PkXwmpgN
18 сен 2020 в 01:01

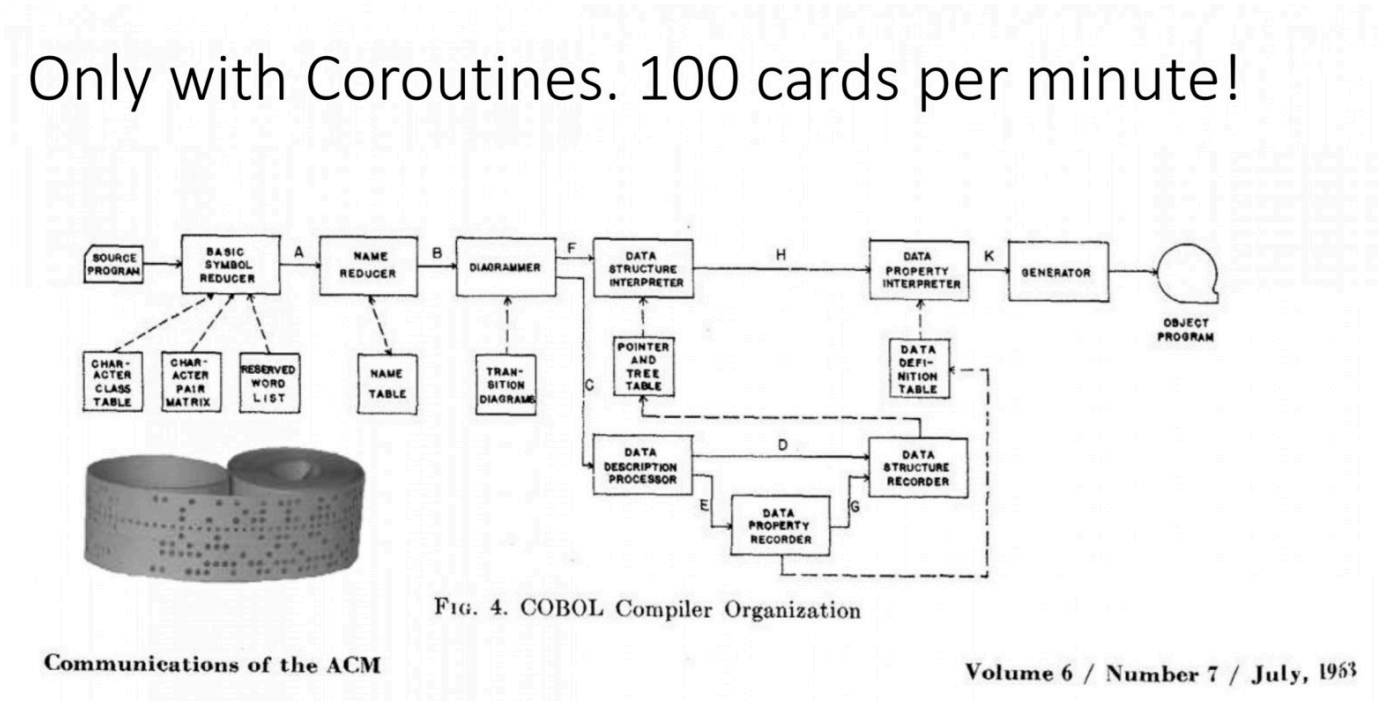
C++20. Coroutines

33 мин 66K

C++*, Программирование*

Туториал Технотекст 2020

В этой статье мы подробно разберем понятие сопрограмм (coroutines), их классификацию, детально рассмотрим реализацию, допущения и компромиссы, предлагаемые новым стандартом C++20.



Общие сведения

Сопрограммы можно рассматривать как обобщение понятия подпрограмм (routines, функций) в срезе выполняемых над ними операций. Принципиальное различие между сопрограммами и подпрограммами заключается в том, что сопрограмма обеспечивает возможность явно приостанавливать свое выполнение, отдавая контроль другим программным единицам 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.

Чтобы внести больше ясности в это определение и дальнейшие рассуждения и ввести вспомогательные понятия и термины, рассмотрим механику обычных функций в C++ и их стековую природу.

Мы будем рассматривать семантику функции в контексте двух операций.

Вызов (call). Передача управления вызываемой процедуре. Выполнение операции можно разделить на несколько этапов:

1. Выделить доступную вызываемой процедуре область памяти — *кадр* (*activation record*, *activation frame*), необходимого размера;
2. Сохранить значения регистров процессора (локальные данные) для последующего их восстановления, когда управление вернётся из вызываемой процедуры;
3. Поместить значения аргументов вызова в доступную для процедуры область памяти. В этой же памяти размещаются локальные переменные;
4. Поместить адрес возврата — адрес команды, следующей за командой вызова в доступную для процедуры область памяти.

После чего процессор переходит по адресу первой команды вызываемой процедуры, передавая поток управления.

Возврат из процедуры (return). Передача управления обратно вызывающей стороне. Выполнение этой операции также состоит из несколько этапов:

1. Сохранить (если необходимо) возвращаемое значение в области памяти доступной вызывающей процедуре;
2. Удалить локальные переменные, переданные аргументы;
3. Восстановить значения регистров.

Дальше процессор переходит по адресу команды из адреса возврата, передавая управление обратно вызывающей стороне. После получения управления вызывающая сторона освобождает выделенную вызываемой процедуре память (кадр).

Важно заметить следующее:

1. Выделяемая вызываемой процедуре память имеет строго вложенную структуру и время жизни (*strictly nested lifetime*) относительно вызывающей стороны. Другими словами, в каждый момент времени есть один активный кадр: кадр вызванной процедуры. После возврата управления активным становится кадр вызывающей стороны.
2. Размер кадра известен на стороне вызывающей процедуры.

Эти свойства позволяют использовать структуру, которая называется аппаратным стеком или просто стеком. **Аппаратный стек** — это непрерывная область памяти, аппаратно поддерживаемая центральным процессором и адресуемая специальными регистрами: **ss** (сегментный регистр стека), **bp** (регистр указателя базы стекового кадра), **sp** (регистр указателя стека), последний хранит адрес вершины стека (смещение относительно сегмента стека). Чтобы выделить память на стеке достаточно просто сместить указатель вершины стека на требуемый размер, чтобы освободить память, нужно вернуть указатель в исходное положение.

Выделенная таким образом память называется стековым кадром или стекфреймом. Стекфрейм имеет строгую организацию, которая определяется соглашением о вызове (*Calling Convention*). Соглашение зависит от компилятора, от особенностей аппаратной платформы и стандартов языка. Основные отличия касаются особенностей оптимизации (использование регистров) и порядка передачи аргументов. Также им определяется, например, сторона, на которой будут восстанавливаться регистры после вызова.

