C++23ジェネレータの紹介

2025/2/21
C++MIX #13

はじめに

誰？

twitter(X) @yohhoy / hatena id:yohhoy

何を？

C++23標準ライブラリ **std::generator** の機能紹介

どうして？

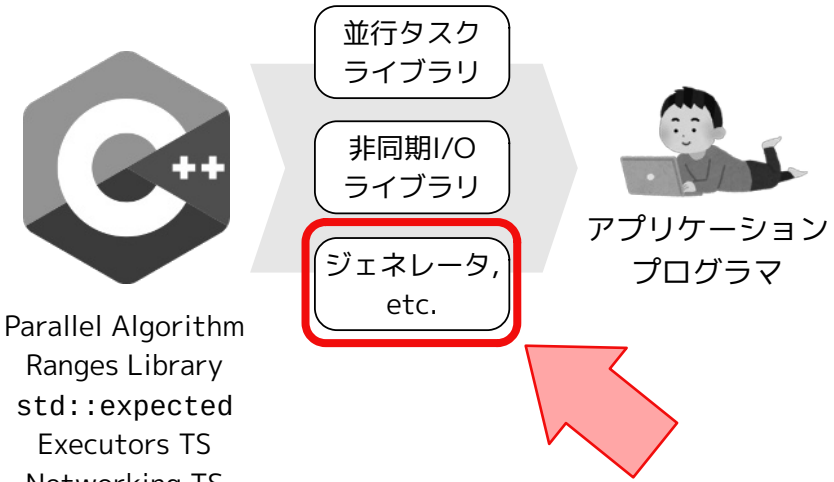
C++MIX #5 (2019年9月) コルーチン紹介の続編

「20分くらいでわかった気分になれるC++20コルーチン」

https://www.docswell.com/s/yohhoy/L57EJK-cpp20coro

コルーチン @ C++20

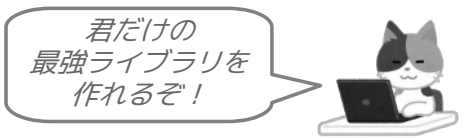
C++23, C++26, or someday



まとめ

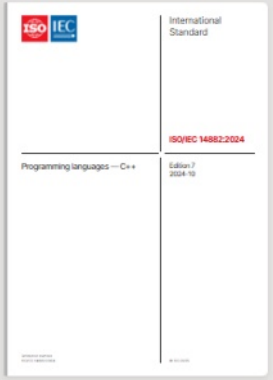
C++20コルーチンは
- 処理の**中断／再開をサポート**する関数
- 軽量な**スタックレス**コルーチン
- ライブラリ実装用の**低レベル部品のみ提供**
- 多数の**カスタマイズポイント**を規定

C++20以後のコルーチン・ライブラリ発展に期待



C++MIX #5「20分くらいでわかった気分になれるC++20コルーチン」より引用

We have C++23!



ISO/IEC 14882:2024

Programming languages — C++

Published (Edition 7, 2024)

26.8 Range generators **[coro.generator]**

26.8.1 Overview **[coroutine.generator.overview]**

1 Class template `generator` presents a view of the elements yielded by the evaluation of a coroutine.

2 A `generator` generates a sequence of elements by repeatedly resuming the coroutine from which it was returned. Elements of the sequence are produced by the coroutine each time a `co_yield` statement is evaluated. When the `co_yield` statement is of the form `co_yield elements_of(r)`, each element of the range `r` is successively produced as an element of the sequence.

[*Example 1*:

```
generator<int> ints(int start = 0) {
  while (true)
    co_yield start++;
}

void f() {
  for (auto i : ints() | views::take(3))
    cout << i << ' ';       // prints 0 1 2
}
```

— *end example*]

※ 通称”C++23”は1年遅れの ISO/IEC 14482:2024 として国際標準発行

<generator>

```
// [generator.syn] Header <generator> synopsis
namespace std {
  template<
    class Ref,
    class V = void,
    class Allocator = void>
  class generator;

  namespace pmr { ... }
}
```

