namespace impl {
  int get_42() { return 42; }
}

export int get_the_answer() { return impl::get_42(); }
```


purview

```
export module my_module;


namespace impl {
  int get_42() { return 42; }
}


export int get_the_answer() { return impl::get_42(); }
```


purview

```
export module my_module;


namespace impl {
  int get_42() { return 42; }
}


export int get_the_answer() { return impl::get_42(); }
```


purview

```
export module my_module;


namespace impl {
  int get_42() { return 42; }
}


export int get_the_answer() { return impl::get_42(); }
```


purview

```
export module my_module;

namespace impl {
  int get_42() { return 42; }
}

export int get_the_answer() { return impl::get_42(); }
```


purview

```
export module my_module;

namespace impl {
  int get_42() { return 42; }
}

export int get_the_answer() { return impl::get_42(); }
```

module linkage

external

purview

```
export module my_module;


namespace impl {
  int get_42() { return 42; }
}


export int get_the_answer() { return impl::get_42(); }
```


purview

```
export module my_module;

namespace impl {
  std::int32_t get_42() { return 42; }
}

export std::int32_t get_the_answer() { return impl::get_42(); }
```


ОЧЕНЬ ПЛОХО!!!!

```
export module my_module;
#include <cstdint>
namespace impl {
  std::int32_t get_42() { return 42; }
}

export std::int32_t get_the_answer() { return impl::get_42(); }
```


ОЧЕНЬ ПЛОХО!!!!

```
export module my_module;
#include <cstdint>

namespace impl {
  std::int32_t get_42() { return 42; }
}


export std::int32_t get_the_answer() { return impl::get_42(); }
```


(почти) глобальный фрагмент

```
#include <cstdint>

export module my_module;

namespace impl {
  std::int32_t get_42() { return 42; }
}

export std::int32_t get_the_answer() { return impl::get_42(); }
```


(почти) глобальный фрагмент

```
#include <cstdint>

export module my_module;

namespace impl {
  std::int32_t get_42() { return 42; }
}

export std::int32_t get_the_answer() { return impl::get_42(); }
```

Глобальный фрагмент

```
module;
#include <cstdint>

export module my_module;

namespace impl {
  std::int32_t get_42() { return 42; }
}

export std::int32_t get_the_answer() { return impl::get_42(); }
```

Глобальный фрагмент

```
module;
#include <cstdint>

export module my_module;
namespace impl {
  std::int32_t get_42();
}
export std::int32_t get_the_answer();
```

Global fragment

Purview

Primary Module Interface unit (PMI)

Как собирать модуль?

CMakeLists.txt

```
add_library(my_module_cmake)
target_sources(my_module_cmake PUBLIC
    FILE_SET modules_public TYPE CXX_MODULES FILES my_module.cppm
)
target_compile_features(my_module_cmake PUBLIC cxx_std_20)
```


CMakeLists.txt

```
add_library(my_module_cmake)
target_sources(my_module_cmake PUBLIC
    FILE_SET modules_public TYPE CXX_MODULES FILES my_module.cppm
)
target_compile_features(my_module_cmake PUBLIC cxx_std_20)
```


CMakeLists.txt

```
add_library(my_module_cmake)
target_sources(my_module_cmake PUBLIC
    FILE_SET modules_public TYPE CXX_MODULES FILES my_module.cppm
)
target_compile_features(my_module_cmake PUBLIC cxx_std_20)
```


CMakeLists.txt

```
add_library(my_module_cmake)
target_sources(my_module_cmake PUBLIC
    FILE_SET modules_public TYPE CXX_MODULES FILES my_module.cppm
)
target_compile_features(my_module_cmake PUBLIC cxx_std_20)
```


CMakeLists.txt

```
add_library(my_module_cmake)
target_sources(my_module_cmake PUBLIC
    FILE_SET modules_public TYPE CXX_MODULES FILES my_module.cppm
)
target_compile_features(my_module_cmake PUBLIC cxx_std_20)
```


CMakeLists.txt

```
add_library(my_module_cmake)
target_sources(my_module_cmake PUBLIC
    FILE_SET modules_public TYPE CXX_MODULES FILES my_module.cppm
)
target_compile_features(my_module_cmake PUBLIC cxx_std_20)
```


CMakeLists.txt