Рассмотрим простой пример:

```
void bar(int a, int b)
{}

void foo()
{
    int a = 1;
    int b = 2;
    bar(a, b);
```

```
}

int main()
{
    foo();
}
```

Без каких-либо оптимизаций будет сгенерирован следующий код (x86-64 clang 10.0.0 -m32, код сгенерирован в 32х битном окружении просто чтобы продемонстрировать работу стека. По соглашению о вызовах для 64х битных систем при передачи аргументов в функцию, в таком простом случае, стек участвовать не будет, аргументы будут переданы напрямую через регистры):

```
bar(int, int):
        push    ebp
        mov     ebp, esp
        mov     eax, dword ptr [ebp + 12]
        mov     ecx, dword ptr [ebp + 8]
        pop     ebp
        ret
foo():
        push    ebp
        mov     ebp, esp
        sub     esp, 24
        mov     dword ptr [ebp - 4], 1
        mov     dword ptr [ebp - 8], 2
        mov     eax, dword ptr [ebp - 4]
        mov     ecx, dword ptr [ebp - 8]
        mov     dword ptr [esp], eax
        mov     dword ptr [esp + 4], ecx
        call    bar(int, int)
        add     esp, 24
        pop     ebp
        ret
main:
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        call    foo()
        xor     eax, eax
        add     esp, 8
```

```
        pop     ebp
        ret
```

Проиллюстрируем работу стека:

Начало работы функции `main`, на стеке лежит адрес возврата из функции, пушим на стек значение регистра `ebp` (указатель базы стекового кадра) т.к. дальше значение регистра будет меняться на базу текущего стекфрейма, сохраняем в `ebp` значение регистра `esp` (адрес вершины стека)

```
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp, esp
+----------------+
```

Выделяем необходимую память под локальные переменные и аргументы для вызова функции `foo`. Локальных переменных и аргументов у нас нет. Но т.к. стек имеет выравнивание в 16 байт и на стеке уже лежит 8 байт (4 для адреса возврата и 4 для сохраненного `ebp`) выделяем дополнительные 8 байт, смещая указатель стека.

```
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp
+----------------+
| ...            |
| 8 byte padding |
| ...            |     <-- esp
-----------------+
```

Вызываем функцию foo. Команда `call` сохраняет на стеке адрес возврата и передает управление вызываемой функции. На стороне функции foo пушим на стек значение регистра `ebp` (указатель базы стекового кадра) и сохраняем в ebp значение регистра `esp` (адрес вершины стека), инициализируем новый стекфрйем.

```
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp
+----------------+
| ...            |
| 8 byte padding |
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp, esp
+----------------+
```

Выделяем необходимую память под локальные переменные и аргументы для вызова функции `bar`. У нас две локальные переменные типа `int` — это 8 байт, два аргумента для функции `bar` типа `int` — это 8 байт. И т.к. у нас уже есть на стеке 8 байт (адрес возврата и сохраненный `ebp`) нужно выделить еще 8 байт чтобы соблюсти требования к выравниванию. Таким образом всего выделяем `8 + 8 + 8 = 24` байта, смещая указатель стека.

```
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp
+----------------+
| ...            |
| 8 byte padding |
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp
+----------------+
| local a        |     <-- ebp - 4
+----------------+
| local b        |     <-- ebp - 8
+----------------+
| ...            |
```

```
| 8 byte padding |
| ...            |
+----------------+
| arg a          |     <-- esp + 4
+----------------+
| arg b          |     <-- esp
+----------------+
```

Вызываем функцию `bar`. Все работает так же, как и при вызове функции `foo`. Команда `call` сохраняет на стеке адрес возврата и передает управление вызываемой функции. На стороне функции `bar` пушим на стек значение регистра `ebp` и сохраняем в `ebp` значение регистра `esp`, инициализируем новый стекфрейм.

```
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp
+----------------+
| ...            |
| 8 byte padding |
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp
+----------------+
| local a        |     <-- ebp - 4
+----------------+
| local b        |     <-- ebp - 8
+----------------+
| ...            |
| 8 byte padding |
| ...            |
+----------------+
| arg a          |     <-- ebp + 12
+----------------+
| arg b          |     <-- ebp + 8
+----------------+
| return address |
+----------------+
```

```
| saved rbp      |     <-- ebp, esp
+----------------+
```

Функция `bar` ничего не делает. Восстанавливаем значение указателя базы предыдущего стекового кадра `ebp` (вызывающей стороны, функция `foo`) и удаляем сохраненное значение со стека, смещая указатель вершины стека на 4 байта вверх. Забираем со стека адрес возврата и удаляем сохраненное значение со стека таким же смещением указателя вершины стека на 4 байта вверх. Передаем управление обратно функции `foo`.

```
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp
+----------------+
| ...            |
| 8 byte padding |
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp
+----------------+
| local a        |     <-- ebp - 4
+----------------+
| local b        |     <-- ebp - 8
+----------------+
| ...            |
| 8 byte padding |
| ...            |
+----------------+
| arg a          |     <-- esp + 4
+----------------+
| arg b          |     <-- esp
+----------------+
```

Функция `foo` после вызова `bar` завершает свою работу. Удаляем локальные переменные и аргументы предыдущего вызова, смещаем указатель вершины обратно на 24 байта вверх.

```
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp
+----------------+
| ...            |
| 8 byte padding |
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp, esp
+----------------+
```

Восстанавливаем значение указателя базы предыдущего стекового кадра `ebp` (вызывающей стороны, функция `main`) и удаляем сохраненное значение со стека. Забираем со стека адрес возврата и удаляем сохраненное значение со стека. Передаем управление обратно функции `main`.

```
| ...            |
+----------------+
| return address |
+----------------+
| saved rbp      |     <-- ebp
+----------------+
| ...            |
| 8 byte padding |
| ...            |     <-- esp
-----------------+
```

`main` завершает свою работу, выполняя ровно те же действия, что и предыдущие вызовы и мы возвращаемся в исходное состояние.

```
| ...            |
+----------------+
| return address |
+----------------+
```

Мы видим, что функции между вызовами никак не сохраняют свое локальное состояние, каждый следующий вызов создаёт новый кадр и удаляет его по завершению работы.

Классификация

Помимо фундаментального отличия сопрограмм в способности сохранять свое состояние, можно выделить три основные конструктивные особенности.

1. Способ передачи управления;
2. Способ представления в языке;
3. Способ локализации внутреннего состояния (состояния выполнения).

По способу передачи управления сопрограммы можно разделить на *симметричные (symmetric)* и *асимметричные (asymmetric, semi-symmetric)*.

Симметричные сопрограммы обеспечивают единый механизм передачи управления между друг другом, являются равноправными, работая на одном иерархическом уровне. В этом случае сопрограмма должна явно указывать кому она передаёт управление: другую сопрограмму, приостанавливая свое выполнение и ожидая пока ей вернут контроль таким же образом. По сути, стек и вложенная природа вызовов функций заменяется на множество приостановленных, равноправных сопрограмм и одной активной, которая может передать управление любой другой сопрограмме.

В то время как асимметричные сопрограммы предоставляют две операции для передачи управления: одна для вызова, передача управления вызываемой сопрограмме и одна для приостановки выполнения, последняя возвращает контроль вызывающей стороне.