※ pmr名前空間ではPolymorphic Memory Resource関連のエイリアスが定義される

std::generator

```
// [coro.generator.class] Class template generator
template<class Ref, class V = void, class Allocator = void>
class std::generator : public ranges::view_interface<generator<Ref, V, Allocator>> {
private:
  using value = conditional_t<is_void_v<V>, remove_cvref_t<Ref>, V>;
  using reference = conditional_t<is_void_v<V>, Ref&&, Ref>;
  class iterator;

public:
  using yielded =
    conditional_t<is_reference_v<reference>, reference, const reference&>;

  class promise_type;

  generator(const generator&) = delete;
  generator(generator&& other) noexcept;
  ~generator();
  generator& operator=(generator other) noexcept;

  iterator begin();
  default_sentinel_t end() const noexcept;
}
```

std::generator::promise_type

```
// [coro.generator.promise] Class generator::promise_type
template<class Ref, class V, class Allocator>
class std::generator<Ref, V, Allocator>::promise_type {
public:
  generator get_return_object() noexcept;

  suspend_always initial_suspend() const noexcept;
  auto final_suspend() noexcept;

  suspend_always yield_value(yielded val) noexcept;

  auto yield_value(const remove_reference_t<yielded>& lval)
    requires is_rvalue_reference_v<yielded> &&
      constructible_from<remove_cvref_t<yielded>,
                         const remove_reference_t<yielded>&>;

  template<class R2, class V2, class Alloc2, class Unused>
    requires same_as<typename generator<R2, V2, Alloc2>::yielded, yielded>
  auto yield_value(ranges::elements_of<generator<R2, V2, Alloc2>&&,
                   Unused> g) noexcept;

  template<ranges::input_range R, class Alloc>
    requires convertible_to<ranges::range_reference_t<R>, yielded>
  auto yield_value(ranges::elements_of<R, Alloc> r) noexcept;

  void await_transform() = delete;

  void return_void() const noexcept;
  void unhandled_exception();
  // (cont.)
```

```
  // (cont.)
  void* operator new(size_t size)
      requires same_as<Allocator, void> ||
               default_initializable<Allocator>;

  template<class Alloc, class... Args>
    requires same_as<Allocator, void> ||
      convertible_to<const Alloc&, Allocator>
  void* operator new(size_t size, allocator_arg_t,
                     const Alloc& alloc, const Args&...);

  template<class This, class Alloc, class... Args>
    requires same_as<Allocator, void> ||
      convertible_to<const Alloc&, Allocator>
  void* operator new(size_t size, const This&,
                     allocator_arg_t, const Alloc& alloc,
                     const Args&...);

  void operator delete(void* pointer, size_t size) noexcept;
};
```

std::generator::promise_type

```
// [coro.generator.promise] Class generator::promise_type
template<class Ref, class V, class Allocator>
class std::generator<Ref, V, Allocator>::promise_type {
public:
  generator get_return_object() noexcept;

  suspend_always in
  auto final_suspe

  suspend_always yi

  auto yield_value(
    requires is_rva
      constructible


  template<class R2
    requires same_a
  auto yield_value(


  template<ranges::
    requires conver
  auto yield_value(

  void await_transf

  void return_void() const noexcept;
  void unhandled_exception();
  // (cont.)
```

```
// (cont.)
void* operator new(size_t size)
    requires same_as<Allocator, v
             default_initializabl

ator>
tor_arg_t,
 const Args&...);

s... Args>

ator>
This&,
nst Alloc& alloc,

ze_t size) noexcept;
```

C++標準ライブラリ仕様から
ジェネレータを読み解くのは
HARD MODE

サンプルコードによる
C++23ジェネレータの紹介

C++23 ジェネレータ

コルーチン言語仕様

co_yield キーワード, co_return キーワード

<generator> ヘッダ

std::generator<Ref> クラステンプレート

<ranges> ヘッダ

std::ranges::elements_of クラステンプレート(タグ型)

ジェネレータコルーチン

戻り値型 **std::generator** かつ **co_yield式** を含む関数 ※
co_yield式がコルーチンの 中断(suspend)・再開(resume) ポイント

```
int main() {
  std::generator<int> g = f();
  std::println("start");
  for (int n: g) {
    std::println("recv {}", n);
  }
}
```

```
std::generator<int> f() {
  std::println("yield 1");
  co_yield 1;
  std::println("yield 2");
  co_yield 2;
}
```

```
start
yield 1
recv 1
yield 2
recv 2
```

