C++20 <coroutine>

Сопрограммы/корутины
Coroutines

М. Полубенцева

C++26 executor:

Базовая концепция — "**make one callable run in some execution context**"

- предлагает стандартную модель асинхронности, которая
- закрывает вопросы совместимости асинхронных интерфейсов,
- деления вычислительных ресурсов,
- а так же предоставляет набор базовых алгоритмов, позволяющих строить сложные асинхронные вычислительные графы

Замечание: далекое C++26 будущее

C++26 std::execution:

- Более тонкий контроль над параллельным выполнением
- Улучшенная безопасность и обработка ошибок
- **Интеграция с C++ корутинами**
- Расширяемость - возможность создания пользовательских политик
- Лучшая производительность через адаптивные алгоритмы

Неэффективность многопоточного подхода (для определенных классов задач)

- Создание потока – дорогая и тяжёлая операция.
- Переключение контекста – вещь дорогая! => при большой нагрузке вместо преимуществ можем получить проигрыш
- Синхронизация доступа к разделяемым данным – может быть очень дорого

Корутины и системные потоки (thread)

Системные потоки	**Корутины**
могут выполняться: на разных ядрах параллельно на одном ядре последовательно (по очереди)	несколько сопрограмм могут по очереди выполнять свой код на одном системном потоке
каждому потоку выделяется для выполнения квант времени	Корутина выполняется в контексте вызова => отдельного кванта никто не выделяет
Для переключения контекста потока затрачиваются ощутимые ресурсы	Вызов корутины сравним с вызовом обычной функции
переключаются системой в произвольные моменты времени (вытесняющая многозадачность)	переключаются «вручную», в местах, указанных программистом (кооперативная многозадачность).
Гонка => требуется синхронизация	**Гонки быть не может**! Так как в любой момент времени активна только одна корутина

Сравнение корутин и потоков (затраты на память => ограничение на количество)

- Корутины:

«стек» обычно небольшой (несколько КБ)

=> тысячи корутин

- Потоки:

каждый поток имеет свой стек (1-8 МБ)

=> ограниченное количество потоков (сотни)

Сравнение корутин и потоков (затраты на переключение контекста)

- Корутины:
  - переключение в контексте потока
  - не требует переключения контекста ядра
  - => очень быстрое (десятки наносекунд)

- Потоки:
  - переключение через планировщик ОС
  - требует переключения контекста потока
  - медленнее (микросекунды)

Кооперативная многозадачность

- Корутины:
  - Явное указание точек приостановки
  - Предсказуемое поведение
  - Не может быть гонки при однопоточном использовании

Упрощение асинхронного кода

```
// Без корутин («callback hell»)
void fetch_data() {
    connect_to_server([](Connection conn) {
        conn.send_request("data", [](Response resp) {
            process_response(resp, [](Result result) {
                // ...
            });
        });
    });
}
```

С корутинами:

```
task<void> fetch_data() {
    auto conn = co_await connect_to_server();
    auto resp = co_await conn.send_request("data");
    auto result = co_await process_response(resp);
    // Читаемый линейный код
}
```

Итог: главное преимущество!

Переключение между задачами происходит без участия ядра ОС (как при переключении потоков)

=> **эффективность**!

Когда что использовать:

Корутины идеальны для:	**Потоки лучше для:**
Асинхронного I/O	Когда нужна настоящая параллельность
Генераторов и ленивых последовательностей	CPU-bound задач (используют несколько ядер)
Кооперативной многозадачности	Задач, требующих изоляции (каждый поток со своим состоянием)
Ситуаций с большим количеством одновременных операций	

Замечание:

- Если программист передает выполнение кода корутины другому потоку,
- то появляется дополнительная возможность **управлять** выполнением кода в другом потоке!
- Но! Возвращаются проблемы гонки!

Гор Нишанов описал следующие цели проектирования корутин

Корутины должны:

- быть высоко масштабируемыми (до миллиардов одновременно работающих корутин).
- высокоэффективно продолжать и приостанавливать работу, сравнимо с накладными расходами функций.
- бесшовно взаимодействовать с существующими особенностями без дополнительных накладных расходов.
- иметь открытый механизм взаимодействия для реализации библиотек с различными вариантами высокоуровневых семантик, например, генераторы, горутины, задачи и тому подобное.
- иметь возможность использования в средах где исключения запрещены или невозможны.

Цель введения

- цель - сделать асинхронное программирование как можно проще???
- Отмена выполнения (=> избежать выполнения неактуальной работы)
- Избежать гонки за данными => необходимости синхронизации

Четыре новости:

- **Хорошая:**
  в C++20 появились средства для организации сопрограмм,
- **Плохая:**
  но **нет** стандартных средств для **облегчения** нашей работы с сопрограммами (возможно, в C++26…)
- **Хорошая**:
  но есть библиотека cppcoro, разработанная Lewis Baker
  https://github.com/lewissbaker/cppcoro
- **Хорошая:**
  C++23 появился класс std::generator<>

Комментарий к «плохой» новости:

- C++20 не предоставляет программисту класс coroutine (бери и пользуйся).
- C++20 предоставляет **framework** для создания корутин (<coroutine>), который состоит из
  - более 20 функций (методов класса) :
    - некоторые программист должен просто определить (сигнатура задана!)
    - некоторые перегрузить

  а компилятор будет вызывать пользовательские функции для управления корутиной
  - типов: std::coroutine_handle, std::suspend_always…
  - оператор co_await
  - ключевые слова co_return, co_yield

Основные составляющие <coroutine>:

- **std::coroutine_handle<>** - низкоуровневый handle для управления корутинами
- **std::coroutine_traits<>** - traits для настройки поведения корутин
- вспомогательные классы **std::suspend_always**, **std::suspend_never** - для управления приостановкой корутины

Чего можно ожидать в будущем?

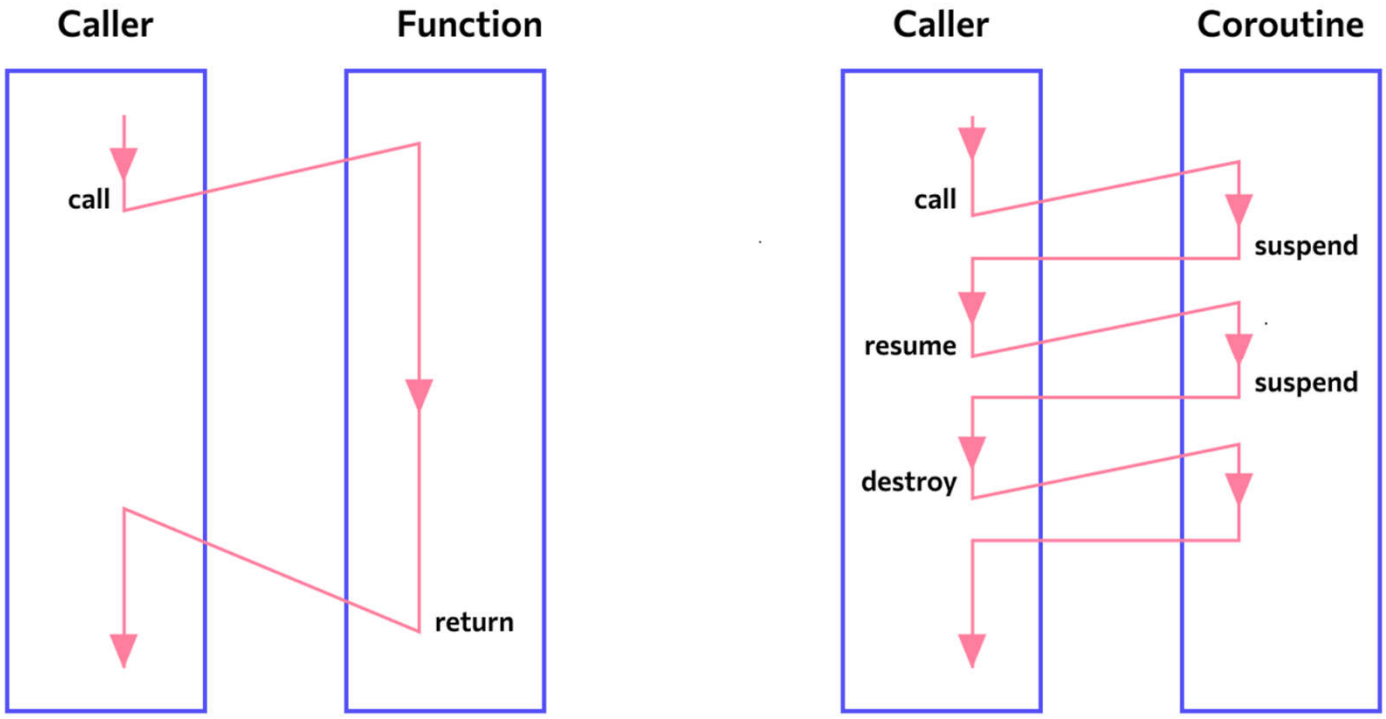
Скорее всего, появятся готовые типы корутин в стандартной библиотеке:

- std::generator<T> (уже есть в C++23)
- std::task<T> для асинхронных операций
- std::lazy<T> для ленивых вычислений

Что такое корутина/сопрограмма

- Сопрограмма (coroutine) – это функция (оформленная **специальным** образом + ограничения), выполнение которой можно прервать в процессе выполнения (с сохранением состояния), а позже продолжить с прерванного места (с восстановлением состояния).
- Обычно??? Выполняется в «родительском» потоке
- Управляет переключением задач программист «вручную»
- Принципиальное различие между сопрограммами и обычными функциями заключается в том, что сопрограмма обеспечивает возможность **явно** приостанавливать свое выполнение, отдавая контроль caller-у и возобновлять свою работу **в той же точке** при получении контроля обратно, с помощью дополнительных операций, сохраняя локальные данные (состояние выполнения), между последовательными вызовами, тем самым обеспечивая более гибкий и расширенный поток управления.