```
add_library(my_module_cmake)
target_sources(my_module_cmake PUBLIC
    FILE_SET modules_public TYPE CXX_MODULES FILES my_module.cppm
)
target_compile_features(my_module_cmake PUBLIC cxx_std_20)
```


Как использовать модуль?

CMakeLists.txt

```
target_link_libraries(${PROJECT_NAME} PRIVATE my_module_cmake)
```


CMakeLists.txt

```
target_link_libraries(${PROJECT_NAME} PRIVATE my_module_cmake)
```


CMakeLists.txt

```
target_link_libraries(${PROJECT_NAME} PRIVATE my_module_cmake)
```


main.cpp

```
import my_module;


int32_t main() { return get_the_answer(); }
```


А теперь, самая вкуснятина...

Имеющийся проект как модуль

libstdc++, libc++

```
module;
#include <bits/all.hpp>


export module std;


export namespace std {
  using std::all_of;
  using std::any_of;
  using std::none_of;
  // ...
```


Плюсы

Минусы

Плюсы

1. Все изменения локализованы в одном файле

Минусы

Плюсы

1. Все изменения локализованы в одном файле
2. ABI совместимо с #include

Минусы

Плюсы

1. Все изменения локализованы в одном файле
2. ABI совместимо с #include
3. Легко реализовать

Минусы

Плюсы

1. Все изменения локализованы в одном файле
2. ABI совместимо с #include
3. Легко реализовать

Минусы

1. Всё в глобальном фрагменте

Плюсы

1. Все изменения локализованы в одном файле
2. ABI совместимо с #include
3. Легко реализовать

Минусы

1. Всё в глобальном фрагменте
2. Легко «забыть» экспортировать новый функционал

Плюсы

1. Все изменения локализованы в одном файле
2. ABI совместимо с #include
3. Легко реализовать

Минусы

1. Всё в глобальном фрагменте
2. Легко «забыть» экспортировать новый функционал
3. Тяжёлая миграция для пользователей

Единый header для модуля и для include

Boost.PFR (попытка 1, header)

```cpp
#pragma once
#include <type_traits>
#include <boost/pfr/detail/fields_count.hpp>

namespace boost::pfr {
BOOST_PFR_BEGIN_MODULE_EXPORT
template <class T> constexpr auto tuple_size_v = tuple_size<T>::value;
BOOST_PFR_END_MODULE_EXPORT
}} // namespace boost::pfr
```


Boost.PFR (попытка 1, header)

```
#pragma once
#include <type_traits>
#include <boost/pfr/detail/fields_count.hpp>

namespace boost::pfr {
BOOST_PFR_BEGIN_MODULE_EXPORT
template <class T> constexpr auto tuple_size_v = tuple_size<T>::value;
BOOST_PFR_END_MODULE_EXPORT
}} // namespace boost::pfr
```


Boost.PFR (попытка 1, header)

```cpp
#pragma once
#include <type_traits>
#include <boost/pfr/detail/fields_count.hpp>

namespace boost::pfr {
BOOST_PFR_BEGIN_MODULE_EXPORT
template <class T> constexpr auto tuple_size_v = tuple_size<T>::value;
BOOST_PFR_END_MODULE_EXPORT
}} // namespace boost::pfr
```


Boost.PFR (попытка 1, cppm файл)

```
module;
#include <type_traits>
#define BOOST_PFR_BEGIN_MODULE_EXPORT export {
#define BOOST_PFR_END_MODULE_EXPORT }

export module boost.pfr;

...
```


Boost.PFR (попытка 1, cppm файл)

```
module;
#include <type_traits>
#define BOOST_PFR_BEGIN_MODULE_EXPORT export {
#define BOOST_PFR_END_MODULE_EXPORT }

export module boost.pfr;

...
```


Boost.PFR (попытка 1, cppm файл)

```
module;
#include <type_traits>
#define BOOST_PFR_BEGIN_MODULE_EXPORT export {
#define BOOST_PFR_END_MODULE_EXPORT }

export module boost.pfr;

...
```


Boost.PFR (попытка 1, cppm файл)

```
module;
#include <type_traits>
#define BOOST_PFR_BEGIN_MODULE_EXPORT export {
#define BOOST_PFR_END_MODULE_EXPORT }

export module boost.pfr;

...
```


Boost.PFR (попытка 1, cppm файл)

```
module;

...

export module boost.pfr;