※ 厳密には co_yield式 または co_return文 を一つ以上含む関数

ジェネレータコルーチン

コルーチン本体の開始前で
中断(suspend)される

```
int main() {
  std::generator<int> g = f();
  std::println("start");
  for (int n: g) {
    std::println("recv {}", n);
  }
}
```

```
std::generator<int> f() {
  std::println("yield 1");
  co_yield 1;
  std::println("yield 2");
  co_yield 2;
}
```

```
start
yield 1
recv 1
yield 2
recv 2
```

ジェネレータコルーチン

```
int main() {
  std::generator<int> g = f();
  std::println("start");
  for (int n: g) {
    std::println("recv {}", n);
  }
}
```

```
std::generator<int> f() {
  std::println("yield 1");
  co_yield 1;
  std::println("yield 2");
  co_yield 2;
}
```

```
start
yield 1
recv 1
yield 2
recv 2
```

ジェネレータコルーチン

範囲for文のイテレータ操作により
コルーチンが**再開(resume)**される

```
int main() {
  std::generator<int> g = f();
  std::println("start");
  for (int n: g) {
    std::println("recv {}", n);
  }
}
```

```
std::generator<int> f() {
  std::println("yield 1");
  co_yield 1;
  std::println("yield 2");
  co_yield 2;
}
```

```
start
yield 1
recv 1
yield 2
recv 2
```

ジェネレータコルーチン

co_yield式でコルーチン**中断(suspend)**し
std::generatorのイテレータを経由して
生成値を呼び出し元へと渡す

```
int main() {
  std::generator<int> g = f();
  std::println("start");
  for (int n: g) {
    std::println("recv {}", n);
  }
}
```

```
std::generator<int> f() {
  std::println("yield 1");
  co_yield 1;
  std::println("yield 2");
  co_yield 2;
}
```

```
start
yield 1
recv 1
yield 2
recv 2
```

ジェネレータコルーチン

```
int main() {
  std::generator<int> g = f();
  std::println("start");
  for (int n: g) {
    std::println("recv {}", n);
  }
}
```

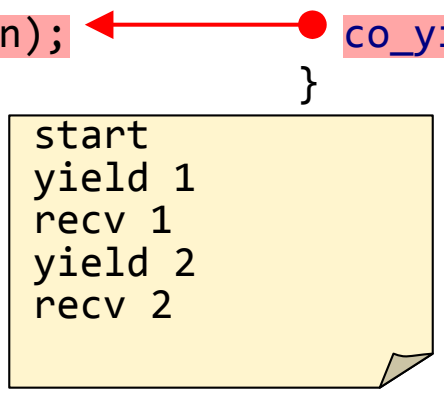
```
std::generator<int> f() {
  std::println("yield 1");
  co_yield 1;
  std::println("yield 2");
  co_yield 2;
}
```

```
start
yield 1
recv 1
yield 2
recv 2
```

ジェネレータコルーチン

```cpp
int main() {
  std::generator<int> g = f();
  std::println("start");
  for (int n: g) {
    std::println("recv {}", n);
  }
}
```

```cpp
std::generator<int> f() {
  std::println("yield 1");
  co_yield 1;
  std::println("yield 2");
  co_yield 2;
}
```



ジェネレータコルーチン

コルーチン本体終端に到達すると
イテレータは番兵(sentinel)と一致し
範囲for文から抜ける

```
int main() {
  std::generator<int> g = f();
  std::println("start");
  for (int n: g) {
    std::println("recv {}", n);
  }
}
```

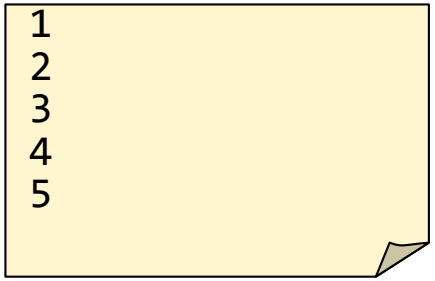
```
std::generator<int> f() {
  std::println("yield 1");
  co_yield 1;
  std::println("yield 2");
  co_yield 2;
}
```

```
start
yield 1
recv 1
yield 2
recv 2
```