Иллюстрация выполнения обычных функций и сопрограмм



Синхронное решение:

```
std::string intToString(int n){
        return  std::to_string(n);
}

int main(){
        std::string s1 = intToString(1);
        //делаем что-нибудь полезное, но только после завершения функции
}
```

Асинхронное решение в отдельном потоке:

```
std::string intToString(int n){
        return  std::to_string(n);
}

int main(){
  std::future<std::string> f = std::async(std::launch::async, intToString, 2);
        //делаем что-нибудь полезное асинхронно, пока функция выполняется параллельно в отдельном потоке
  std::string s2 = f.get();
}
```

Асинхронно в родительском потоке

```
Smth intToStringCoro(int n){
        co_return  std::to_string(n);
}

int main(){
        Smth  smth = intToStringCoro(3);
        //делаем что-нибудь полезное
        std::string s = smth.getString(); //если результат еще не готов,
                возобновляем корутину и получаем результат
}
```

Асинхронно в родительском потоке с возвратом управления родителю и возобновлением выполнения

```
Smth intToStringCoro(int n){
        … //std::cout<<“Hello ”;
        co_await std::suspend_always{};
        … //std::cout<<“Coroutine!”;
        co_return  std::to_string(n);
}

int main(){
   Smth  smth = intToStringCoro(3);
   std::cout<<“before resume!”; //делаем что-нибудь полезное асинхронно (в родительском коде)
   smth.resume();
   std::cout<<“after resume!”; //снова делаем что-нибудь полезное асинхронно (в родительском коде)-
   std::string s = smth.getString(); //если результат еще не готов, возобновляем корутину и получаем результат
}
```

Корутина – это функция, которая должна содержать одно из ключевых co_слов:

- ключевое слово **co_return**: для завершения работы функции (return не допускается в корутине!). Возобновить выполнение корутины после вызова co_return невозможно!
- оператор **co_await**: для прерывания корутины с возможностью последующего продолжения
- ключевое слово **co_yield**: для приостановки корутины с одновременным возвратом результата (Это синтаксический сахар для конструкции с co_await). Формирует значение (которое может получить клиент/caller и приостанавливает корутину. Возобновление корутины может инициировать caller - продолжает ее с того места, где она была приостановлена.

Ключевые слова co_return, co_yeild и оператор co_await

- заменяются компилятором **на соответствующий код**, аналогично тому, как используется range based for loop или лямбда выражение.
  Но! важно!!! при автоматической генерации кода компилятор вызывает функции (с предопределенным назначением и сигнатурой), которые **должен предоставить программист**!
  Компилятор, генерируя код сопрограммы,
  - использует определенные пользователем типы и
  - вызывает в строго определенные моменты методы определенных пользователем типов,

  позволяя полностью настраивать и контролировать поведение сопрограммы
- кроме того, встретив в функции любое из этих ключевых слов, компилятор генерирует дополнительный код за открывающей { скобкой функции и }

Разница:

auto func(){ return 42; }	**… coro(){ co_return 42; }**
Возвращаемое значение? **42**	Возвращаемое значение? **НЕ 42**
Тип возвращаемого значения? **int**	Тип возвращаемого значения? **НЕ int**
Это сопрограмма? **НЕТ!**	Это сопрограмма? **ДА!**

Виды корутин (где располагается стек корутины):

- **stackfull** (не поддерживается C++20) – имеют собственный стек => могут запоминать не только текущее состояние функции, но и иерархию вызовов других функций внутри корутины => могут быть приостановлены с любого уровня вложенного вызова, так как должны сохранять весь стек
- **stackless** – не имеют собственного стека (C++20)– могут быть приостановлены только в теле самой корутины, так как сохраняют только текущий стековый кадр. При остановке stackless coroutine сохраняются только локальные переменные текущей функции (весь стек вызовов не сохраняется)
  => затраты по памяти минимальные!
  => что позволяет параллельно выполнять огромное количество корутин!

Важно! C++20 stackless coroutines

В C++20 реализованы только
stackless coroutines.
Данные, необходимые для возобновления выполнения сопрограммы, **в большинстве случаев хранятся НЕ в стеке caller-а, а в heap** => их называют stackless в отличие от stackfull сопрограмм, каковыми являются системные fibers

C++20 асимметричные coroutines

- Симметричные – при взаимодействии две корутины могут вызывать/приостанавливать друг друга,
- а в C++20 – ассиметричные –
  - caller вызывает сопрограмму и приостанавливает свое выполнение + имеет возможность возобновляет сопрограмму
  - сопрограмма приостанавливается и возвращает управление caller

caller	**coroutine**
вызов возобновление прием результата	**co_await** – приостановка и возврат управления caller-у с возможностью возобновления **co_return** - возврат управления caller-у – всегда!

C++ стандартизировал асимметричные корутины как основную модель

- Асимметричные корутины гораздо проще в использовании
- Симметричные корутины не входят в стандарт, но могут быть реализованы «вручную»
- Асимметричная модель лучше интегрируется с существующими библиотеками (Ranges) и шаблонами
- Большинство практических применений (генераторы, асинхронность) используют асимметричный подход

=> Для большинства задач в C++ рекомендуется использовать стандартные асимметричные корутины, так как они лучше поддерживаются компиляторами, более эффективны и проще в использовании.

Ограничения. Сопрограммой не может быть:

- Функция main;
- Функция с инструкцией return;
- Нельзя использовать goto через границы приостановки
- constexpr, consteval функция;
- Функция с автоматическим выведением типа возвращаемого значения (auto) (и даже с trailing return type);
- variadic templates функция;
- Функция с auto-параметрами;
- Конструктор и деструктор;
- Функции с noexcept (кроме `noexcept(false))

Кроме того, программист обязан предоставить для каждой корутины:

- Структуру promise_type с обязательнми составляющими
- Явную обработку исключений (умолчаний быть не может)

Как можно обойти ограничения variadic

```
template<typename... Args>
std::future<void> create_coroutine(Args... args) {
    // Создаем лямбду-корутину
    auto lambda = [](auto... captured_args) -> std::future<void> {
        // Используем captured_args...
        co_return;
    };

    return lambda(args...);
}
```

Как можно обойти ограничения constexpr

```
constexpr int compute_value() {
    return 42;
}

std::future<int> coroutine() {
    constexpr int value = compute_value();  // constexpr часть
    co_return value;  // асинхронная часть
}
```

Вызов корутины

- С точки зрения caller корутина
  - выглядит как обычная функция **со специфическим возвращаемым значением**.
  - и вызывается корутина как обычная функция (то есть компилятор генерирует при вызове такой же низкоуровневый код как и при вызове обычной функции)
- Но! Тот код, который компилятор генерирует за открывающей скобкой тела корутины принципиально отличается!
- Возвращает корутина специфический объект пользовательского типа, посредством которого можно
  - получить результат
  - и осуществлять взаимодействие с корутиной

Обязательные составляющие корутины:

- **promise_type** object– управление корутиной **изнутри** + осуществляет доставку результата из корутины.
- **coroutine_handle** - это невладеющий handle для продолжения работы или уничтожения frame-а корутины **снаружи**.
- **coroutine_frame** - это внутреннее (обычно размещенное в heap) состояние. Содержит объект promise, копии параметров корутины (или ссылки), состояние корутины - индекс приостановки/этапа (suspension point), локальные переменные…
- **awaiter type** - для приостановки и возобновления корутины
  - стандартные
  - пользовательские

Оператор co_await

«Нулевое» приближение

М. Полубенцева

Назначение/смысл унарного оператора co_await:

co_await выражение;

- co_await-ed выражение – это указание компилятору сгенерировать вызов/последовательность_вызовов методов **для объекта awaitable**
- позволяет задавать точки приостановки - возможность прерывания функции с последующим возобновлением
- co_await оператор можно использовать только в определенном контексте (в нашем случае – в теле корутины)
- тип, поддерживающий co_await operator называется **Awaiter type**

Синтаксис:

co_await <выражение/awaiter>

awaiter должен предоставлять следующие методы

await_ready(), await_suspend() и await_resume()

которые будут вызываться компилятором как часть co_await выражения

Замечание: std::suspend_always и std::suspend_never удовлетворяют awaitable-требованиям

Как получить awaitable объект?

- Непосредственно создать (в общем случае любое выражение, в результате которого формируется awaitable объект, в частности получить в качестве возвращаемого функцией значения)

  co_await < awaitable объект > ;

- трансформировать с помощью await_transform() – функции

  co_await promise.await_transform(expr);

Пример функционирования корутины:

```
CoroTask coro(int max){
  std::cout << "CORO " << max << " start\n";
  for (int val = 1; val < max; ++val) {
    std::cout << " CORO " << val << '/' << max << '\n'; // текущее значение
    co_await std::suspend_always{};  // точка приостановки
  }
  std::cout << " CORO " << max << " end\n";
}
```

Взаимодействие с корутиной:

```
int main(){
   std::cout << "coro() started\n";
   CoroTask  coroTask = coro(3); // инициализация и начало выполнения корутины
   while (coroTask.resume()){ //возобновление
       std::cout << "coro() suspended\n";
    }
   std::cout << "coro() done\n";
}
```

Пишем самую простую корутину

Реализуем пользовательский promise_type

М. Полубенцева

Что и зачем возвращает корутина?

Возвращаемое значение – это сущность **для управления выполнением корутины снаружи и для получения результата**!