Стоить заметить что обе концепции обладают одинаковой выразительной силой. Асимметричные сопрограммы семантически ближе к функциям, в то время как симметричные интуитивно понятнее в контексте кооперативной многозадачности.

По способу представления в языке сопрограммы могут быть представлены как объекты *первого класса (first-class object, first-class citizen)* или как *ограниченные низкоуровневые языковые конструкции (constrained, compiler-internal)*, скрывающие детали реализации и предоставляющие управляющие описатели (handles), дескрипторы.

Объекты первого класса в контексте языка программирования — это сущности, которые могут быть сохранены в переменные, могут передаваться в функции как аргументы или возвращаться в качестве результата, могут быть созданы в рантайме и не зависят от именования (внутренне самоопознаваемы). Например, нельзя создавать функции во

время выполнения программы, поэтому функции не являются объектами первого класса. В то же время, существует понятие функционального объекта (function object): пользовательский тип данных, реализующий эквивалентную функциям семантику, который является объектом первого класса.

По способу локализации внутреннего состояния сопрограммы можно разделить на *стековые (stackful)* и *стеконезависимые (stackless)*. Чтобы понять детальнее, что лежит в основе такого разделения, необходимо дать некоторую классификацию аппаратному стеку (proccesor stack).

Аппаратный стек может быть назначен разным уровням приложения:

- *Стек приложения (Application stack)*. Принадлежит функции `main`. Система управления памятью операционной системы может определять переполнения или недопустимые аллокации такого стека. Стек расположен в адресном пространстве таким образом, что его можно расширять по мере необходимости;
- *Стек потока выполнения (Thread stack)*. Стек назначенный явно запущенному потоку. Обычно используются стеки фиксированного размера (до 1-2 мб);
- *Стек контекста выполнения (Side stack)*. Контекст (Execution context) это некоторое окружение, пользовательский поток управления или функция (top level context function, контекстная функция верхнего уровня) со своим назначенным стеком. Стек обычно выделяется в пользовательском режиме (библиотечным кодом), а не операционной системой. Контекст имеет свойство сохранять и восстанавливать свое состояние выполнения: регистры центрального процессора, счётчик команд и указатель стека, что позволяет в пользовательском режиме переключатся между контекстами.

Каждая стековая сопрограмма использует отдельный контекст выполнения и назначенный ему стек. Передача управления между такими сопрограммами — это переключение контекста (переключение пользовательского контекста достаточно быстрая операция, в сравнение с переключением контекстов операционной системы: между потоками) и соответственно активного стека. Важное следствие из стековой архитектуры таких сопрограмм заключается в том, что из-за того, что все фреймы вложенных вызовов полностью сохраняются на стеке, выполнение сопрограммы может быть приостановлено в любом из этих вызовов.

Приведем пример простой сопрограммы, мы воспользуемся семейством функций для управления контекстами: `getcontext`, `makecontext` и `swapcontext` (см. Complete Context Control)

```cpp
#include <iostream>
#include <ucontext.h>

static ucontext_t caller_context;
static ucontext_t coroutine_context;

void print_hello_and_suspend()
{
     // выводим Hello
    std::cout << "Hello";
    // точка передачи управления вызывающей стороне,
    // переключаемся на контекст caller_context
    // в контексте сопрограммы coroutine_context сохраняется текущая точка выпо
    // после возвращения контроля, выполнение продолжится с этой точки.
    swapcontext(&coroutine_context, &caller_context);
}

void simple_coroutine()
{
    // точка первой передачи управления в coroutine_context
    // чтобы продемонстрировать преимущества использование стека
    // выполним вложенный вызов функции print_hello_and_suspend.
    print_hello_and_suspend();
    // функция print_hello_and_suspend приостановила выполнение сопрограммы
    // после того как управление вернётся мы выведем Coroutine! и завершим рабо
    // управление будет передано контексту,
    // указатель на который хранится в coroutine_context.uc_link, т.е. caller_c
    std::cout << "Coroutine!" << std::endl;
}

int main()
{
    // Стек сопрограммы.
    char stack[256];

    // Инициализация контекста сопрограммы coroutine_context
    // uc_link указывает на caller_context, точку возврата при завершении сопро
    // uc_stack хранит указатель и размер стека
    coroutine_context.uc_link          = &caller_context;
    coroutine_context.uc_stack.ss_sp   = stack;
    coroutine_context.uc_stack.ss_size = sizeof(stack);
    getcontext(&coroutine_context);

    // Заполнение coroutine_context
```

```
    // Контекст настраивается таким образом, что переключаясь на него
    // исполнение начинается с точки входа в функцию simple_coroutine
    makecontext(&coroutine_context, simple_coroutine, 0);

    // передаем управление сопрограмме, переключаемся на контекст coroutine_con
    // в контексте caller_context сохраняется текущая точка выполнения,
    // после возвращения контроля, выполнение продолжится с этой точки.
    swapcontext(&caller_context, &coroutine_context);
    // сопрограмма приостановила свое выполнение и вернула управление
    // выводим пробел
    std::cout << " ";
    // передаём управление обратно сопрограмме.
    swapcontext(&caller_context, &coroutine_context);

    return 0;
}
```

Отметим, что контексты выполнения лежат в основе реализации стековых сопрограмм библиотеки Boost: Boost.Coroutine, Boost.Coroutine2, только в Boost по умолчанию вместо `ucontext_t` используется `fcontext_t` — собственная, более производительная реализация(ассемблерная, с ручным сохранением/восстановлением регистров, без системных вызовов) POSIX стандарта.

В случае стеконезависимых сопрограмм контекст выполнения и назначенный ему стек не используются, сопрограмма выполняется на стеке вызывающей стороны, как обычная подпрограмма, до момента передачи управления. В силу того что стековые кадры имеют строго вложенную структуру и время жизни, такие сопрограммы, приостанавливая выполнение, сохраняют свое состояние, динамически размещая его в куче. Возобновление работы приостановленной сопрограммы, не отличается от вызова обычной подпрограммы, с тем исключением, что регистры и переменные восстанавливаются из сохраненного состояния также как и счетчик команд, т.е. управление передается в точку последней остановки. Структурно такие сопрограммы похожи на устройство Даффа и машину состояний.

Можно вывести несколько важных следствий стеконезависимых сопрограмм:

1. Передача управления возможна только из самой сопрограммы (top level function), все вложенные вызовы к этому моменту должны быть завершены;
2. Передача управления возможна только вызывающей стороне;

3. Возобновление работы сопрограммы происходит на стеке вызывающей стороны, он может отличатся от стека первоначального вызова, это может быть даже другой поток;
4. Для сохранения состояния (определения набора переменных), восстановления кадра и генерации шаблонного кода для возобновления работы с точки последней остановки, необходима поддержка со стороны стандартов и компиляторов языка.

Мы описали общую теорию и классификацию сопрограмм, рассмотрим техническую спецификацию сопрограмм нового C++20, какое место они занимают в общей теории, их особенности, семантику и синтаксис.

C++20.