遅延評価

std::generator は**遅延評価**される軽量なRange ※

```
int main() {
  for (int n: f() | std::views::take(5)) {
    std::println("{}", n);
  }
}
```




```
std::generator<int> f() {
  int n = 1;
  while (true) {
    co_yield n++;
  }
}
```

※ std::ranges::input_rangeコンセプトおよびstd::ranges::viewコンセプトのモデル

C++例外伝播

ジェネレータコルーチンの送出例外は呼出元に伝播（通常関数と同じ）

```
int main() {
  try {
    for (int n: f()) {
      std::println("{}", n);
    }
  }
  catch (int e) {
    std::println("catch {}", e);
  }
}
```

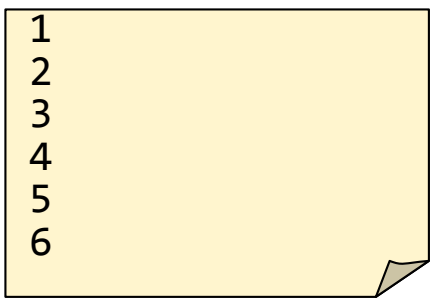
```
std::generator<int> f() {
  co_yield 1;
  co_yield 2;
  throw 42;
}
```

```
1
2
catch 42
```

std::ranges::elements_of

co_yield std::ranges::elements_of(*rng*) でRange要素を逐次生成

```
int main() {
  for (int n: f()) {
    std::println("{}", n);
  }
}
```




```
std::generator<int> f() {
  int arr[] = {1, 2};
  co_yield std::ranges::elements_of(arr);
  std::vector vec{3, 4};
  co_yield std::ranges::elements_of(vec);
  std::list lst{5, 6};
  co_yield std::ranges::elements_of(lst);
}
```

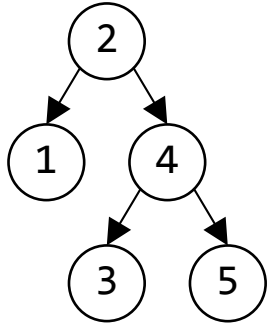
ジェネレータのネスト

```
struct node {
  int value;
  std::unique_ptr<node> left = nullptr;
  std::unique_ptr<node> right = nullptr;
};
node tree = {
  2,
  std::make_unique<node>(1),
  std::make_unique<node>(
    4,
    std::make_unique<node>(3),
    std::make_unique<node>(5)
  )};

int main() {
  for (int n: f(tree)) {
    std::println("{}", n);
  }
}
```

```
std::generator<int> f(const node& e) {
  if (e.left) {
    co_yield std::ranges::elements_of( f(*e.left) );
  }
  co_yield e.value;
  if (e.right) {
    co_yield std::ranges::elements_of( f(*e.right) );
  }
}
```

```
1
2
3
4
5
```



ジェネレータのネスト

```
struct node {
  int value;
  std::unique_ptr<node> left = nullptr;
  std::unique_ptr<node> right = nullptr;
};
node tree = {
  2,
  std::make_unique<node>(1),
  std::make_unique<node>(
    4,
    std::make_unique<node>(3),
    std::make_unique<node>(5)
  )};

int main() {
  for (int n: f(tree)) {
    std::println("{}", n);
  }
}
```

```
std::generator<int> f(const node& e) {
  if (e.left) {
    co_yield std::ranges::elements_of( f(*e.left) );
  }
  co_yield e.value;
  if (e.right) {
    co_yield std::ranges::elements_of( f(*e.right) );
  }
}
```

親/子コルーチン間では
Symmetric Transfer が行われ
コールスタックが深くならない
効率的な動作が仕様保証される

MoveOnly型

コピー不可 & ムーブ操作のみサポートする型も生成可能

```
using UPtr = std::unique_ptr<int>;

int main() {
  for (UPtr up: f()) {
    std::println("{}", *up);
  }
}
```


```
std::generator<UPtr> f() {
  for (int n = 1; n <= 5; n++) {
    UPtr up = std::make_unique<int>(n);
    co_yield std::move(up);
  }
}
```