Пользовательский тип => название любое (в нашем случае MyFuture), который должен:

- получать данные,
- пробуждать/возобновлять или уничтожать корутину посредством внедренного coroutine_handle.

Важно: программист

- **должен** предоставлять типы, данные и методы, которые будет использовать компилятор
- **+ может** дополнить любыми другими возможностями

Как будет выглядеть вызов корутины:

```
MyFuture intToStringCoro(int n){  //здесь не могу auto
        …
        co_return  std::to_string(n);
}

int main(){
        MyFuture  lazy = intToStringCoro(3); // аналог std::future
        //или
        auto  lazy = intToStringCoro(3); //а здесь уже auto можно!
        //делаем что-нибудь полезное
        std::string s = lazy.get_value();
} //~lazy
```

Две главные составляющие!

```
class MyFuture {
public:
        struct promise_type { //имя типа предопределено!!!
          promise_type() =default;
          ~promise_type() =default;

          ... //содержит обработчики всевозможных событий корутины.
        std::optional<std::string> current_value; //здесь будет интересующее нас значение
        };

        ...
private:
      std::coroutine_handle<promise_type> m_coroutine; //обеспечивает
                                передачу управления (возобновление выполнения) и удаление.
};
```

Связь co_ ключевых слов с promise_type:

co_	Метод promise_type
co_yield	yield_value(expr) Генерация значения + приостановка
co_return expr **co_return**	return_value(expr) return_void() unhandled_exception() Завершение
co_await expr	initial_suspend() final_suspend() await_transform() управление приостановкой

Интерфейс **promise_type** - позволяет настраивать поведение сопрограммы

Обязательные составляющие. Программист должен определять (но может реализовать по разному. Задан только интерфейс):

- <возвр_значение> **get_return_object**(); //аналог future – обеспечивает управление корутиной извне + получение результата
- <приостановить/продолжить> **initial_suspend**(); // поведение корутины при вызове
- <приостановить/продолжить> **final_suspend**(); // при возврате из сопрограммы
- **void unhandled_exception(); //**поведение при необработанном исключении в корутине, т.е. стратегию обработки исключительных ситуаций;

promise_type – необязательные составляющие

При использовании	**Нужно определить**
co_return	**void** return_value(T value); Или **void** return_void();
co_yield	<awaitable> yield_value(T value);

Важно! Набор зависит от того, какую функциональность хочется получить от корутины!

.Три способа использования предопределенного имени promise_type:

- Просто назвать вложенную структуру promise_type
- Сопоставить псевдоним promise_type пользовательскому типу с любым именем (имя типа произвольное – Promise, MyPromise). Будет использоваться при реализации под псевдонимом promise_type
- Другой способ определить тип Promise — это явно специализировать шаблон std::coroutine_traits<>

А можно и так:

```
struct MyFuture
{
    struct MyPromise
    {
        ...
    };
    using promise_type = MyPromise;
};
```

Специфика:

Объект типа promise_type

- создаётся компилятором неявно при каждом вызове/создании корутины (не! возобновлении!),
- хранится во фрейме корутины,
- содержит текущее состояние корутины
- и определяет ее поведение

Интерфейс promise_type:

get_return_object()	Для заготовки возвращаемого значения
yield_value()	Для co_yield
return_void(), return_value()	Для co_return
initial_suspend(), final_suspend()	Для управления приостановкой в начале и конце
unhandled_exception()	При генерации исключения в теле корутины

Разбираемся с promise_type – 1 вариант формирования MyFuture:

```
struct promise_type {
  promise_type() =default;
  ~promise_type() =default;
  MyFuture get_return_object(){return std::coroutine_handle<promise_type>::from_promise (*this);}
  std::optional<std::string> current_value;
};
```

// Этот метод вызывается в начале работы корутины. Он конструирует «традиционное» возвращаемое значение — то, что получает функция, вызвавшая корутину.

//а это placeholder для желаемого результата

Разбираемся с promise_type – 2 вариант формирования MyFuture::

```
struct promise_type {
  promise_type() =default;
  ~promise_type() =default;
  MyFuture get_return_object(){return MyFuture(*this);} // Этот метод вызывается в начале работы корутины. Он конструирует «традиционное» возвращаемое значение — то, что получает функция, вызвавшая корутину.
  std::optional<std::string> current_value; //а это placeholder для желаемого результата
};
```

Продолжение второго варианта:

```
class MyFuture {
public:
        struct promise_type {
          std::optional<std::string> current_value;
          MyFuture get_return_object(){return MyFuture(*this);}
        };

        …
private:
    MyFuture(promise_type& pr): m_coroutine
        (std::coroutine_handle<promise_type>::from_promise(pr)){}
    std::coroutine_handle<promise_type> m_coroutine; //обеспечивает
                передачу управления (возобновление выполнения) и удаление.
};
```

Что такое std::coroutine_handle<>:

- специализация для void:

```
template<> struct coroutine_handle { //базовая функциональность:
        //address(), static from_address()
        //done(), resume(), destroy()
};
```

- генеральный шаблон:

```
template<typename Promise>
struct coroutine_handle : coroutine_handle<void>{ //добавлены
        //promise(), static  from_promise()
};
```

Замечание - std::coroutine_handle<>:

- Можно/удобно сделать членом класса
- Но! В любой момент можно получить из promise_type – std::coroutine_handle<promise_type>::**from_promise()**

Важно! coroutine_handle — это **невладеющий** указатель

=>

- Необходимо где-то вызывать destroy() для освобождения ресурсов

- После вызова destroy() coroutine_handle становится недействительным

Преобразования

```
std::coroutine_handle<MyPromise> specialized_handle;
std::coroutine_handle<> generic_handle = specialized_handle;  // ОК

// Но! обратное преобразование требует проверки
if (generic_handle.address() != nullptr) {
    auto specialized =
        std::coroutine_handle<MyPromise>::from_address(
            generic_handle.address());
}
```

Важно!

```
auto MyFuture = some_coroutine();

   if (! MyFuture.done()) {
         MyFuture.resume();
   }
         MyFuture.destroy();  // явное уничтожение
```

std::coroutine_handle<promise_type>

Объект такого типа осуществляет взаимодействие с текущим состоянием корутины. Функциональность, необходимая для нашего примера:

promise_type& **promise**() const; //доступ

static **coroutine_handle** **from_promise**(promise_type& p); //создание и ассоциация с уже существующим объектом promise (в частности в возвращаемом дескрипторе формируется **указатель на фрейм корутины)**

Функциональность std::coroutine_handle:

- **resume(), operator()** - возобновляет приостановленную корутину
- **destroy()** окончательно завершает её выполнение.
  Корутина должна быть готова к уничтожению посредством destroy, так как однажды уснув, она может и не проснуться. При этом все её локальные переменные должны быть корректно деинициализированы и деаллоцированы
- **done(**) – проверка завершения корутины
- **operator bool**() - если указатель на фрейм nullptr (эквивалентно return bool(address())

Разбираемся с promise_type:

```
class MyFuture {
public:
  struct promise_type {
    std::optional<std::string> current_value;
    MyFuture get_return_object(){return MyFuture(*this);}
    static auto initial_suspend() noexcept { return std::suspend_never{}; }
    static auto final_suspend() noexcept { return std::suspend_never{}; }
};

        ...
};
```

<тип возвр. значения> initial_suspend()
- определяет поведение корутины при её вызове.

Для формирования возвращаемого значения в <coroutine> определены:

- **std::suspend_never** – если корутина должна стартовать сразу при вызове
- **std::suspend_always**, если это “ленивое” вычисление и приостановка работы происходит сразу

<тип возвр. значения> **final_suspend()** — аналогично определяет поведение корутины при завершении (штатном!);

Пояснение - псевдокод:

```
{
        promise_type ::initial_suspend();

        //тело корутины

        promise_type ::final_suspend();
}
```

Проверка пройденного - ???

```
MyFuture HelloCoro(){
  std::cout<<“Hello from coro ”;
  co_await std::suspend_always{};
  std::cout<<“Bye from coro ”;
}

int main(){
  MyFuture myF= HelloCoro();
  std::cout<<“1”;
  myF.resume();
  std::cout<<“2”;
} std::suspend_never
```

```
class MyFuture {
public:
  struct promise_type {
    …
    MyFuture get_return_object(){return MyFuture(*this);}
    static std::suspend_never initial_suspend() noexcept { return {}; }
    static std::suspend_never final_suspend() noexcept { return {}; }
};
```

Проверка пройденного - ???

```
MyFuture HelloCoro(){
  std::cout<<“Hello from coro ”;
  co_await std::suspend_always{};
  std::cout<<“Bye from coro ”;
}

int main(){
 MyFuture myF= HelloCoro();
 std::cout<<“1”;
 myF.resume();
 std::cout<<“2”;
}
```

```
class MyFuture {
public:
  struct promise_type {
    …
    MyFuture get_return_object(){return MyFuture(*this);}
    static std::suspend_always initial_suspend() noexcept { return {}; }
    static std::suspend_always final_suspend() noexcept { return {}; }
};
```

Проверка пройденного - ???

```
MyFuture HelloCoro(){
  std::cout<<“Hello from coro ”;
  co_await std::suspend_never{};
  std::cout<<“Bye from coro ”;
}

int main(){
  MyFuture myF= HelloCoro();
  std::cout<<“1”;
  std::cout<<“2”;
}
```

```
class MyFuture {
public:
  struct promise_type {
    …
    MyFuture get_return_object(){return MyFuture(*this);}
    static std::suspend_never initial_suspend() noexcept { return {}; }
    static std::suspend_never final_suspend() noexcept { return {}; }
};
```