#ifdef BOOST_PFR_ATTACH_TO_GLOBAL_MODULE
extern "C++" {
  #include <boost/pfr.hpp>
}
#else
#  include <boost/pfr.hpp>
#endif
```


Плюсы

Минусы

Плюсы

1. Сложно «забыть» экспортировать новый функционал

Минусы

Плюсы

1. Сложно «забыть» экспортировать новый функционал
2. ABI совместимо с #include при наличии такого требования

Минусы

Плюсы

1. Сложно «забыть» экспортировать новый функционал
2. ABI совместимо с #include при наличии такого требования
3. Легко реализовать

Минусы

Плюсы

1. Сложно «забыть» экспортировать новый функционал
2. ABI совместимо с #include при наличии такого требования
3. Легко реализовать

Минусы

1. Два ABI, две библиотеки

Плюсы

1. Сложно «забыть» экспортировать новый функционал
2. ABI совместимо с #include при наличии такого требования
3. Легко реализовать

Минусы

1. Два ABI, две библиотеки
2. Очень много приёмов «на грани фола»

Плюсы

1. Сложно «забыть» экспортировать новый функционал
2. ABI совместимо с #include при наличии такого требования
3. Легко реализовать

Минусы

1. Два ABI, две библиотеки
2. Очень много приёмов «на грани фола»
3. Тяжёлая миграция для пользователей

Единый header для модуля, и include с авто import

Boost.PFR (попытка 1, header)

```
#pragma once
#include <type_traits>
#include <boost/pfr/detail/fields_count.hpp>

namespace boost::pfr {
BOOST_PFR_BEGIN_MODULE_EXPORT
template <class T> constexpr auto tuple_size_v = tuple_size<T>::value;
BOOST_PFR_END_MODULE_EXPORT
}} // namespace boost::pfr
```


Boost.PFR (попытка 2, header)

```
#pragma once

#if defined(BOOST_USE_MODULES) && !defined(BOOST_PFR_INTERFACE_UNIT)

import boost.pfr;
#endif

#if !defined(BOOST_USE_MODULES) || defined(BOOST_PFR_INTERFACE_UNIT)
#include <boost/pfr/detail/fields_count.hpp>

#if !defined(BOOST_PFR_INTERFACE_UNIT)

#include <type_traits>
#endif

namespace boost::pfr { /* ... */ }
```


Boost.PFR (попытка 2, header)

```
#pragma once

#if defined(BOOST_USE_MODULES) && !defined(BOOST_PFR_INTERFACE_UNIT)
import boost.pfr;
#endif

#if !defined(BOOST_USE_MODULES) || defined(BOOST_PFR_INTERFACE_UNIT)
#include <boost/pfr/detail/fields_count.hpp>

#if !defined(BOOST_PFR_INTERFACE_UNIT)
#include <type_traits>
#endif

namespace boost::pfr { /* ... */ }
```


Boost.PFR (попытка 2, header)

```
#pragma once

#if defined(BOOST_USE_MODULES) && !defined(BOOST_PFR_INTERFACE_UNIT)

import boost.pfr;
#endif

#if !defined(BOOST_USE_MODULES) || defined(BOOST_PFR_INTERFACE_UNIT)
#include <boost/pfr/detail/fields_count.hpp>

#if !defined(BOOST_PFR_INTERFACE_UNIT)

#include <type_traits>
#endif

namespace boost::pfr { /* ... */ }
```


Boost.PFR (попытка 2, header)

```
#pragma once

#if defined(BOOST_USE_MODULES) && !defined(BOOST_PFR_INTERFACE_UNIT)

import boost.pfr;
#endif

#if !defined(BOOST_USE_MODULES) || defined(BOOST_PFR_INTERFACE_UNIT)
#include <boost/pfr/detail/fields_count.hpp>

#if !defined(BOOST_PFR_INTERFACE_UNIT)

#include <type_traits>
#endif

namespace boost::pfr { /* ... */ }
```


Boost.PFR (попытка 2, header)

```
#pragma once

#if defined(BOOST_USE_MODULES) && !defined(BOOST_PFR_INTERFACE_UNIT)

import boost.pfr;
#endif

#if !defined(BOOST_USE_MODULES) || defined(BOOST_PFR_INTERFACE_UNIT)
#include <boost/pfr/detail/fields_count.hpp>

#if !defined(BOOST_PFR_INTERFACE_UNIT)

#include <type_traits>
#endif

namespace boost::pfr { /* ... */ }
```


Boost.PFR (попытка 2, header)