Copy/Move不可型

コピー・ムーブともに不可能な型も取り扱える

```
struct X {
  int value;
  X(int n) : value(n) {}
  X(X&&) = delete;
  void operator=(X&&) = delete;
};


int main() {
  for (X&& ref: f()) {
    std::println("{}", ref.value);
  }
}
```

```
std::generator<X> f() {
  for (int n = 1; n <= 5; n++) {
    co_yield X{n};
  }
}
```

```
1
2
3
4
5
```

コルーチンスタック上の
一時オブジェクト X は
移動しない(できない)

std::generator<Ref>

```
std::generator<Ref, V=void, Alloc=void>
```

98%のケースはRefのみでOK
by 提案文書(P2502R2)

referenceメンバ型

std::generatorイテレータの間接参照(operator*)結果型

V=voidなら Ref&&※ / それ以外なら Ref

valueメンバ型

std::generatorイテレータの値型(value_type)

V=voidなら remove_cvref_t<Ref> / それ以外なら V

std::generator 自身は
valueメンバ型を利用しない

※ 型導出ではReference Collapsing規則が適用される

std::generator<Ref, V>

```
int main() {
  auto g = fizzbuzz()
    | std::views::take(15);

  for (auto&& s: g) {
    std::print("{} ", s);
  }
}
```

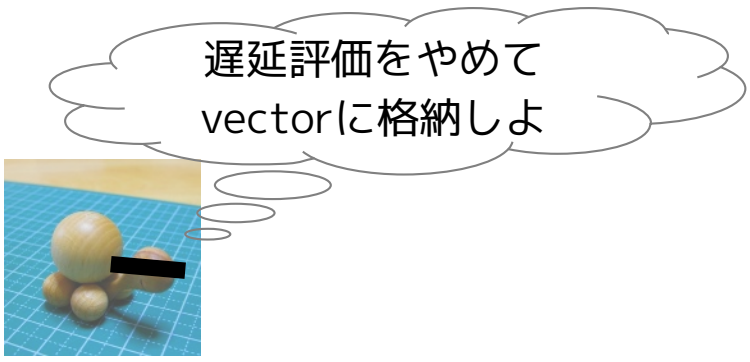
```
1 2 Fizz 4 Buzz Fizz
7 8 Fizz Buzz 11
Fizz 13 14 FizzBuzz
```

```
auto fizzbuzz() ->
  std::generator<std::string_view>
{
  for (size_t i = 1; ; ++i) {
    if (i % 15 == 0) {
      co_yield "FizzBuzz";
    } else if (i % 3 == 0) {
      co_yield "Fizz";
    } else if (i % 5 == 0) {
      co_yield "Buzz";
    } else {
      co_yield std::to_string(i);
    }
  }
}
```

std::generator<Ref, V>

```
int main() {
  auto vec = fizzbuzz()
    | std::views::take(15)
    | std::ranges::to<std::vector>();

  for (auto&& s: vec) {
    std::print("{} ", s);
  }
}
```




```
auto fizzbuzz() ->
  std::generator<std::string_view>
{
  for (size_t i = 1; ; ++i) {
    if (i % 15 == 0) {
      co_yield "FizzBuzz";
    } else if (i % 3 == 0) {
      co_yield "Fizz";
    } else if (i % 5 == 0) {
      co_yield "Buzz";
    } else {
      co_yield std::to_string(i);
    }
  }
}
```

std::generator<Ref, V>

vector<string_view>に推論

```
int main() {
  auto vec = fizzbuzz()
    | std::views::take(15)
    | std::ranges::to<std::vector>();

  for (auto&& s: vec) {
    std::print("{} ", s);
  }
}
```

```
1 1 Fizz 1 Buzz Fizz
1 1 Fizz Buzz 16
Fizz 16 16 FizzBuzz
```