Проверка пройденного - ???

```
MyFuture HelloCoro(){
   std::cout<<“Hello from coro ”;
   std::cout<<“Bye from coro ”;
}

int main(){
  MyFuture myF= HelloCoro(3);
  std::cout<<“1”;
  myF.resume();
  std::cout<<“2”;
}
```

Продолжаем разбираться с promise_type:

```
class MyFuture {
public:
  struct promise_type {
    std::optional<std::string> current_value;
    MyFuture get_return_object(){return MyFuture(*this);}
    static std::suspend_never initial_suspend() noexcept { return {}; }
    static std::suspend_never final_suspend() noexcept { return {}; }
    void return_value(string value) noexcept { //ничего не возвращает, а кладет значение в current_value!
                    current_value = std::move(value);
     }
};

          …
};
```

Продолжаем разбираться с promise_type:

```
class MyFuture {
public:
  struct promise_type {
    std::optional<std::string> current_value;
    MyFuture get_return_object(){return MyFuture(*this);}
    /*static*/ std::suspend_never initial_suspend() noexcept { return {}; }
    /*static*/ std::suspend_never final_suspend() noexcept { return {}; }
    void return_value(string value) noexcept {  current_value = std::move(value);  }
     /*static*/ void unhandled_exception() { std::terminate();  }
  };

        …
};
```

Frame корутины

Что такое frame корутины?

```
struct coroutine_frame{
    void (*resume)(coroutine_frame *); //указатель на функцию,  которая вызывается при передаче/возврате управления корутине
    void (*destroy)(coroutine_frame *); //указатель на функцию,  которая вызывается при удалении корутины
    promise_type promise;
    int16_t state; //текущее состояние (точка с которой корутина должна продолжить свое выполнение)
    bool heap_allocated; //был ли фрейм при создание размещен в куче или фрейм был создан на стеке вызывающей стороны
    // args аргументы вызова сопрограммы
    // locals локальные переменные функции (все!) + параметры
    //...
};
```

void (*resume)(coroutine_frame *);

Корутина - это state-машина.

Функция resume вызывается при

- инициализации сопрограммы
- и при каждой следующей передаче управления.

Т.е. по сути это исходный код функции корутины, **разбитый на состояния** посредством co_await/co_yield. В нашем случае только два состояния, начальное и конечное.

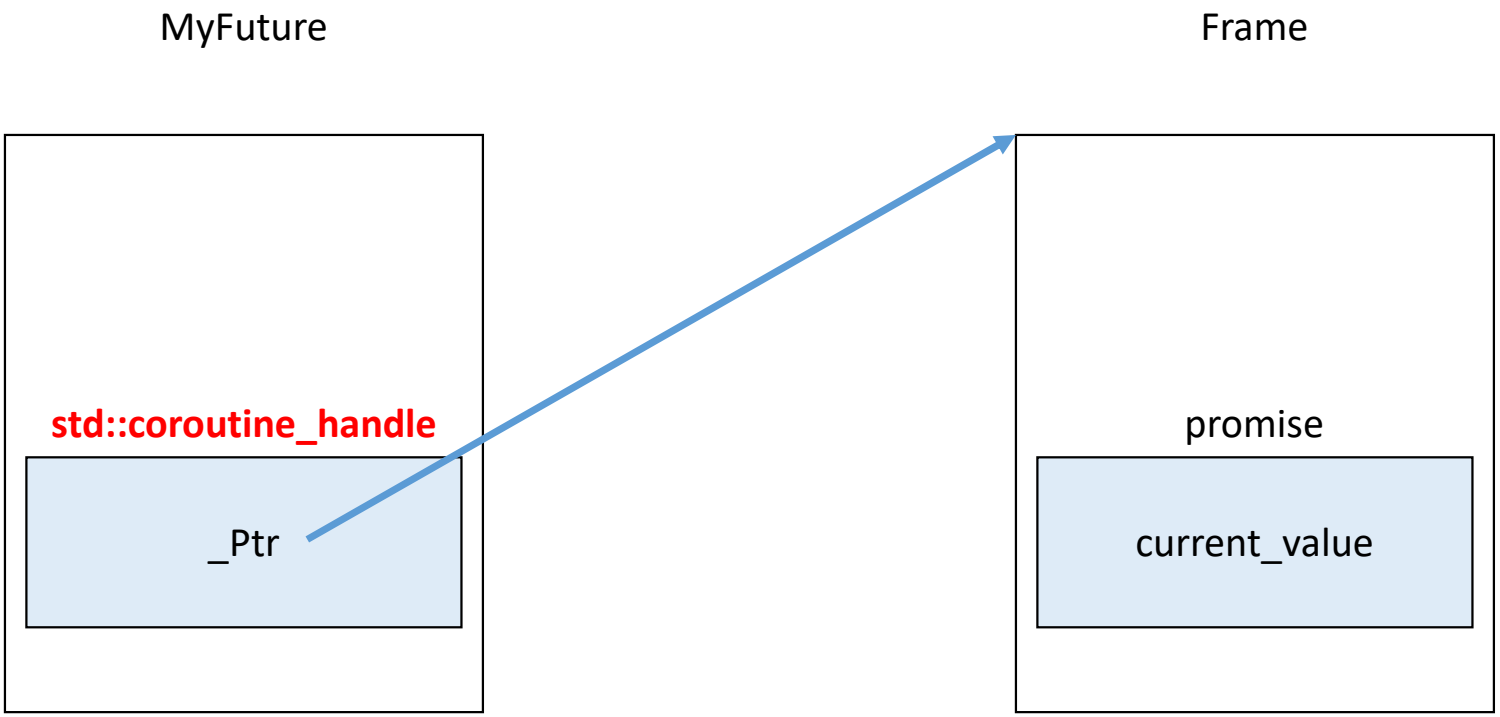
Псевдокод resume()

```
void resume(HelloCoro_frame * frame) {
  try {
   switch (frame->state) {
       case 0: //первый вызов при  initial_suspend()
              std::cout<<“Hello ”;
               frame->state=1;
              //co_await suspend_always{}
              return; // Приостановка
        case 1:
              std::cout<<“Bye ”;
              return;
         case 2:  //вызов при  final_suspend()
              return;
} catch(…) {frame->promise.unhandled_exception();}
              return;
}
```

```
MyFuture HelloCoro(){
  std::cout<<“Hello ”;
  co_await std::suspend_always{};
  std::cout<<“Bye ”;
}
```

```
class MyFuture {
public:
  struct promise_type {
    …suspend_always initial_suspend() …
    …suspend_always final_suspend() …
};
```

Как связаны MyFuture, promise и frame



Завершаем реализацию MyFuture:

```
class MyFuture {
public:
 struct promise_type {…};
 explicit MyFuture(std::coroutine_handle<promise_type> coro) :
                 m_coroutine{ coro } {    }
 MyFuture(MyFuture && other) noexcept : m_coroutine{other.m_coroutine}
                 {other.m_coroutine = nullptr;}
 MyFuture(const MyFuture &) = delete;
 MyFuture & operator=(const MyFuture &) = delete;
~ MyFuture() { if (m_coroutine) { m_coroutine.destroy(); }     }
```

Завершаем реализацию MyFuture - продолжение

```
 std::string get_value() {
while(!m_coroutine.promise().current_value)
        { m_coroutine.resume();} //Если значения нет, дадим корутине команду поработать
         return std::move(m_coroutine.promise().current_value);
    }

private:
   std::coroutine_handle<promise_type> m_coroutine;
};
```

Анатомия выполнения простейшей корутины

co_return

М. Полубенцева

Возвращаемое корутиной значение и ключевое слово co_return

Ключевое слово co_return используется для:

- формирования результата (если он предусмотрен)
  co_return выражение;
  и завершения корутины
- Если формирование результата не предусмотрено, то просто завершения корутины
  co_return;

Замечание: возвращаемое корутиной значение это хитрый объект пользовательского типа, посредством которого

- работает магия асинхронного выполнения
- и в частности можно получить требуемый результат

Напоминание:

возвращаемое_значение **coro(){ co_return 42; }**
Возвращаемое значение? **НЕ 42**
Тип возвращаемого значения? **НЕ int**

Пример использования co_return для ленивых вычислений:

```
<возвращаемое значение>
                        coro(<параметры>)
{
  std::cout<<“Before co_return”; //выполнится при вызове coro()
   co_return <выражение>; //это то, что реально хочется получить
  std::cout<<“After co_return”; //а это уже недостижимый код!
}
```

Очень нулевое приближение:

Что написал программист:	**Код, который генерирует компилятор:**
<тип_возвр_знач> coro() { **co_return 42;** }	{ promise_type **promise**; <тип_возвр_знач> ret = promise.get_return_object(); **co_await promise.initial_suspend()** ; try { **//тело корутины** **promise.return_value(42)**; //формирование рез. и **goto final_suspend;** } catch (...) { promise.unhandled_exception() ; } final_suspend : **co_await promise.final_suspend() ;** **//завершение сопрограммы – уничтожение фрейма корутины** }

Как же получить желаемый результат асинхронно?