Техническая спецификация сопрограмм в новом C++ носит название **Coroutine TS**. Coroutine TS предоставляет низкоуровневые средства обеспечивающие характерную возможность передачи управления, описывает обобщенный механизм взаимодействия и настройки сопрограмми и набор вспомогательных высокоуровневых типов стандартной библиотеки, задача которых сделать разработку сопрограмм более доступной и безопасной.

Подход который применяется для реализации обобщенных механизмов уже встречается и используется стандартом. Это *range based for*, суть его в том что компилятор генерирует код цикла, вызывая определенный набор методов строго описанным способом, в данном случае это методы `begin` и `end`, тем самым давая возможность программистам настраивать необходимое поведение цикла, определяя эти методы и тип итератора, который они возвращают. Точно также компилятор генерирует код сопрограммы, вызывая в строго определенный момент методы определенных пользователем типов, позволяя полностью настраивать и контролировать поведение сопрограммы.

В описанной нами классификации предоставляемые средства подпадают под определение *compile-internal* сопрограмм.

Далее, чтобы дать возможность компилятору более эффективно работать с памятью, оптимизировать размер и выравнивание сохраняемых состояний используется стеконезависимая модель. Более того, во-первых, выделенная ранее память может переиспользовать, по сути, работая в режиме высокопроизводительного блочного аллокатора, во-вторых, *в случаях когда сопрограмма имеет строго вложенное время жизни и размер кадра известен на вызывающей стороне (как в примере с контекстами выполнения)*, компилятор может избавится от динамического размещения состояния в куче и хранить состояние на стеке вызывающей стороны если это обычная подпрограмма или внутри уже размещенного состояния вызывающей сопрограммы.

И в завершение, чтобы снизить сложность восприятия и сделать более понятным поток управления, сопрограммы семантически приближены к подпрограммам т.е. являются асимметричными.

В итоге, C++20 даёт нам возможность работать с **compile-internal asymmetric stackless coroutines**.

Мы начнем с того, что сделаем обзор тех выразительных средств, которые предоставляет новый стандарт для описания и работы с сопрограммами и будем постепенно углубляться в детали реализации.

New Keywords.

Для оперирования сопрограммами стандарт вводит три ключевых оператора:

- **co_await**. Унарный оператор, позволяющий, в общем случае, приостановить выполнение сопрограммы и передать управление вызывающей стороне, пока не завершатся вычисления представленные операндом;
- **co_yield**. Унарный оператор, частный случай оператора co_await, позволяющий приостановить выполнение сопрограммы и передать управление и значение операнда вызывающей стороне;
- **co_return**. Оператор завершает работу сопрограммы, возвращая значение, после вызова сопрограмма больше не сможет возобновить свое выполнение.

Если в определении функции встречается хотя бы один из этих операторов, то функция считается сопрограммой и обрабатывается соответствующим образом.

Сопрограммой не может быть:

- Функция `main`;
- Функция с оператором `return`;
- Функция помеченная `constexpr`;
- Функция с автоматическим выведением типа возвращаемого значения (auto);
- Функция с переменным числом аргументов (variadic arguments, не путать с variadic templates);
- Конструктор;

- Деструктор.

User types.

С сопрограммами ассоциировано несколько интерфейсных типов, позволяющих настраивать поведение сопрограммы и контролировать семантику операторов.

Promise.

Объект типа **Promise** позволяет настраивать поведения сопрограммы как программной единицы. Должен определять:

- Поведение сопрограммы при первом вызове;
- Поведение при выходе из сопрограммы;
- Стратегию обработки исключительных ситуаций;
- Необходимость в дополнительном уточнении типа выражения операторов co_await;
- Передача промежуточных и конечных результатов выполнения вызывающей стороне. Также тип promise участвует в разрешении перегрузки операторов new и delete, что позволяет настраивать динамическое размещение фрейма сопрограммы.
  Объект типа promise создаётся и хранится в рамках фрейма сопрограммы для каждого нового вызова.

Тип Promise определяется компилятором согласно специализации шаблона `std::coroutine_traits` по типу сопрограммы, в специализации участвует: тип возвращаемого значения, список типов входных параметров, тип класса, если сопрограмма представлена методом. Шаблон `std::coroutine_traits` определен следующем образом:

```
template <typename Ret, typename = std::void_t<>>
struct coroutine_traits_base
{};

template <typename Ret>
struct coroutine_traits_base<Ret, std::void_t<typename Ret::promise_type>>
{
    using promise_type = typename Ret::promise_type;
};
```

```
template <typename Ret, typename... Ts>
struct coroutine_traits : coroutine_traits_base<Ret>
{};
```

Тип должен иметь строгое имя **promise_type**. Из определения `std::coroutine_traits`, следует что существует как минимум одна специализация, которая ищет определение типа `promise_type` в пространстве имен типа возвращаемого результата. `promise_type` может быть как именем типа, так и псевдонимом.

Самый простой способ определения типа Promise для сопрограммы.

```
struct Task
{
    struct Promise
    {
        ...
    };
    using promise_type = Promise;
};
...

Task foo()
{
    ...
}
```

Также определение подобного типа `Task` полезно в более сложных ситуациях, тип может определять дополнительную семантику внешнего оперирования сопрограммой: передавать и получать данные, передавать поток управления (пробуждать) или уничтожать сопрограмму.

Другой способ определить тип Promise — это явно специализировать шаблон `std::coroutine_traits`. Это удобно, например, для сопрограмм представленных методом пользовательского типа

```
class Coroutine
{
public:
    void call(int);
```

```
};

namespace std
{
    template<>
    struct coroutine_traits<void, Coroutine, int>
    {
        using promise_type = Coroutine;
    };
}
```

Если тип Promise имеет конструктор соответствующий параметрам сопрограммы, то он будет вызван, иначе будет вызван конструктор по умолчанию. Важно, что все аргументы будут переданы как lvalues, это нужно для того чтобы мы не смогли *случайно* переместить данные из переданных аргументов в объект Promise т.к. мы ожидаем аргументы в теле сопрограммы. Более подробно создание объекта типа Promise мы рассмотрим ниже.

Прежде чем определить интерфейс типа Promise, необходимо описать второй тип ассоциированный со сопрограммой: **Awaitable**.

Awaitable.

Объекты типа **Awaitable** определяют семантику потока управления сопрограммы. Позволяют:

- Определить, следует ли приостанавливать выполнение сопрограммы в точке вызова оператора co_await;
- Выполнить некоторую логику после приостановления выполнения сопрограммы для дальнейшего планирования возобновления ее работы (асинхронные операции);
- Получить результат вызова оператора co_await, после возобновления работы.

Объект типа Awaitable определяется в результате разрешения перегрузки (overload resolution) и вызова оператора co_await. Если жизнеспособной перегрузки не было найдено, то результат вычисления самого операнда является объектом типа Awaitable. Далее вызов оператора транслируется в последовательность вызовов методов объекта данного типа.