未定義動作

```
auto fizzbuzz() ->
  std::generator<std::string_view>
{
  for (size_t i = 1; ; ++i) {
    if (i % 15 == 0) {
      co_yield "FizzBuzz";
    } else if (i % 3 == 0) {
      co_yield "Fizz";
    } else if (i % 5 == 0) {
      co_yield "Buzz";
    } else {
      co_yield std::to_string(i);
    }
  }
}
```

生成される std::string は
コルーチン中断期間だけ有効

std::generator<Ref, V>

vector<string>に推論

```
int main() {
  auto vec = fizzbuzz()
    | std::views::take(15)
    | std::ranges::to<std::vector>();

  for (auto&& s: vec) {
    std::print("{} ", s);
  }
}
```

```
1 2 Fizz 4 Buzz Fizz
7 8 Fizz Buzz 11
Fizz 13 14 FizzBuzz
```

```
auto fizzbuzz() ->
  std::generator<std::string_view, std::string>
{
  for (size_t i = 1; ; ++i) {
    if (i % 15 == 0) {
      co_yield "FizzBuzz";
    } else if (i % 3 == 0) {
      co_yield "Fizz";
    } else if (i % 5 == 0) {
      co_yield "Buzz";
    } else {
      co_yield std::to_string(i);
    }
  }
}
```

reference型 = std::string_view
value型 = std::string

Darkside

コルーチン固有の分かりづらい落とし穴が存在する
（C++ Core Guidelines, CP.coroより関連ガイドライン引用）

CP.53: Parameters to coroutines should not be passed by reference
　　戻り値型 std::generator × 関数の**参照型引数**

CP.51: Do not use capturing lambdas that are coroutines
　　戻り値型 std::generator × ラムダ式の**変数キャプチャ**

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#cpcoro-coroutines

ジェネレータコルーチンと参照型引数

参照型引数をとるジェネレータコルーチンは危険

```
int main() {
  auto g = f({1, 2, 3, 4});
  for (int n: g) {
    std::println("{}", n);
  }
}
```

```
auto f(const std::vector<int>& vec)
  -> std::generator<int>
{
  for (int m: vec) {
    co_yield m * m;
  }
}
```

ジェネレータコルーチンと参照型引数

1. std::vectorオブジェクト
生存期間はここまで

```
int main() {
  auto g = f({1, 2, 3, 4});
  for (int n: g) {
    std::println("{}", n);
  }
}
```

```
1607410369
0
-1292001839
-1347369775
```

未定義動作

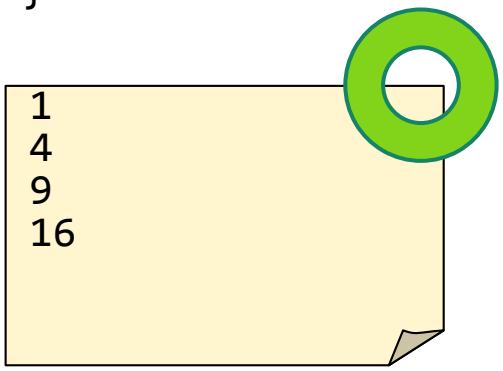
```
auto f(const std::vector<int>& vec)
  -> std::generator<int>
{
  for (int m: vec) {
    co_yield m * m;
  }
}
```

2. コルーチン本体の再開時には
ダングリング参照となっている

ジェネレータコルーチンと参照型引数

ジェネレータコルーチンの引数型は「値型」とする

```
int main() {
  auto g = f({1, 2, 3, 4});
  for (int n: g) {
    std::println("{}", n);
  }
}
```




```
auto f(std::vector<int> vec)
  -> std::generator<int>
{
  for (int m: vec) {
    co_yield m * m;
  }
}
```