Поясняем магию:

- Что сделает автоматически компилятор
- Что мы должны предоставить для свершения магии

Под капотом – продолжение (детальнее):

Что написал программист:	**Код, который генерирует компилятор:**
Lazy<int> lazy_sum(int x, int y) **{** **co_return x+y;** }	**Lazy<int>** lazy_sum (int x, int y) **{** //код по открывающей { //компилятор генерирует структуру: struct **CoroFrame** { void (*resume)(CoroFrame*); void (*destroy)(CoroFrame*); **Lazy<int>::promise_type promise;** **//самое главное!** bool initial_await_resume_called = false; int state = 0; **int m_x, int m_y;** … }; **…**

Под капотом - продолжение:

Что написал программист:	**Код, который генерирует компилятор:**
Lazy<int> lazy_sum (int x, int y) { **co_return x+y**; }	**Lazy<int>** lazy_sum (int x, int y) **{** //код по открывающей { //генерируется структура struct CoroFrame {…}; //создается динамический объект **auto coroFrame = new CoroFrame();** **//формируются поля структуры!!!** … }

Под капотом - продолжение:

Что написал программист:	**Код, который генерирует компилятор:**
Lazy<int> lazy_sum (int x, int y) { **co_return x+y;** }	**Lazy<int>** lazy_sum (int x, int y) **{** //генерируется структура struct CoroFrame {…}; //создается динамический объект auto coroFrame = new CoroFrame(); //формируются поля структуры, в частности promise //заготавливается возвращаемое значение **Lazy<int> returnObject{ coroFrame->promise.get_return_object() };** … }

Под капотом - продолжение:

Что написал программист:	**Код, который генерирует компилятор:**
Lazy<int> lazy_sum (int x, int y) { **co_return x+y;** }	**Lazy<int>** lazy_sum (int x, int y) **{** //генерируется структура struct CoroFrame {…}; //создается динамический объект auto coroFrame = new CoroFrame(); //формируются поля структуры, в частности promise //заготавливается возвращаемое значение **new (&возвр_знач){ coroFrame->promise.get_return_object() };** … }

Под капотом - продолжение:

Что написал программист:	**Код, который генерирует компилятор:**
Lazy<int> lazy_sum (int x, int y) { **co_return x+y;** }	**Lazy<int>** lazy_sum (int x, int y) **{** //генерируется структура struct CoroFrame {…}; //создается динамический объект и формируются поля auto coroFrame = new CoroFrame(x,y); //заготавливается возвращаемое значение new (&возвр_знач){ coroFrame->promise.get_return_object() }; **// вызывается promise_type::initial_suspend()** **if(coroFrame->promise.initial_suspend().await_ready())** **coroFrame->resume();** //если возвращается std::suspend_never, await_ready() возвращает true => продолжается выполнение; … }

Под капотом - продолжение:

Что написал программист:	**Код, который генерирует компилятор:**
Lazy<int> lazy_sum(int x, int y) { std::cout<<“!”; **co_return x+y**; }	**Lazy<int>** lazy_sum(int x, int y) **{** struct CoroFrame { … }; auto coroFrame = new CoroFrame(x,y); new (&возвр_знач){ coroFrame->promise.get_return_object() }; if(coroFrame->promise. initial_suspend().await_ready()) coroFrame->resume(); **std::cout<<“!”;** … }

Под капотом - продолжение:

Что написал программист:	**Код, который генерирует компилятор:**
Lazy<int> lazy_sum(int x, int y) { std::cout<<“!”; **co_return x+y;** }	**Lazy<int>** lazy_sum(int x, int y) **{** struct CoroFrame { … }; auto coroFrame = new CoroFrame(x,y); new (&возвр_знач){ coroFrame->promise.get_return_object() }; if(coroFrame->promise. initial_suspend().await_ready()) coroFrame->resume(); std::cout<<“!”; coroFrame->promise.return_value(coroFrame->m_x + coroFrame->m_y); goto finalSuspend; finalSuspend: if(coroFrame->promise.final_suspend().await_ready()) coroFrame->resume() ; … }

Последовательность вызовов:

- Lazy<int>::promise_type **promise**;
- **promise**.get_return_object(); //заготовка возвращаемого значения Lazy<int>
- **promise**.initial_suspend();
- Выполнение кода перед co_return – (в нашем случае std::cout<<“!”;)
- **promise**.return_value(x+y);
- **promise**.unhandled_exception(); //этот метод в нашем упрощенном примере не используется => реализуем как заглушку
- **promise**.final_suspend();

Когда освобождаются ресурсы корутины?

1. если корутина выполнилась до конца, то код после final_suspend() вызывает corutine_handle::destroy()

2. если корутина еще не завершилась, но вызван деструктор ~lazy

```
{
        Lazy<int>  lazy = lazy_sum(1,2);
        //делаем что-нибудь полезное
        int res= lazy.get_value();
        //…
}//вызывается деструктор ~lazy():  - corutine_handle::destroy() очищает ресурсы сопрограммы
```

Если корутина завершается посредством unhandled_exception

- ловится исключение и вызывается promise.unhandled_exception() из catch блока
- вызывается promise.final_suspend()

return_void() и return_value()

promise_type должен предоставить:

метод **return_void()**, если в корутине

- явно нет **co_return statement;**
- присутствует **co_return;**
- **co_return выражение;** но выражение имеет тип void

метод **return_value()**, если в корутине **co_return выражение;** которое не имеет тип void

Выполнение корутины в другом потоке

```
int main() {
    std::cout << "main id:  "
              << std::this_thread::get_id() << '\n';

    auto fut = createFuture();
    auto res = fut.get();
    std::cout << "res: " << res << '\n';

}
```

Продолжение:

```
template<typename T> struct MyFuture {
   struct promise_type {…};

   MyFuture(std::coroutine_handle<promise_type> h) : coro(h) {}
   ~MyFuture() {  if ( coro ) coro.destroy();  }
…
private:
std::coroutine_handle<promise_type>  coro;
};
```

Реализация promise_type

```
struct promise_type {
  promise_type(){ }
  ~promise_type(){ }
 auto get_return_object() { return
          MyFuture{std::coroutine_handle<promise_type> ::from_promise(*this)};  }
 void return_value(T v) { result = v;  }
 std::suspend_always initial_suspend() noexcept {  return {};  }
 std::suspend_always final_suspend() noexcept {  return {};  }
 void unhandled_exception() { std::exit(1); }
private:
   T result;
};
```

Продолжение:

```
template<typename T>
struct MyFuture {
    struct promise_type{…};
    …

    T get(){
        std::thread t([this] { coro.resume(); });
        t.join();
        return coro.promise().result;
    }
std::coroutine_handle<promise_type> coro;
```

Или так:

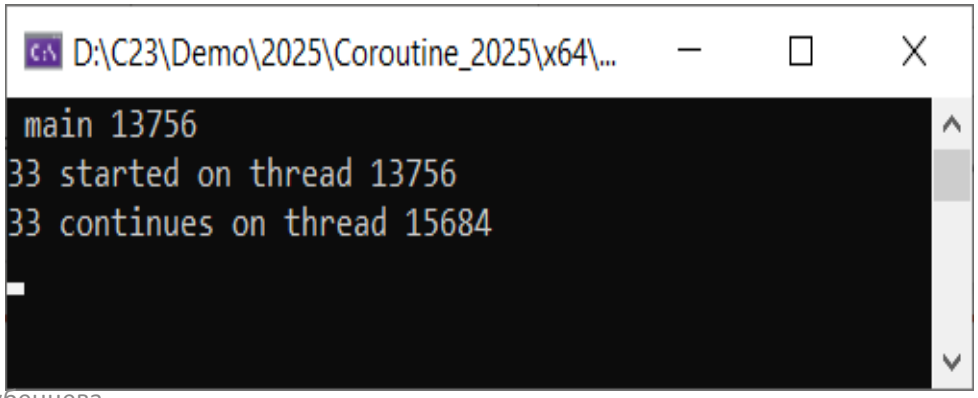
```
template<typename T>
struct MyFuture {
    struct promise_type{…};
    …

    T get(){
     std::jthread([this] { coro.resume(); });
     return coro.promise().result;
    }
std::coroutine_handle<promise_type> coro;
```

std::async() и совсем простая корутина

```
struct Task {
   struct promise_type {
      Task get_return_object() {  return Task{
          std::coroutine_handle<promise_type>::from_promise(*this) };  }
       std::suspend_never initial_suspend() noexcept { return {}; }
       std::suspend_always final_suspend() noexcept { return {}; }
       void return_void() {}
       void unhandled_exception() { std::terminate(); }
    };
   std::coroutine_handle<promise_type> handle;
   Task(std::coroutine_handle<promise_type> h) : handle(h) {}
   ~Task() { if (handle) handle.destroy(); }
   void resume() {  if (handle && !handle.done()) { handle.resume();  }  }
};
```

```
Task simple_coroutine(int n) {
    std::osyncstream(std::cout) <<  n << " started on thread "
        << std::this_thread::get_id() << std::endl;
    co_await std::suspend_always{};
    std::osyncstream(std::cout) << n << " continues on thread "
        << std::this_thread::get_id() << std::endl;
}
```

```
int main() {

        std::cout<<  " main "
         << std::this_thread::get_id() << std::endl;

      Task t = simple_coroutine(33);
      std::future<void> f = std::async(std::launch::async,
         [h = t.handle() ]  { h.resume();  });

      f.wait();
}
```



co_yield == co_await promise.yield_value(expr)

генератор на корутинах

Простой пример

```
struct promise_type {
    MyFuture get_return_object()  { return
        std::coroutine_handle<promise_type>::from_promise(*this); }
    void return_void() { }
    auto yield_value(int value) {
        current = value;
        return std::suspend_always{};
    }
    std::suspend_never initial_suspend() noexcept { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void unhandled_exception() { std::terminate(); }
    int current;
};
```

```
class MyFuture {
public:
 struct promise_type {…};
 int get_value() const { return m_coroutine.promise().current; }
  void move_next() {
       bool b = m_coroutine.done();
       if(!b) m_coroutine.resume();
    }
  MyFuture(std::coroutine_handle<promise_type> h) :m_coroutine(h) {}
private:
    std::coroutine_handle<promise_type> m_coroutine;
};
```

Продолжение:

```
MyFuture Producer() {
        co_yield 1;
        co_yield 2;
}

int main() {
            auto nums = Producer();
            std::cout << nums.get_value();
            nums.move_next();
            std::cout << nums.get_value();
   //     nums.move_next(); //безопасно?
}
```

co_yield – для реализации функции-генератора

Генератор — это функция, которая возвращает новое значение с каждым последующим возобновлением.