Например, мы хотим создать механизм отложенного выполнения, который позволит функции уснуть на некоторое время, вернет управление вызывающей стороне, после чего продолжить свое выполнение через заданное время.

```
Task foo()
{
    using namespace std::chrono_literals;

    // выполнить некоторый набор операций
    // вернуть управление
    co_await 10s;
    // через 10 секунд выполнить еще один набор операций.
}
```

В этом примере выражение переданное в качестве операнда имеет тип `std::chrono::duration<long long>`, чтобы скомпилировать этот код нам нужно определить перегрузку оператора co_await для выражений такого типа.

```
template<typename Rep, typename Period>
auto operator co_await(std::chrono::duration<Rep, Period> duration) noexcept
{
    struct Awaitable
    {
        explicit Awaitable(std::chrono::system_clock::duration<Rep, Period> dur
            : duration_(duration)
        {}

        ...

    private:

        std::chrono::system_clock::duration duration_;
    };

    return Awaitable{ duration };
}
```

Внутри перегрузки мы описываем тип Awaitable, задача которого запланировать и вернуть управление сопрограмме через заданный промежуток времени, и возвращаем объект данного типа как результат.

Нам осталось определить интерфейс типа Awaitable, чтобы это сделать рассмотрим подробнее вызов оператора `co_await <expr>` и код, который компилятор генерирует в

месте вызова.

```
{
    // в начале мы определили тип Promise
    using coroutine_traits = std::coroutine_traits<ReturnValue, Args...>;
    using promise_type = typename coroutine_traits::promise_type;

    ...
    // вызов co_await <expr> в рамках сопрограммы

    // 1.
    // Создаем объект типа Awaitable, находим подходящую перегрузку оператора c
    // результат сохраняем во фрейме сопрограммы (как создается фрейм мы рассмо
    // в рамках описания типа Promise), это необходимо
    // т.к. с помощью Awaitable мы вернем результат вычисления, после возобновл
    frame->awaitable = create_awaitable(<expr>);

    // 2.
    // Вызываем метод await_ready().
    // Основная задача метода позволить нам избежать остановки сопрограммы
    // в случаях когда вычисления могут быть завершены синхронно
    // или уже завершены, сохранив вычислительные ресурсы.
    if (!awaitable.await_ready())
    {
        // 3.
        // Если вызов await_ready() вернул false,
        // то сопрограмма приостанавливает свое выполнение,
        // сохраняет состояние: состояние локальных переменных, точку остановки
        // (это идентификатор состояния, на которое сопрограмма перейдет
        // после возобновления своей работы,
        // достаточная информация что бы перейти в точку <resume-point>)

        <suspend-coroutine>

        // 4.
        // Определяем тип coroutine_handle
        // corotine_handle - это дескриптор фрейма сопрограммы.
        // он обеспечивает низкоуровневую функциональность оперирования сопрогр
        // передача управления (возобновление выполнения) и удаление.

        using handle_type = std::coroutine_handle<promise_type>;
        using await_suspend_result_type =
            decltype(frame->awaitable.await_suspend(handle_type::from_promise(p
```

```
        // 5.
        // Вызов метода await_suspend(handle),
        // задача метода await_suspend выполнить некоторую логику
        // на клиентской стороне после приостановления выполнения сопрограммы
        // для дальнейшего планирования возобновления ее работы (если необходим
        // Метод принимает один аргумент - дескриптор сопрограммы.
        // Тип возвращаемого результата, определяет семантику передачи управлен

        if constexpr (std::is_void_v<await_suspend_result_type>)
        {
            // Тип возвращаемого результата void,
            // мы безусловно передаем управление вызывающей стороне
            // (под вызывающей стороной здесь понимается сторона,
            // которая передала управление сопрограмме)
            frame->awaitable.await_suspend(handle_type::from_promise(promise));
            <return-to-caller-or-resumer>;
        }
        else if constexpr (std::is_same_v<await_suspend_result_type, bool>)
        {
            // Тип возвращаемого результата bool,
            // если метод вернул false, то управление не передается вызывающей
            // и сопрограмма возобновляет свое выполнение
            // Это полезно, например, когда асинхронная операция
            // инициированная объектом Awaitable завершилась синхронно
            if (frame->awaitable.await_suspend(handle_type::from_promise(promis
                <return-to-caller-or-resumer>;
        }
        else if constexpr (is_coroutine_handle_v<await_suspend_result_type>)
        {
            // Тип возвращаемого результата std::coroutine_handle<OtherPromise>
            // т.е. вызов возвращает дескриптор другой сопрограммы,
            // то мы передаем управление этой сопрограмме, это семантика позвол
            // эффективно реализовывать симметричный механизм передачи потока
            // управления между сопрограммами
            auto&& other_handle = frame->awaitable.await_suspend(
                handle_type::from_promise(promise));
            other_handle.resume();
        }
        else
        {
            static_assert(false);
        }
    }
```

```
    // 6.
    // Точка возобновления выполнения (пробуждения)
    // Вызов метода await_resume(). Задача метода получить результат вычисления
    // Возвращаемое значение рассматривается как результат вызова оператора co_
resume_point:
    return frame->awaitable.await_resume();
}
```

Здесь есть несколько важных замечаний:

1. Если в процессе обработки возбуждается исключение, то исключение пробрасывается дальше, наружу оператора co_await. Если во время исключения выполнение сопрограммы было приостановлено, то исключение перехватывается, сопрограмма автоматически возобновляет свое выполнение и только после этого пробрасывается дальше;
2. Крайне важно, что сопрограмма полностью останавливает свое выполнение до вызова метода `await_suspend` и передачи дескриптора сопрограммы пользовательскому коду. В этом случае дескриптор сопрограммы может свободно передаваться между потоками выполнения без дополнительной синхронизации. Например, дескриптор может быть передан в запланированную в пуле-потоков асинхронную операцию. Конечно здесь следует очень внимательно следить за тем, в какой момент метода `await_suspend` мы передаем дескриптор другому потоку и как другой поток оперирует этим дескриптором. Поток получивший дескриптор может возобновить выполнение сопрограммы до того как мы вышли из `await_suspend`. После возобновление работы и вызова метода `await_resume`, объект Awaitable может быть удален. Также потенциально фрейм и объект Promise может быть удалены, до того как мы завершим метод `await_suspend`. Поэтому основное чего следует избегать, после передачи контроля над сопрограммой другому потоку в `await_suspend`: это не обращаться к полям (this может быть удален) и объекту Promise, они могут быть уже удалены.