```
#pragma once

#if defined(BOOST_USE_MODULES) && !defined(BOOST_PFR_INTERFACE_UNIT)
import boost.pfr;
#endif

#if !defined(BOOST_USE_MODULES) || defined(BOOST_PFR_INTERFACE_UNIT)
#include <boost/pfr/detail/fields_count.hpp>

#if !defined(BOOST_PFR_INTERFACE_UNIT)
#include <type_traits>
#endif

namespace boost::pfr { /* ... */ }
```


Boost.PFR (попытка 2, header)

```
#include <boost/pfr/detail/config.hpp>


#if !defined(BOOST_USE_MODULES) || defined(BOOST_PFR_INTERFACE_UNIT)
#include <boost/pfr/detail/fields_count.hpp>


#if !defined(BOOST_PFR_INTERFACE_UNIT)
#include <type_traits>
#endif
namespace boost::pfr { /* ... */ }
#endif // defined(BOOST_USE_MODULES) && !defined(BOOST_PFR_INTERFACE_UNIT)
```


Boost.PFR (попытка 2, cppm файл)

```
module;
#include <type_traits>
#define BOOST_PFR_BEGIN_MODULE_EXPORT export {
#define BOOST_PFR_END_MODULE_EXPORT }
#define BOOST_PFR_INTERFACE_UNIT

export module boost.pfr;
#include <boost/pfr.hpp>
```


Boost.PFR (попытка 2, cppm файл)

```
module;
#include <type_traits>
#define BOOST_PFR_BEGIN_MODULE_EXPORT export {
#define BOOST_PFR_END_MODULE_EXPORT }
#define BOOST_PFR_INTERFACE_UNIT

export module boost.pfr;
#include <boost/pfr.hpp>
```


Boost.PFR (попытка 2, cppm файл)

```
module;
#include <type_traits>
#define BOOST_PFR_BEGIN_MODULE_EXPORT export {
#define BOOST_PFR_END_MODULE_EXPORT }
#define BOOST_PFR_INTERFACE_UNIT

export module boost.pfr;
#include <boost/pfr.hpp>
```


Плюсы

Минусы

Плюсы

1. Сложно «забыть» экспортировать новый функционал

Минусы

Плюсы

1. Сложно «забыть» экспортировать новый функционал
2. ABI совместимо с #include при наличии такого требования в рамках одной настройки сборки

Минусы

Плюсы

1. Сложно «забыть» экспортировать новый функционал
2. ABI совместимо с #include при наличии такого требования в рамках одной настройки сборки
3. Простейшая миграция для пользователей

Минусы

Плюсы

1. Сложно «забыть» экспортировать новый функционал
2. ABI совместимо с #include при наличии такого требования в рамках одной настройки сборки
3. Простейшая миграция для пользователей

Минусы

1. Очень муторно править код

Больше примеров

import std;

```
module;
#include <type_traits>
#define BOOST_PFR_BEGIN_MODULE_EXPORT export {
#define BOOST_PFR_END_MODULE_EXPORT }
#define BOOST_PFR_INTERFACE_UNIT

export module boost.pfr;
#include <boost/pfr.hpp>
```


import std;

```
module;
#include <type_traits>
#define BOOST_PFR_BEGIN_MODULE_EXPORT export {
#define BOOST_PFR_END_MODULE_EXPORT }
#define BOOST_PFR_INTERFACE_UNIT

export module boost.pfr;
#include <boost/pfr.hpp>
```


import std;

```
module;
#include <any>
#include <array>
#include <limits>
#include <string>
#include <string_view>
#include <type_traits>
#include <tuple>
#include <utility>
#include <variant>
```


import std;

```
module;

#define BOOST_PFR_BEGIN_MODULE_EXPORT export {
#define BOOST_PFR_END_MODULE_EXPORT }
#define BOOST_PFR_INTERFACE_UNIT

export module boost.pfr;
import std;
#include <boost/pfr.hpp>
```


import std;

```
module;
#ifndef BOOST_PFR_USE_STD_MODULE
#include <type_traits>
#endif

export module boost.pfr;
#ifdef BOOST_PFR_USE_STD_MODULE
import std;
#endif
```


Связанные модули

Boost.Any (header)

```
#include <boost/any/detail/config.hpp>


#if !defined(BOOST_USE_MODULES) || defined(BOOST_ANY_INTERFACE_UNIT)