Функцию-генератор можно рассматривать как поток данных (data stream), из которого можно получать значения.

После формирования очередного значения корутина приостанавливается

Потоки данных могут быть бесконечными.

«Жадный» генератор (без корутин)

```
std::vector<int> getNumbers(int begin, int end, int inc = 1) {
    std::vector<int> numbers;
    numbers.reserve(abs(end-begin));
    for (int i = begin; i != end; i += inc) { numbers.push_back(i); }
    return numbers;
}
int main() {
   auto numbers = getNumbers(-10, 11);
   for (auto n : numbers) { std::cout << n << " ";  }
 //или
    for (auto n : getNumbers(0, 101, 5)) {  std::cout << n << " "; } //C++20
}
```

Ленивый генератор на корутинах

```
generator<int> generatorForNumbers(int begin, int inc = 1) {
    for (int i = begin; ; i += inc) {
        co_yield i; // сформировали очередное значение и приостановили
            корутину, чтобы дать возможность это значение обработать
    }
}
```

Пишем бесконечный генератор на корутинах:

```
Generator<int> getNext(int start = 0, int step = 1) noexcept {
    int current = start;
    while(true){
        co_yield current; //это точка приостановки!
        current += step;
    }
}

int main() {
  Generator<int> gen = getNext();
  for (int i = 0; i <= 10; ++i) {
        gen.next(); //здесь вызывается возобновление корутины
        std::cout << " " << gen.getValue();    // а здесь получаем очередной результат
    }
} //~gen
```

Начинаем реализовывать Generator<>:

```
template<typename T>
struct Generator {
public:
   class promise_type{ …  };
…
private:
std::coroutine_handle<promise_type> handle;
};
```

promise_type:

```
struct promise_type {
    auto initial_suspend() { return std::suspend_always{};  } //сразу приостановка
    auto final_suspend() {  return std::suspend_always{}; } //тоже приостановка
    auto get_return_object()
        { return Generator{std::coroutine_handle<promise_type> ::from_promise(*this)};    }
    auto return_void() {}

    auto yield_value(const T value) {
      current_value = value;
      return std::suspend_always{};
    }
    void unhandled_exception() {
      std::exit(1);
    }
    T current_value;
};
```

class Generator (аналог future)

```
template<typename T> class Generator {
public:
   struct promise_type { … };
   Generator(std::coroutine_handle<promise_type> h): handle(h) {}
   ~Generator() {  if ( handle ) handle.destroy(); }
   Generator(const Generator&) = delete;
   Generator& operator = (const Generator&) = delete;
   Generator(Generator&& other) noexcept : handle(other. handle)
                                                { other. handle = nullptr;  }
…
 private:
   std::coroutine_handle<promise_type> handle;
};
```

bool std::coroutine_handle<promise_type>::done() const;

Проверка:

true – если приостановленная корутина находится в последней точке приостановки

false – если в любой другой

class Generator – продолжение

```
template<typename T> class Generator {
public:
   struct promise_type { … };
  …
   T getValue() { return handle.promise().current_value; }
   bool next() {
      bool b = handle.done();
      if(b) {return false;}
           else {handle.resume();  return true;}
   }

};
```

запускаем бесконечный генератор:

```
 Generator<int> getNext(int start = 0, int step = 1) noexcept {
    auto value = start;
    for (int i = 0;; ++i) { //бесконечный цикл
        co_yield value; //формируем очередное значение и приостанавливаем корутину
        value += step;
    }
}

int main() {
    auto gen = getNext(100, -10);
    for (int i = 0; i <= 20; ++i) {  //количество значений задает caller
        gen.next();
        std::cout << " " << gen.getValue();
    }
}
```

Хотелось бы иметь возможность формировать заданное количество значений + использовать для этого диапазонный for:

```
//так:
 auto gen = Range(1, 10, 2); //вызов корутины
      for (int n : gen)
      {
         std::cout << n << ‘ ';
      }
  //или даже так:
for (int n : Range(-10, 10, -1)) {  //C++20
  std::cout << n << ‘ ‘;
}
```

Корутина:

```
Generator<int> Range(int start , int stop, int step = 1) noexcept {
    for(; start != stop;  start+=step){
      co_yield start;
    }
   co_return;
}
```

Что нужно реализовать для пользовательской структуры данных, чтобы ее можно было использовать в диапазонном for?

???

Что нужно реализовать для пользовательского итератора?

???

Добавляем в наш генератор тип итератора + методы для получения итераторов:

```
template<typename T> struct Generator {
            …
public:
  class iterator {
      …
 private:
      iterator(Generator<T>* self = nullptr) : myGenerator{ self } {}
      Generator<T>* myGenerator;
      friend struct Generator<T>;
};

   iterator begin() { return iterator{ coro.done() ? nullptr : this }; }
   iterator end() { return iterator{ nullptr }; }
};
```

Реализация методов итератора:

```
 class iterator {
public:
  bool operator != (iterator second) const {  return myGenerator!= second.myGenerator;    }

  iterator& operator++() {
     if (myGenerator->coro.done()) {myGenerator = nullptr;} //проверяем, завершилась ли корутина
              else {myGenerator->coro.resume();  }//возобновить выполнение корутины
       return *this;
   }

   T operator*() {
              return myGenerator->coro.promise().current_value; //достаем значение напрямую из promise
   }
};
```

C++23 - **std::generator<>**

<generator>

М. Полубенцева

```
// Генератор последовательности чисел [start, end]

std::generator<int> generate_sequence(int start, int end) {
    for (int i = start; i <= end; ++i) {
        co_yield i;  // Возвращаем значение и приостанавливаемся
    }
}

int main() {
    for (int num : generate_sequence(1, 5)) {
        std::cout << num << " ";  // Вывод: 1 2 3 4 5
    }
}
```

Пример - бесконечные последовательности Фибоначчи

```
std::generator<int> fibonacci() {
    int a = 0, b = 1;
    while (true) {
        co_yield a;
        int next = a + b;
        a = b;
        b = next;
    }
}
```

```
// Использование с ограничением
#include <ranges>
int main() {
    for (int fib : fibonacci() | std::views::take(10)) {
        std::cout << fib << " ";  // 0 1 1 2 3 5 8 13 21 34
    }
}
```

Чтение из файла

```
// Ленивое чтение файла построчно
std::generator<std::string> read_lines(const std::string& filename) {
    std::ifstream file(filename);
    std::string line;

    while (std::getline(file, line)) {
        co_yield line;
    }
}
```

Возврат по ссылке

```
std::generator<const std::string&> get_names(const std::vector<std::string>& names) {
    for (const auto& name : names) {
        co_yield name;  // Возвращаем ссылку, а не копию
    }
}

std::generator<int&> get_modifiable(std::vector<int>& vec) {
    for (int& elem : vec) {
        co_yield elem;  // Можно изменять исходный вектор
    }
}
```

co_await

объекты ожидания

М. Полубенцева

Механизм приостановки:

- Точки приостановки помечены ключевым словом co_await
- Когда выполнение программы доходит до такой точки:
  - все значения, которые в данный момент компилятор разместил на регистрах, записываются во фрейм корутины
  - точка приостановки тоже записывается во фрейм (индекс), чтобы операция возобновления (resume) знала, «куда» возвращаться, + чтобы операция уничтожения (destroy) знала, какие объекты на этот момент были проинициализированы => для них нужно вызвать деструкторы
- Напоминание: управление фреймом осуществляется посредством coroutine_handle:
  - возобновление
  - уничтожение
  - получение доступа к объекту promise_type из фрейма корутины
  - проверка – корутина закончила выполнение

Оператор co_await – что это?

co_await <выражение>; //результатом выражения является объект

- объект такого типа, который реализует предопределенный интерфейс (концепцию awaitable)!
  Примеры – структуры std::suspend_never и std::suspend_always
- объект awaitable типа можно получить двумя способами:
  - Использование стандартных (suspend_always, suspend_never)
  - прямое создание объекта (тип должен удовлетворять концепции awaitable)
  - трансформацию объекта в awaitable с помощью await_transform() - функции

co_awaitable тип должен предоставлять предопределенный интерфейс :

предопределенный интерфейс co_awaitable типа состоит из трёх методов:

- **bool await_ready**(), проверка готовности результата
- **void/bool/coroutine_handle await_suspend**(coroutine_handle<>) – для того, чтобы указать, что должно выполняться после приостановки корутины
  - void – корутина остается приостановленной, а управление возвращается caller или тому, кто вызовет resume() корутины
  - bool – чтобы в методе await_suspend() программист мог предусмотреть продолжение выполнения корутины (false)
  - возврат управления другой корутине
- и **await_resume**() – значение, которое возвращает co_await <выражение> ==возвращаемому значению await_resume()

Из того, что мы уже рассматривали простейшими ожидающими типами являются std::suspend_never и std::suspend_always

Псевдокод:

```
Awaitable_type awaitable{параметры конструктора};
  if (!awaitable.await_ready()) {
      // Компилятор получает handle текущей корутины
      std::coroutine_handle<> h =
          std::coroutine_handle<promise_type>::from_promise(
              get_current_coroutine_promise()
          );

      // И передает его в await_suspend
      awaitable.await_suspend(h);

      // Здесь корутина приостанавливается
      return; // suspension point
  }
awaitable.await_resume();  // Если await_ready() вернул true, сразу вызываем await_resume()
```

co_await **Awaitable_type**{параметры конструктора};

