Учимся готовить C++ корутины на практике
часть 2: генераторы

Understanding C++ coroutines by example
part 2: generators

Pavel Novikov

@cpp_ape

Goals of this talk

Develop intuition about how generators work:

- coroutine generators in general
- range generators
  + how recursive generators work in principle
- async generators

Disclaimer

Code on the slides is intended for educational purposes,

it is somewhat suboptimal and should not be used in production as it is.

What is a C++ coroutine?

```
Generator<int> foo() {
  co_yield 42;
}
```

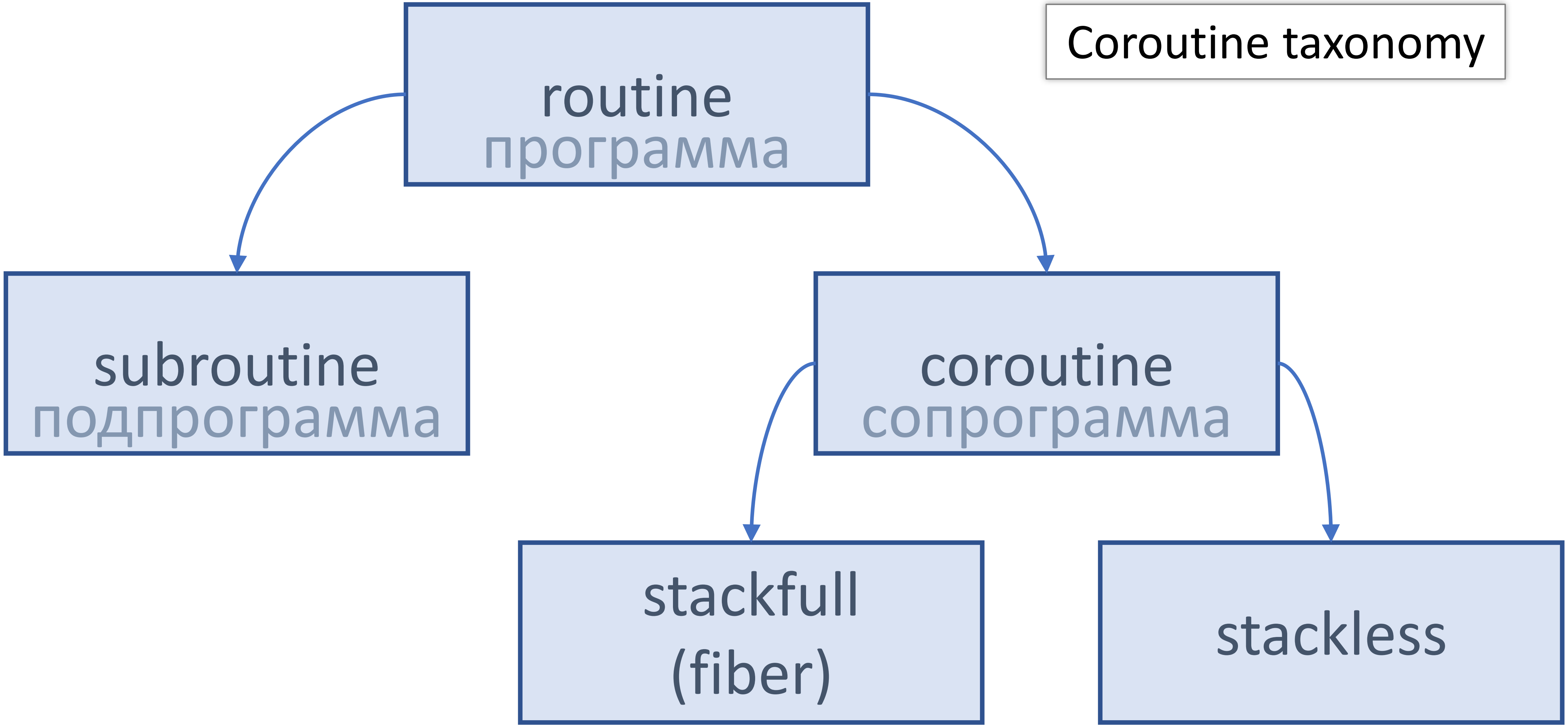
A function is a coroutine if it contains one of these:

`co_return` (coroutine return statement)