#if !defined(BOOST_ANY_INTERFACE_UNIT)
#include <boost/type_index.hpp>
#include <type_traits>
#endif
namespace boost::anys { /* ... */ }
#endif // defined(BOOST_USE_MODULES) && !defined(BOOST_ANY_INTERFACE_UNIT)
```

Boost.Any (header)

```
#include <boost/any/detail/config.hpp>


#if !defined(BOOST_USE_MODULES) || defined(BOOST_ANY_INTERFACE_UNIT)


#if !defined(BOOST_ANY_INTERFACE_UNIT)
#include <boost/type_index.hpp>
#include <type_traits>
#endif
namespace boost::anys { /* ... */ }
#endif // defined(BOOST_USE_MODULES) && !defined(BOOST_ANY_INTERFACE_UNIT)
```


Boost.Any (header)

```
#include <boost/any/detail/config.hpp>


#if !defined(BOOST_USE_MODULES) || defined(BOOST_ANY_INTERFACE_UNIT)


#if !defined(BOOST_ANY_INTERFACE_UNIT)
#include <boost/type_index.hpp>
#include <type_traits>
#endif
namespace boost::anys { /* ... */ }
#endif // defined(BOOST_USE_MODULES) && !defined(BOOST_ANY_INTERFACE_UNIT)
```


Boost.Any (cppm файл)

```
module;
#include <boost/type_index.hpp>
#include <type_traits>
#define BOOST_ANY_BEGIN_MODULE_EXPORT export {
#define BOOST_ANY_END_MODULE_EXPORT }
#define BOOST_ANY_INTERFACE_UNIT

export module boost.any;
#include <boost/any.hpp>
#include <boost/any/basic_any.hpp>
```


Boost.Any (cppm файл)

```
module;
#include <boost/type_index.hpp>
#include <type_traits>
#define BOOST_ANY_BEGIN_MODULE_EXPORT export {
#define BOOST_ANY_END_MODULE_EXPORT }
#define BOOST_ANY_INTERFACE_UNIT

export module boost.any;
#include <boost/any.hpp>
#include <boost/any/basic_any.hpp>
```


Плюсы

1. Сложно «забыть» экспортировать новый функционал
2. ABI совместимо с #include при наличии такого требования в рамках одной настройки сборки
3. Простейшая миграция для пользователей

Минусы

1. Очень муторно править код

Плюсы

1. Сложно «забыть» экспортировать новый функционал
2. ABI совместимо с #include при наличии такого требования в рамках одной настройки сборки
3. Простейшая миграция для пользователей
4. Дружелюбно к зависящим модулям, не требует синхронизаций усилий разработчиков

Минусы

1. Очень муторно править код

Неожиданные проблемы

Проблемы с модулями

(для разработчиков проекта)

Проблемы с модулями
(для разработчиков проекта)

1. Сложно протестировать impl

Проблемы с модулями
(для разработчиков проекта)

1. Сложно протестировать impl

2. Надо тестировать модуль

Проблемы с модулями
(для разработчиков проекта)

1. Сложно протестировать impl
2. Надо тестировать модуль
3. Макросы не экспортируются

Проблемы с модулями
(для разработчиков проекта)

1. Сложно протестировать impl
2. Надо тестировать модуль
3. Макросы не экспортируются
4. ADL не работает для явно не экспортированных частей

Проблемы с модулями
(для разработчиков проекта)

1. Сложно протестировать impl
2. Надо тестировать модуль
3. Макросы не экспортируются
4. ADL не работает для явно не экспортированных частей:

```
namespace impl {
struct foo;
std::ostream& operator<<(std::ostream&, const foo&);
}