Получить promise текущей корутины из awaiter

```
struct Task {
   struct promise_type {…};

      static promise_type& current_promise() {
       auto handle = std::coroutine_handle<promise_type>::from_address(
          std::coroutine_handle<>::from_address(nullptr).address()
       );
       return handle.promise();
   }
```

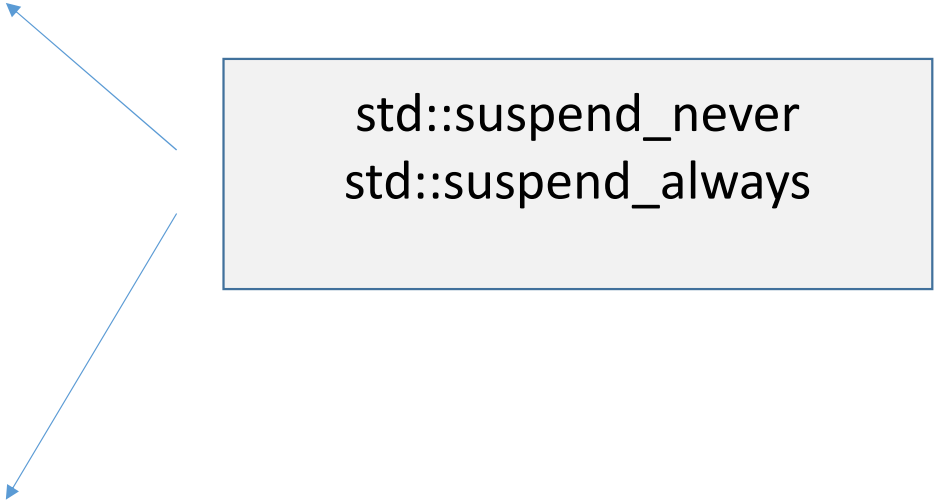
Стандартные awaiter-ы.

Простые awaiter-ы, которые предоставляет стандартная библиотека – структуры, которые удовлетворяют концепции awaitable:

- std::**suspend_always** – тип, который заставляет оператор co_await всегда приостанавливать корутину и возвращать управление caller
- std::**suspend_never** – никогда не приостанавливать корутину

на самом деле вызовы initial_suspend() и final_suspend() возвращают co_await объекты:

```
{
   Promise promise;
   co_await promise.initial_suspend();
   try {
      <тело функции>
   } catch (...) {
      promise.unhandled_exception();
   }
FinalSuspend:
   co_await promise.final_suspend();
}
```



std::suspend_never

```
struct suspend_never {
  constexpr bool await_ready() const noexcept { return true; }
  constexpr void await_suspend(coroutine_handle<>) const noexcept {}
  constexpr void await_resume() const noexcept {}
};
```

std::suspend_always

```
struct suspend_always {
  constexpr bool await_ready() const noexcept { return false; }
  constexpr void await_suspend(coroutine_handle<>) const noexcept {}
  constexpr void await_resume() const noexcept {}
};
```

Простой пример – экспериментируем с корутиной lazy_sum():

```
template<typename T> class Lazy {
public:
 struct promise_type {
     …
   static std::suspend_never initial_suspend()
              noexcept { return {}; }
    static std::suspend_always final_suspend()
              noexcept {  return {};  }
    };
    …

private:
std::coroutine_handle<promise_type> m_coroutine;
};
```

```
template<typename T> Lazy<T> lazy_sum(T x, T y){
    std::cout << "Before co_return" << std::endl ;
    co_return x + y;
}

int main() {
  std::cout << "Ready to call coroutine  " <<  std::endl;
  Lazy<int> x = lazy_sum<int>(1, 2);

  std::cout << "Before get_value  ";
 int res = x.get_value();
 std::cout<<res<< std::endl;
}
```

Простой пример – экспериментируем с корутиной lazy_sum():

```
template<typename T> class Lazy {
public:
 struct promise_type {
     …
   static std::suspend_always initial_suspend()
              noexcept { return {}; }
    static std::suspend_always final_suspend()
              noexcept {  return {};  }
    };
    …

  void resume()    {
      if (m_coroutine) m_coroutine.resume();
   }
private:
std::coroutine_handle<promise_type> m_coroutine;
};
```

```
template<typename T> Lazy<T> lazy_sum(T x, T y){
    std::cout << "Before co_return" << std::endl ;
    co_return x + y;
}

int main() {
  std::cout << "Ready to call coroutine  " <<  std::endl;
  Lazy<int> x = lazy_sum<int>(1, 2);
  std::cout << "Before resume  ";
  x.resume();
  std::cout << "Before get_value  ";
 int res = x.get_value();
 std::cout<<res<< std::endl;
}
```

Простой пример – экспериментируем с корутиной lazy_sum():

```
template<typename T> class Lazy {
public:
 struct promise_type {
    …
  static std::suspend_never initial_suspend()
            noexcept { return {}; }
   static std::suspend_always final_suspend()
            noexcept {  return {};  }
   };
   …

  void resume()   {
     if (m_coroutine) m_coroutine.resume();
   }
private:
std::coroutine_handle<promise_type> m_coroutine;
};
```

```
template<typename T> Lazy<T> lazy_sum(T x, T y){
  std::cout << "Before co_await" << std::endl ;
   co_await std::suspend_always{};
   std::cout << "Before co_return" << std::endl ;
   co_return x + y;
}

int main() {
 std::cout << "Ready to call coroutine  " <<  std::endl;
 Lazy<int> x = lazy_sum<int>(1, 2);
 std::cout << "Before resume  ";
 x.resume();
 std::cout << "Before get_value  ";
int res = x.get_value();
std::cout<<res<< std::endl;
}
```

Подробнее + простые примеры пользовательских awaiter-ов

await_ready() – проверка готовности

```
bool await_ready() {
   // true, если приостановка не нужна
   //  false, если требуется приостановка и ожидание
   return false; // обычно false для асинхронных операций
}
```

Пример:

```
struct ImmediateAwaitable {
    bool await_ready() {
        return true; // Never suspend
    }

    void await_suspend(std::coroutine_handle<>) {
        // Never called since await_ready returns true
    }

    int await_resume() {
        return 42; // Immediate result
    }
};
```

await_suspend(std::coroutine_handle<>)

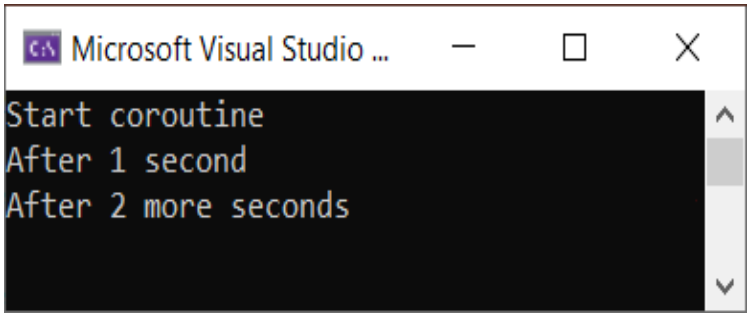
```
void await_suspend(std::coroutine_handle<> handle) {
    // 'handle’ приостановленной корутины
    // возобновляет корутину «при готовности»
}
```

```
struct DelayAwaitable {
    std::chrono::milliseconds delay;
    DelayAwaitable(std::chrono::milliseconds ms) : delay(ms) {}

    bool await_ready() const noexcept { return false; }// { return delay.count() <= 0; }
    void await_suspend(std::coroutine_handle<> h) const {
        std::thread([h, delay = this->delay]() mutable {
            std::this_thread::sleep_for(delay);
            h.resume();
            }).detach();
    }
    void await_resume() const noexcept {} //не используется
};
```

Продолжение

```
struct Task {
    struct promise_type {
        Task get_return_object() { return {}; }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() {}
    };
};
```

```
Task example_coroutine() {
    std::cout << "Start coroutine\n";
    co_await DelayAwaitable{ std::chrono::seconds(1) };
    std::cout << "After 1 second\n";
    co_await DelayAwaitable{ std::chrono::seconds(2) };
    std::cout << "After 2 more seconds\n";
}

int main() {
    example_coroutine();
   // Даем время выполниться асинхронным операциям
    std::this_thread::sleep_for(std::chrono::seconds(5));
}
```



await_resume() – возвращает результат при возобновлении корутины

```
auto await_resume() {
    return result; // Can return any type, including void
}
```

```
struct ValueAwaitable { // Returning a value
    int value;
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<>) {}
    int await_resume() { return value; }
};
struct VoidAwaitable { //  void
    bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<>) {}
    void await_resume() {} // void return
};
struct ReferenceAwaitable { // Returning a reference
    std::string& str;
     bool await_ready() { return false; }
    void await_suspend(std::coroutine_handle<>) {}
    std::string& await_resume() { return str; }
};
```

Пример **await_resume()**

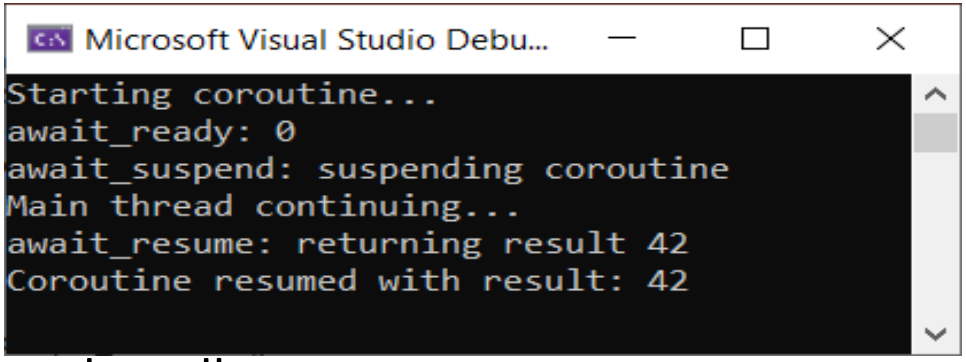
```
struct SimpleAwaitable {
    bool completed = false;
    int result = 0;
    std::coroutine_handle<> continuation;

    bool await_ready() {
        std::cout << "await_ready: " << completed << std::endl;
        return completed; // Only ready if already completed
    }
```