Формально концепцию Awaitable можно определить в терминах *type-traits* примерно так:

- is_awaitable

Дополним предыдущий пример:

```
template<typename Rep, typename Period>
auto operator co_await(std::chrono::duration<Rep, Period> duration) noexcept
{
    struct Awaitable
    {
        explicit Awaitable(std::chrono::system_clock::duration duration)
            : duration_(duration)
        {}

        bool await_ready() const noexcept
        {
            return duration_.count() <= 0;
        }

        void await_resume() noexcept
        {}

        void await_suspend(std::coroutine_handle<> h)
        {
            // Реализация timer::async в данном контексте не очень интересна.
            // Важно что это асинхронная операция, которая через заданный
            // промежуток времени вызовет переданный callback.
            timer::async(duration_, [h]()
            {
                h.resume();
            });
        }

    private:

        std::chrono::system_clock::duration duration_;
    };

    return Awaitable{ duration };
}

// сопрограмма, которая через каждую секунду будет выводить текст на экран
Task tick()
{
    using namespace std::chrono_literals;

    co_await 1s;
```

```
    std::cout << "1..." << std::endl;

    co_await 1000ms;
    std::cout << "2..." << std::endl;
}

int main()
{
    tick();
    std::cin.get();
}
```

1. Вызываем функцию `tick`;
2. Находим нужную перегрузку оператора co_await и создаем объект Awaitable, передаем в конструктор временной интервал в 1 секунду;
3. Вызываем метод `await_ready`, проверяем необходимо ли ожидание;
4. Приостанавливаем работу функции `tick`, сохраняем состояние;
5. Вызываем метод `await_suspend` и передаем дескриптор сопрограммы;
6. Метод await_suspend инициирует асинхронную операцию `timer::async`, которая ожидает заданное время и вызывает переданный `callback`. Предаем в `callback` дескриптор сопрограммы чтобы после ожидания передать ей управление;
7. Передаем управление вызывающей стороне — функции `main`;
8. Функция `main` вызывает метод стандартного потока ввода `get`, это синхронная операция, ожидающая ввода. Мы висим, чтобы просто дать завершиться инициированным асинхронным операциям;
9. Ждем одну секунду, асинхронная операция вызывает переданный нами `callback`, вызов осуществляется в том же потоке, в котором происходило ожидание;
10. Вызываем метод `resume` у дескриптора. Метод передает управление сопрограмме: вызывается функция `tick` на стеке потока, восстанавливаем сохраненное во фрейме состояние, управление передается в точку последней остановки;
11. Вызывается метод `await_resume` у объекта Awaitable, созданного при вызове оператора co_await и сохраненного во фрейме;
12. Метода `await_resume` ничего не делает и не возвращает результата, оператор co_await завершает свою работу и передает управление, следующей за ним команде;
13. Функция `tick` выводит сообщение на экран с помощью стандартного потока вывода "1...";

14. Вызов следующего оператора co_await. Выполняем все шаги начиная с пункта 2. Отличие только в том, что управление возвращается не функции `main`, а асинхронной операции, которая вызвала наш `callback`, т.е. resumer'y. После это асинхронная операция завершает свое выполнение;
15. Сопрограмма `tick` завершает свое выполнение (более детально этот процесс мы рассмотрим ниже)

После того как мы определили интерфейс и семантику оператора co_await и типа Awaitable, мы можем вернутся к более детальному изучению типа Promise и настройке сопрограммы.

Promise.

Также как и в случае с Awaitable, чтобы понять роль объекта Promise мы начнем с кода, который генерирует компилятор в процессе обработки сопрограммы.

Генерируемый код можно разделить на три часть:

1. Создание и инициализация кадра сопрограммы. Инициация выполнения сопрограммы. Т.е. это код который выполняется при первом вызове сопрограммы;
2. Описание стейт-машины согласно пользовательским запросам передачи управления (вызовы операторов `co_await` / `co_yield` / `co_return`). Это код, который выполняется при передаче управления сопрограмме т.е. при первом вызове и при возобновление работы;
3. Завершение выполнения, освобождение ресурсов и удаление кадра. Код выполняется при естественном завершении или принудительном удалении.

```
// Примерная организация кадра сопрограммы.
// Здесь отражены наиболее важные для понимая части
// 1. resume - указатель на функцию,
//    которая вызывается при передаче управления сопрограмме, описывает стейт-ма
// 2. promise - объект типа Promise
// 3. state - текущее состояние
// 4. heap_allocated - был ли фрейм при создание размещен в куче
//    или фрейм был создан на стеке вызывающей стороны
// 5. args - аргументы вызова сопрограммы
// 6. locals - сохраненные локальные переменные текущего состояния
// ...
```

```
struct coroutine_frame
{
    void (*resume)(coroutine_frame *);
    promise_type promise;
    int16_t state;
    bool heap_allocated;
    // args
    // locals
    //...
};

// 1. Создание и инициализация кадра сопрограммы. Инициация выполнения.
template<typename ReturnValue, typename ...Args>
ReturnValue Foo(Args&&... args)
{
    // 1.
    // Определяем тип Promise
    using coroutine_traits = std::coroutine_traits<ReturnValue, Args...>;
    using promise_type = typename coroutine_traits::promise_type;

    // 2.
    // Создание кадра сопрограммы.
    // Размер кадра определяется встроенными средствами компилятора
    // и зависит от размера объекта Promise, количества и размера локальных пер
    // и аргументов, и набора вспомогательных данных,
    // необходимых для управления состоянием сопрограммы.
    // 1. Если тип promise_type имеет статический метод
    //    get_return_object_on_allocation_failure,
    //    то вызывается версия оператора new, не генерирующая исключений
    //    и в случае неудачи вызывается метод get_return_object_on_allocation_f
    //    результат вызова возвращается вызывающей стороне.
    // 2. Иначе вызывается обычная версия оператора new.
    coroutine_frame* frame = nullptr;
    if constexpr (has_static_get_return_object_on_allocation_failure_v<promise_
    {
        frame = reinterpret_cast<coroutine_frame*>(
            operator new(__builtin_coro_size(), std::nothrow));
        if(!frame)
            return promise_type::get_return_object_on_allocation_failure();
    }
    else
    {
        frame = reinterpret_cast<coroutine_frame*>(operator new(__builtin_coro_
    }
```

```
    // 3.
    // Сохраняем переданные функции аргументы во фрейме.
    // Аргументы переданные по значению перемещаются.
    // Аргументы переданные по ссылке (lvalue и rvalue) сохраняют ссылочную сем
    <move-args-to-frame>

    // 4.
    // Создаем объект типа promise_type и сохраняем его во фрейме
    new(&frame->promise) create_promise<promise_type>(<frame-lvalue-args>);

    // 5.
    // Вызываем метод Promise::get_return_object().
    // Результат вычисления будет возвращен вызывающей стороне
    // при достижение первой точки остановки и передачи потока управления.
    // Результат сохраняется как локальная переменная до вызова тела функции,
    // т.к. фрейм сопрограммы может быть удален (см. оператор co_await).
    auto return_object = frame->promise.get_return_object();

    // 6.
    // Вызываем функцию описывающую стейт-машину согласно
    // пользовательским запросам передачи управления
    // В реализации GCC, например, эти две функции называются
    // ramp-fucntion (создание и инициализация) и
    // action-function (пользовательская стейт-машина) соответственно
    void couroutine_states(coroutine_frame*);
    couroutine_states(frame);

    // 7.
    // Возвращаем результат вызывающей стороне,
    // мы достигнем этой точки в коде только при первом вызове,
    // все последующие запросы на возобновление работы будут вызывать функцию
    // стейт-машины couroutine_states, указатель на функцию сохранен во фрейме
    return return_object;
}
```