export impl::foo bar();
...
std::cout << bar();
```

Проблемы с модулями
(для разработчиков проекта)

1. Сложно протестировать impl
2. Надо тестировать модуль
3. Макросы не экспортируются
4. ADL не работает для явно не экспортированных частей
5. Компиляторы чуть сыроваты

Проблемы с модулями
(для разработчиков проекта)

1. Сложно протестировать impl
2. Надо тестировать модуль
3. Макросы не экспортируются
4. ADL не работает для явно не экспортированных частей
5. Компиляторы чуть сыроваты
6. Системы сборки чуть сыроваты

Проблемы с модулями
(для разработчиков проекта)

1. Сложно протестировать impl
2. Надо тестировать модуль
3. Макросы не экспортируются
4. ADL не работает для явно не экспортированных частей
5. Компиляторы чуть сыроваты
6. Системы сборки чуть сыроваты
7. ?? Манглинг меняется ??

Попробуйте в своём проекте!

Спасибо

Полухин Антон

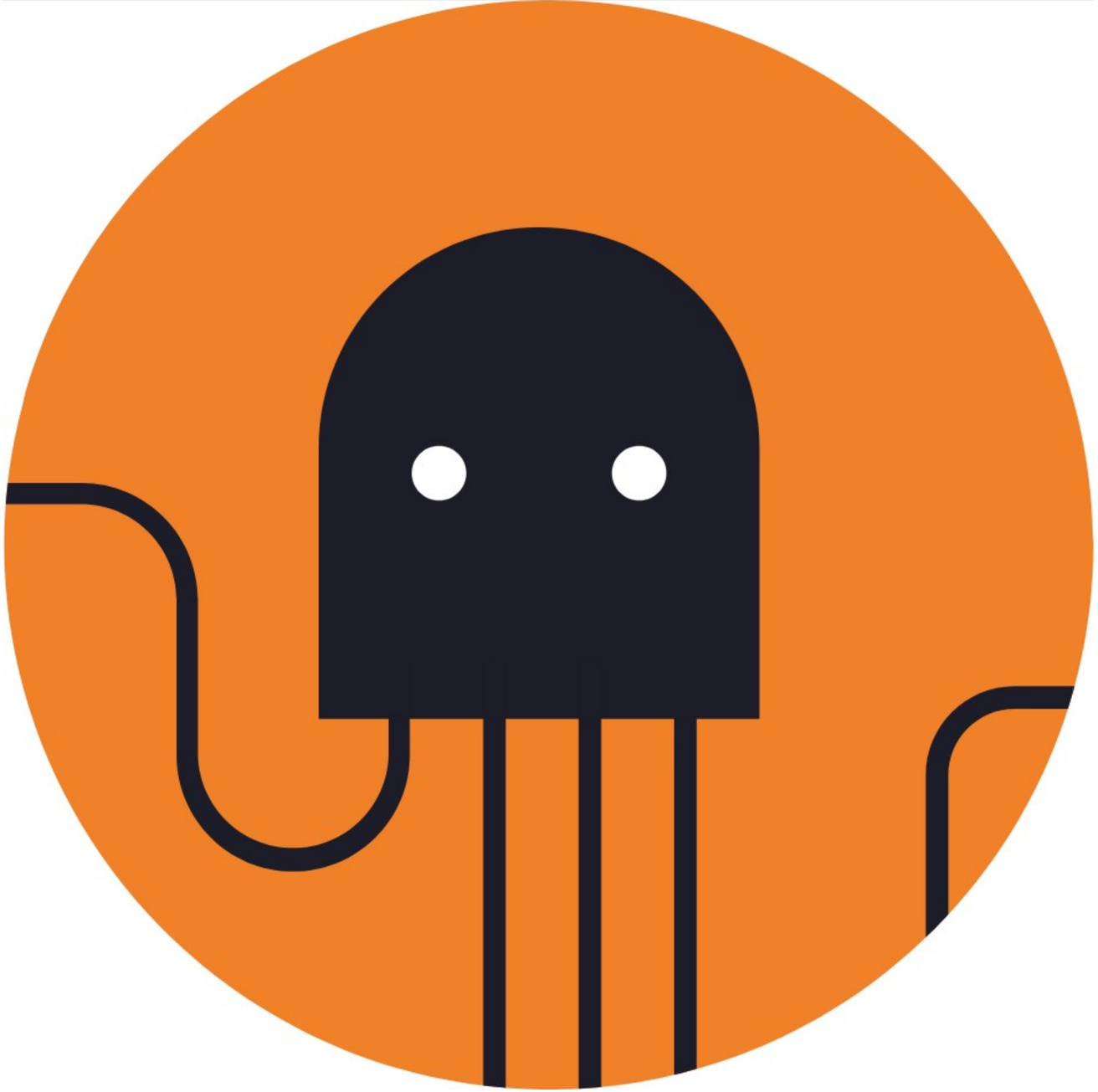
Эксперт-разработчик C++

antoshkka@gmail.com

antoshkka@yandex-team.ru

https://github.com/apolukhin

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



https://github.com/userver-framework