コルーチンフレーム上に
Vector{1,2,3,4} を構築・保持

ジェネレータコルーチン(ラムダ式)と変数キャプチャ

変数キャプチャを行うジェネレータコルーチン・ラムダ式は危険

```
int main() {
  for (int n: f(4)) {
    std::println("{}", n);
  }
}
```

```
std::generator<int> f(size_t N) {
  std::vector<int> vec(N);
  std::ranges::iota(vec, 1);
  // vec={1,2,...N}
  auto lm = [vec]()
       -> std::generator<int> {
    for (int m: vec) {
      co_yield m * m;
    }
  };
  return lm();
}
```

https://yohhoy.hatenadiary.jp/entry/20211111/p1

ジェネレータコルーチン(ラムダ式)と変数キャプチャ

```cpp
int main() {
  for (int n: f(4)) {
    std::println("{}", n);
  }
}
```

```
814990336
1073676289
2076770304
0
814990336
...
```

未定義動作

```cpp
std::generator<int> f(size_t N) {
  std::vector<int> vec(N);
  std::ranges::iota(vec, 1);
  // vec={1,2,...N}
  auto lm = [vec]()
       -> std::generator<int> {
    for (int m: vec) {
      co_yield m * m;
    }
  };
  return lm();
}
```

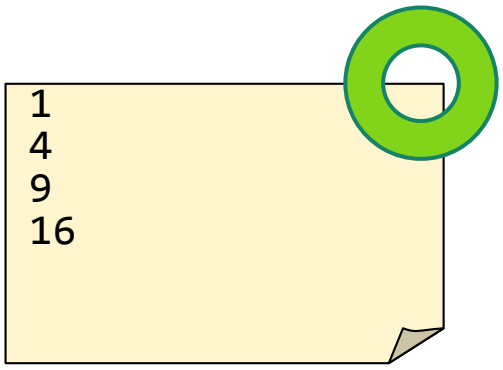
2. コルーチン本体の再開時には
キャプチャ変数の実体は破棄済み

1. キャプチャ変数 vec を包含する
クロージャオブジェクトの寿命ここまで

ジェネレータコルーチン(ラムダ式)と変数キャプチャ

ジェネレータコルーチン・ラムダ式へは「値型の引数」経由で渡す

```
int main() {
  for (int n: f(4)) {
    std::println("{}", n);
  }
}
```




```
std::generator<int> f(size_t N) {
  std::vector<int> vec(N);
  std::ranges::iota(vec, 1);
  // vec={1,2,...N}
  auto lm = [](auto vec)
       -> std::generator<int> {
    for (int m: vec) {
      co_yield m * m;
    }
  };
  return lm(vec);
}
```

https://yohhoy.hatenadiary.jp/entry/20211111/p1

まとめ

C++23 ジェネレータの特徴

戻り値型 **std::generator<Ref>** と **co_yield文** を利用

std::generator は遅延評価される軽量Range(View)

std::ranges::elements_of による効率的なネスト動作

MoveOnly型やコピー/ムーブ不可型も扱える

参照型引数やラムダ式キャプチャとの組合せは危険

Visit “cpprefjp” website !

https://cpprefjp.github.io/reference/generator.html

[参考] std::generator<Ref>とco_yield

第1テンプレート引数 Ref にて値生成時のコピー／ムーブ動作を制御する

Ref型	co_yield式オペランド		
	左辺値	const左辺値	右辺値
T T&&	コピー1回	コピー1回	0回
T&	コピー1回	コピー1回	不適格
const T&	0回	0回	0回

TとT&&で
差異なし

[参考] コンパイル時ジェネレータ

コンパイル時のコルーチンサポートに関する提案文書
P3367R3 constexpr coroutines
　　「AST変換方式を用いてClangへ実験的に実装してみた」
P3590R0 Constexpr Coroutine Burdens
　　「時期尚早／コンパイラ基盤の進化(VMベース移行)を待つべき」

C++標準化委員会@2025-02
　　「C++26には含めない／C++29以降を再ターゲット」

[参考] 関連リンク

P2502R2 std::generator: Synchronous Coroutine Generator for Ranges
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2502r2.pdf
P2529R0 generator should have T&& reference_type
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2529r0.html

コルーチン×ラムダ式キャプチャ＝鼻から悪魔
https://yohhoy.hatenadiary.jp/entry/20211111/p1
std::generator<R>
https://yohhoy.hatenadiary.jp/entry/20220801/p1
std::generator<T／T&&／T&／const T&>
https://yohhoy.hatenadiary.jp/entry/20220810/p1