Мы упоминали выше что тип Promise участвует в разрешение перегрузки операторов `new` и delete. Например, при таком определение, будут вызваны пользовательские операторы `new` и `delete`:

```
struct Promise
{
```

```
    void* operator new(std::size_t size, std::nothrow_t) noexcept
    {
        ...
    }

    void operator delete(void* ptr, std::size_t size)
    {
        ...
    }

    // определяем поведение сопрограммы если не удалось создать фрейм
    static auto get_return_object_on_allocation_failure() noexcept
    {
        // создаем и возвращаем вызывающей стороне невалидный объект
        return make_invalid_task();
    }
};
```

Более того у нас есть возможность добавить перегрузку оператора `new` с дополнительными аргументами, набор параметров должен быть согласован с параметрами функции. Это позволяет использовать, например, стандартные механизмы такие как leading-allocator convention.

```
// тип Promise с перегрузкой оператора new с пользовательским аллокатором
template<typename Allocator>
struct Promise : PromiseBase
{
    // std::allocator_arg_t - это tag-тип
    // нужен для устранения неоднозначных ситуаций при перегрузке
    void* operator new(std::size_t size, std::allocator_arg_t, Allocator alloca
    {
        ...
    }

    void operator delete(void* ptr, std::size_t size)
    {
        ...
    }
};

// добавляем соответствующую специализацию в std::coroutine_traits
namespace std
```

```
{
    template<typename... Args>
    struct coroutine_traits<Task, Args...>
    {
        using promise_type = PromiseBase;
    };

    template<typename Allocator>
    struct coroutine_traits<Task, std::allocator_arg_t, Allocator>
    {
        using promise_type = Promise<Allocator>;
    };
}

// мы можем вызывать сопрограммы с передачей конкретного аллокатора
int main()
{
    MyAlloc alloc;
    coro(std::allocator_arg, alloc);
    ...
}
```

Независимо от того, перегрузим мы операторы `new` и `delete` или нет, компилятор может избавиться от динамического размещения кадра и сохранить состояние на стеке вызывающей стороны.

Следующим шагом мы сохраняем аргументы сопрограммы во фрейм. Здесь есть несколько важных замечаний.

Во-первых, нам следует быть крайне осторожными с передачей ссылок. Важно понимать, что сопрограмма *никак не меняет время жизни переданных аргументов, сохраняя ссылочную семантику*, но при этом время жизни кадра сопрограммы отличается от стекового кадра обычной функции. Следующий, казалось бы простой пример, в случае с сопрограммой, является случаем Undefined Behavior.

```
void Coroutine(const std::vector<int>& data)
{
    co_await 10s;
    for(const auto& value : data)
        std::cout << value << std::endl;
}
```

```
void Foo()
{
    // 1. Мы передаем сопрограмме  временный объект типа vector<int>;
    // 2. Ссылка на этот временный объект сохраняется в константной ссылке data
    // 3. Аргументы вызова сохраняются во фрейме сопрограммы, т.к. ссылочная се
    //    сохраняется, то поле, в котором мы сохранили data
    //    будет указывать на тот же временный объект;
    // 4. Временные объекты удаляются после полного вычисления выражения;
    // 5. Выражение, в котором участвует временный объект типа vector<int>,
    //    будет вычислено, когда оператор co_await вернет управление вызывающей
    //    и функция Foo продолжит свое выполнение;
    // 6. Через 10 секунд, когда сопрограмма вновь получит поток управления и
    //    продолжит свое выполнение с цикла, вектор, на который ссылается data
    //    (поле фрейма, в котором сохранена ссылка), будет уже удален.
    Coroutine({1, 2, 3});
    ...
}
```

Второе замечание касается порядка сохранения аргументов и создания объекта типа Promise. Нужно иметь ввиду, что все дальнейшее оперирование с аргументами в теле сопрограммы осуществляется с копиями аргументов, сохраненных в кадре сопрограммы, поэтому мы сначала сохраняем аргументы, а потом создаем объект типа Promise и вызываем (если нашли) соответствующим перегруженный конструктор, в конструктор передаем именно скопированные аргументы, а не изначальные объекты. Это позволяет сохранить естественное восприятие потока управления и избежать неприятных эффектов, если Promise в конструкторе сохранил ссылку на какой-нибудь аргумент, тогда объект, на который мы будем ссылаться из Promise, будет существовать на протяжение всего времени жизни кадра сопрограммы.

Далее мы обращаемся к объекту типа Promise и вызываем метод `get_return_object`. Создаваемый объект не обязательно должен в точности соответствовать типу возвращаемого результата, по необходимости и возможности может быть выполнено неявное преобразование. В стандарте нет требования в какой момент выполнять неявное преобразование: в момент создания объекта и вызова `get_return_object` или в момент возвращения вызываемой стороне. Это важно, например, если мы используем какие-то операции с побочными эффектами, реализуя императивную последовательность выполнения. Пример эксплуатации такого преобразования Monadic composition.

```
class Task
{
```

```
public:

    struct promise_type
    {
        auto get_return_object() noexcept
        {
            return Task{ std::coroutine_handle<promise_type>::from_promise(*thi
        }
        ...
    };

    void resume()
    {
        if(coro_handle)
            coro_handle.resume();
    }

private:

    Task() = default;
    explicit Task(std::coroutine_handle<> handle)
        : coro_handle(handle)
    {}

    std::coroutine_handle<> coro_handle;
};
```

Мы уже встречали тип `std::coroutine_handle` — эти дескриптор сопрограммы, обеспечивает низкоуровневую функциональность оперирования сопрограммой: передача управления (возобновление выполнения) и удаление. Статический метод `from_promise`, позволяет получить дескриптор сопрограммы по объекту Promise.

В заключение мы вызываем функцию `couroutine_states` и передаем управление стейт-машине, после того как функция `couroutine_states` вернет управление, приостановив или завершив выполнение сопрограммы, мы возвращаем ранее созданный методом `get_return_object` объект вызывающей стороне.

State Machine.

Функция `couroutine_states` описывает стейт-машину согласно пользовательскому набору вызовов операторов co_await/co_yield/co_return и вызывается автоматически при передаче управления сопрограмме: при первом вызове или при возобновление работы,

вызовом метода дескриптора `resume`. Указатель на функцию сохранен в кадре сопрограммы.

```
void couroutine_states(coroutine_frame* frame)
{
    switch(frame->state)
    {
        case 0:
        ... goto resume_point_0;
        case N:
            goto resume_point_N;
        ...
    }

    co_await promise.initial_suspend();

    try
    {
        // function body
    }
    catch(...)
    {
        promise.unhandled_exception();
    }

final_suspend:
    co_await promise.final_suspend();
}
```

Если вернутся к коду, который генерируется вызовом оператора co_await, то можно заметить метку `resume_point` — точка пробуждения.

```
{
    ...
resume_point:
    return frame->awaitable.await_resume();
}
```

Для каждого вызова оператора co_await в служебном или пользовательском коде генерируется уникальная метка — порядковый номер оператора co_await, этот номер

также идентифицирует текущие состояние сопрограммы и сохраняется в кадре в поле `state`, перед передачей управления вызывающей стороне. И первое что мы делаем при вызове — это смотрим на текущее состояние, безусловно переходим к соответствующей ветке, возобновляем работу и завершаем вызов оператора co_await в пользовательском коде.