Продолжение:

```
void await_suspend(std::coroutine_handle<> handle) {
        std::cout << "await_suspend: suspending coroutine" << std::endl;
        continuation = handle;

        std::thread([this]() {// Simulate async work
            std::this_thread::sleep_for(std::chrono::seconds(1));
            result = 42;
            completed = true;
            continuation.resume(); // Resume the coroutine
            }).detach();
    }
int await_resume() {
        std::cout << "await_resume: returning result " << result << std::endl;
        return result;
    }
};
```

```
struct AsyncOperation {
   struct promise_type {
      int result;

      AsyncOperation get_return_object() {
         return AsyncOperation{ std::coroutine_handle<promise_type>::from_promise(*this)
};
      }
      std::suspend_never initial_suspend() { return {}; }
      std::suspend_never final_suspend() noexcept { return {}; }
      void return_value(int value) { result = value; }
      void unhandled_exception() {}
   };

   std::coroutine_handle<promise_type> handle;
};
```

```
AsyncOperation example_coroutine() {
    std::cout << "Starting coroutine..." << std::endl;
    int result = co_await SimpleAwaitable{};
    std::cout << "Coroutine resumed with result: " << result << std::endl;
    co_return result;
}

int main() {
    auto task = example_coroutine();
    std::cout << "Main thread continuing..." << std::endl;

    // Wait for async operation
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 0;
}
```



Демонстрация (псевдокод) - co_await a;

```
std::exception_ptr exception = nullptr;
if (not a.await_ready()) {
  suspend_coroutine();

  //if await_suspend returns void - if constexpr (std::is_void_v<await_suspend_result_type>)
  try {
    a.await_suspend(coroutine_handle);
    return_to_the_caller();
  } catch (...) {
    exception = std::current_exception();
     goto resume_point;
}
goto resume_point;
  //endif
```

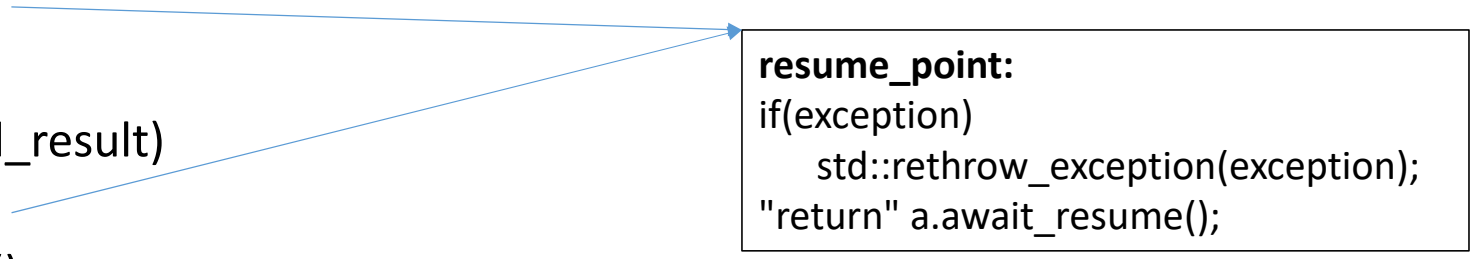
```
resume_point:
if(exception)
     std::rethrow_exception(exception);
"return" a.await_resume();
```

Замечание:

- Приостановка корутины не всегда означает возврат управления caller
- Приостановка означает, что данные сохраняются во фрейме для последующего восстановления при возврате управления корутине (resume). В приведенном примере точкой возврата будет метка resume_point

Продолжение

```
//if await_suspend returns bool
  bool await_suspend_result;
  try {
    await_suspend_result = a.await_suspend(coroutine_handle);
  } catch (...) {
    exception = std::current_exception();
    goto resume_point;
  }
  if (not await_suspend_result)
    goto resume_point;
  return_to_the_caller();
  //endif
```



Продолжение:

```
 //if await_suspend returns another coroutine_handle
  decltype(a.await_suspend(std::declval<coro_handle_t>())) another_coro_handle;
  try {
    another_coro_handle = a.await_suspend(coroutine_handle);
  } catch (...) {
    exception = std::current_exception();
    goto resume_point;
  }
  another_coro_handle.resume();
  return_to_the_caller();
  //endif
}
```

Продолжение:

resume_point:

```
if(exception)
    std::rethrow_exception(exception);
"return" a.await_resume();
```

Пользовательский awaitable объект

Подписка на результат.

Синхронизация с помощью awaitable объекта

М. Полубенцева

Задача:

Есть разделяемые глобальные данные:

```
int g_value;
```

и глобальный объект ожидания, состоянием которого мы будем управлять:

```
evt_awaiter_t g_event;
```

consumer-ы – корутины с разным количество приостановок

MyFuture consumer1() //1 точка приостановки

{

std::cout << "consumer1 started" << std::endl;

co_await g_event; //co_await сделал suspend и поставил точку для resume

std::cout << "consumer1 resumed" << std::endl;

}

MyFuture consumer2() //2 точки приостановки

{

std::cout << "consumer2 started" << std::endl;

co_await g_event;

std::cout << "consumer2 resumed" << std::endl;

co_await g_event;

std::cout << "consumer2 resumed again" << std::endl;

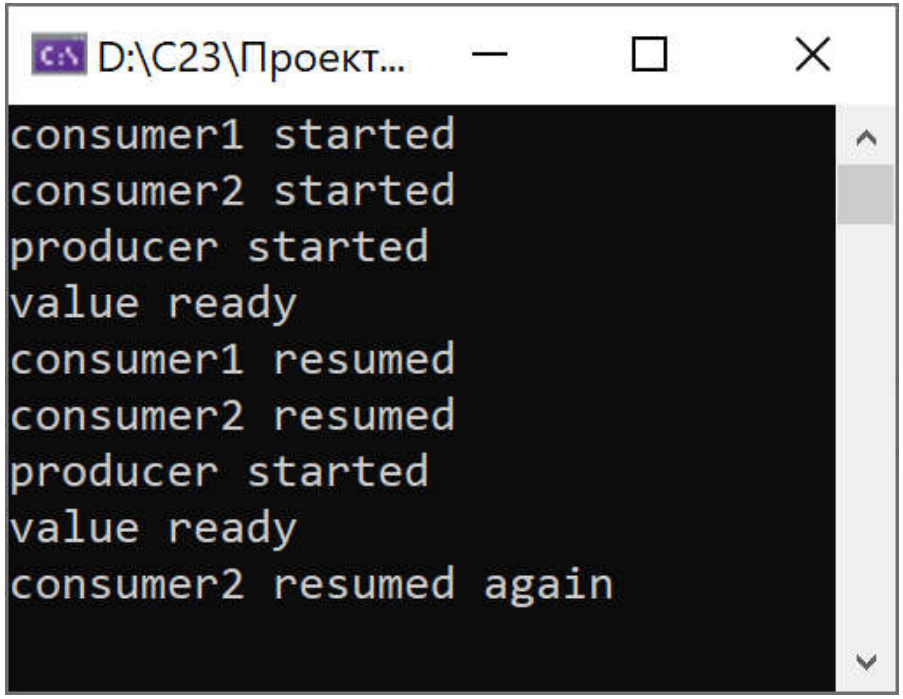
}

Producer – обычная функция

```
void producer() //это обычная функция
{
        std::cout << "producer started" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        g_value = 42;
        std::cout << "value ready" << std::endl;
        g_event.set();
}
```

Как должно работать:

```
int main(){
    consumer1();
    consumer2();
    producer();
    producer();
}
```



```
struct MyFuture {//невладеющий возвращаемым значением объект
  struct promise_type {
        MyFuture get_return_object(){
                        return std::coroutine_handle<promise_type>::from_promise(*this); }
        void return_void() {}
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void unhandled_exception() { std::terminate(); }
        };
        MyFuture(std::coroutine_handle<promise_type> h) :m_coroutine(h) {}
private:
        std::coroutine_handle<promise_type> m_coroutine;
        ~MyFuture(); //???
};
```

```
using coro_t = std::coroutine_handle<>;
struct awaiter {
        evt_awaiter_t& event_; //адрес общего await-ера
        coro_t coro_ = nullptr; //
        awaiter(evt_awaiter_t& event) noexcept :event_(event) {}
        bool await_ready() const noexcept { return event_.is_set(); }
        void await_suspend(coro_t coro) noexcept {
                coro_ = coro; //сохраняем дескриптор
                event_.push_awaiter(*this); //заносим "себя" в список ожидания
                }
        void await_resume() noexcept { event_.reset(); }
};
```

```
class evt_awaiter_t {
        struct awaiter;
        std::list<awaiter> lst_; //список ожидающих resume
        bool set_; //флаг
        struct awaiter {…};
        bool is_set()const noexcept { return set_; }
        void push_awaiter(awaiter a) { lst_.push_back(a); }
        void set() noexcept {
                set_ = true;
                size_t n = lst_.size(); //количество корутин, которое нужно resume
                while (n != 0) {
                        lst_.front().coro_.resume();
                        lst_.pop_front();
                        n--;
                }
        }
        void reset() noexcept  {  set_ = false;  }
};
```

Не работает! Почему?

```
class evt_awaiter_t {
…
public:
   awaiter operator co_await() noexcept { return awaiter(*this); }
..
};
```