При первом вызове сопрограммы состояние не будет задано и мы перейдем к вызову метода `initial_suspend` с последующей передачей результата оператору co_await. Задача этого вызова определить: следует ли начать выполнение пользовательского кода немедленно или выполнение должно быть отложено. Стандарт предоставляет два тривиальных типа реализующих концепцию Awaitable: `std::suspend_never`, `std::suspend_always`, которые упрощают реализацию метода `initial_suspend`, позволяя реализовать две наиболее распространенных модели поведения.

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

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

// В данном случае, вызов сопрограммы приводит к немедленному выполнению
// пользовательского кода
class Task
{
public:
    struct promise_type
    {
        ...
        auto init_suspend() const noexcept
        {
            return std::suspend_never{};
```

```
        }
    }
    ...
};

// В этом же случае, при вызове сопрограммы
// управление сразу передается вызывающей стороне
// и выполнение пользовательского кода будет отложено,
// до явной передачи управление обратно через вызов метода resume.
class TaskManual
{
public:
    struct promise_type
    {
        ...
        auto init_suspend() const noexcept
        {
            return std::suspend_always{};
        }
    }
    ...
};
```

Далее следует выполнение пользовательского кода. Если включена поддержка исключений, то пользовательский код заключается в блок `try-catch` с вызовом метода `unhandled_exception` в случае возбуждения исключения.

В пользовательском коде, помимо оператора co_await, могут встречаться операторы co_yield и co_return. Эти операторы позволяют передавать вызываемой стороне промежуточный или конечный результат выполнения сопрограммы. Передача результата осуществляется средствами объекта Promise, но с разной семантикой.

Оператор `co_yield <expr>` эквивалентен вызову:

```
co_await frame->promise.yield_value(<expr>);
```

Т.е. оператор позволяет сохранить промежуточный результат в объекте Promise и вернуть управление вызывающей стороне для дальнейшей манипуляции полученными данными. Типичная реализация метода `yield_value` выглядит так:

```
template<typename Type>
class Task
{
public:
    struct promise_type
    {
        ...
        // Сохраняем переданное сопрограммой значение,
        // передаем управление вызывающей стороне,
        // возвращая и передавая оператору co_await объект типа std::suspend_al
        auto yield_value(Type value)
        {
            current_value = std::move(value);
            return std::suspend_always{};
        }
    };
    ...
};
```

Пользовательский тип `Task` может реализовывать разные стратегии получения доступа к сохраненным в объекте `Promise` значениям. Один из наиболее выразительных вариантов использования семантики оператора `co_yield` — это генераторы. Пример генератора из библиотеки cppcoro

В свою очередь оператор co_return в контексте пользовательского кода обладает такой же семантикой как и оператор `return`. Позволяет принудительно завершить выполнение и вернуть результат вызывающей стороне.

- Вызов co_return без операндов эквивалентен:

```
// co_return;
frame->promise.return_void();
goto final_suspend;
```

- Если тип результат вычисления выражения, переданного оператору в качестве аргумента, отличен от `void`, то вызов co_return эквивалентен следующему коду

```
// co_return <expr>;
frame->promise.return_value(<expr>);
goto final_suspend;
```

- Если же тип результата вычисления выражения `void`, то вызов генерирует следующий код

```
// co_return <expr>;
<expr>;
frame->promise.return_void();
goto final_suspend;
```

Важно, если в пользовательском коде нет операторов co_return, то в конце тела функции генерируется вызов оператора без аргументов `co_return;`. Т.е. выражение `frame->promise.return_void()` должно быть валидно.

После передачи результата вычислений в Promise через вызов методов `return_value` или `return_void`, мы завершаем выполнение пользовательского кода и переходим к служебному вызову `final_suspend`.

В противовес методу `initial_suspend`, вызов метода `final_suspend` позволяет настроить поведение сопрограммы в момент завершения своего выполнения. Задача вызова дать возможность пользователю обработать и опубликовать результат выполнения, сигнализировать о завершение вычислений или передать выполнение другой сопрограмме.

```
// В этом случае, после выполнения пользовательского кода сопрограмма завершит
// и все ресурсы будут удалены автоматически. Объекту Promise будет вызван дест
// аргументы будут удалены, память выделенная под кадр сопрограммы будет очищен
// вызовом оператора delete, затем управление будет передано вызывающей стороне
// Передача управления обратно сопрограмме приведет к Undefined Behavior.
// Это поведение полезно, когда мы не ожидаем результата выполнения сопрограммы
class Task
{
public:
    struct promise_type
    {
```

```
        ...
        auto final_suspend() const noexcept
        {
            // не передаем управление вызывающей стороне
            return std::suspend_never{};
        }
    };
    ...
};

// В этом же случае, после выполнения пользовательского кода сопрограмма переда
// управление вызывающей стороне и ответственность за удаление ресурсов сопрогр
// Передача управления обратно сопрограмме на этом этапе приведет к Undefined B
// Удаление ресурсов сопрограммы осуществляется принудительным вызовом
// на стороне пользователя coroutine_handle::destroy()
// Это стратегия необходима, когда нам нужно получить результат работы сопрогра
// в противном случае они будут удалены вместе с объектом Promise.
class TaskManual
{
public:
    struct promise_type
    {
        ...
        auto final_suspend() const noexcept
        {
            // передаем управление вызывающей стороне
            return std::suspend_always{};
        }
    }
    ...
};
```

Мы рассмотрели все случаи использования оператора co_await как в пользовательском так и в служебном коде. Он вызывается с результатами вызова `init_suspend` и `final_suspend`, приостанавливает работу сопрограммы в случае вызова оператора co_yield, может быть вызван в пользовательском коде с произвольным выражением. У последнего использования есть одна особенность. Если тип Promise определяет метод `await_transform`, то любой вызов оператора co_await в пользовательском коде транслируется в вызов

```
// co_await <expr>
co_await frame->promise.await_transform(<expr>);
```

Это достаточно мощный инструмент, например, помимо вызова оператора co_await с типами, которые не поддерживают концепцию Awaitable, мы можем запрещать использование или менять поведение уже определенных сущностей. Например можно запретить прямые вызовы оператора `co_await` в генераторе.

```
class Task
{
public:
    struct promise_type
    {
        ...
        template<typename Type>
        auto await_transform(Type&& Whatever) const noexcept
        {
            static_assert(false,
                "co_await is not supported in coroutines of type Generator");
            return std::suspend_never{};
        }
    };
    ...
};
```

Вместо заключения

Примеры кода:

- cppcoro. Библиотека примитивов асинхронной и кооперативной композиции построенная на сопрограммах Coroutine TS;
- folly. Реализована экспериментальная поддержка стандартных сопрограмм;

Возможное дальнейшее развитие:

- Coroutines TS Simplifications;
- Revisiting allocator model for coroutine;