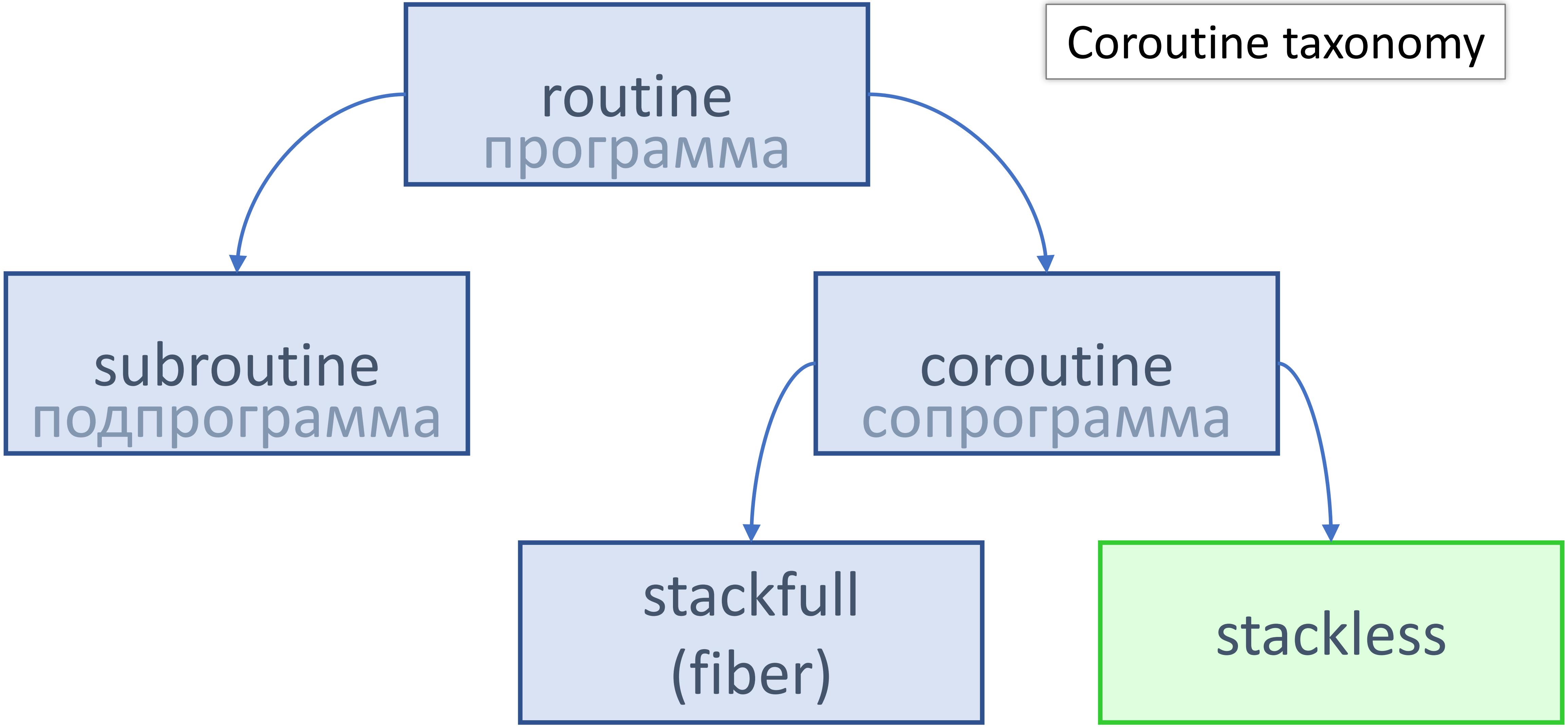
`co_await` (await expression)

`co_yield` (yield expression)

What is a C++ coroutine?



What is a C++ coroutine?



What is a C++ coroutine?

Simula

From Wikipedia, the free encyclopedia

This article is about the programming language. For the village in Estonia, see Simula, Estonia.

Not to be confused with Simulia.

Simula is the name of two simulation programming languages, Simula I and Simula 67, developed in the 1960s at the Norwegian Computing Center in Oslo, by Ole-Johan Dahl and Kristen Nygaard. Syntactically, it is an approximate superset of ALGOL 60,[1]: 1.3.1 and was also influenced by the design of Simscript.[2]

Simula 67 introduced objects,[1]: 2, 5.3 classes,[1]: 1.3.3, 2 inheritance and subclasses,[1]: 2.2.1 virtual procedures,[1]: 2.2.3 coroutines,[1]: 9.2 and discrete event simulation,[1]: 14.2 and featured garbage collection.[1]: 9.1 Other forms of subtyping (besides inheriting subclasses) were introduced in Simula derivatives.[*citation needed*]

Simula is considered the first object-oriented programming language. As its name suggests, the first Simula version by 1962 was designed for doing simulations; Simula 67 though was designed to be a general-purpose programming language[3] and provided the framework for many of the features of object-oriented languages today.

Simula has been used in a wide range of applications such as simulating

Simula

Paradigms	Multi-paradigm: procedural, imperative, structured, object-oriented
Family	ALGOL
Designed by	Ole-Johan Dahl
Developer	Kristen Nygaard
First appeared	1962; 60 years ago
Stable release	Simula 67, Simula I
Typing discipline	Static, nominative
Scope	Lexical
Implementation language	ALGOL 60 (primarily; some components Simscript)

What is a C++ coroutine?

Simula

From Wikipedia, the free encyclopedia

This article is about the programming language. For the village in Estonia, see Simula, Estonia.

Not to be confused with Simulia.

Simula is the name of two simulation programming languages, Simula I and Simula 67, developed in the 1960s at the Norwegian Computing Center in Oslo, by Ole-Johan Dahl and Kristen Nygaard. Syntactically, it is an

Simula

Simula 67 introduced objects,[1]:2,5.3 classes,[1]:1.3.3,2 inheritance and subclasses,[1]:2.2.1 virtual procedures,[1]:2.2.3 coroutines,[1]:9.2 and discrete event simulation,[1]:14.2 and featured garbage collection.[1]:9.1 Other forms of subtyping (besides inheriting subclasses) were introduced in Simula derivatives.[*citation needed*]

simulations; Simula 67 though was designed to be a general-purpose programming language[3] and provided the framework for many of the features of object-oriented languages today.

Simula has been used in a wide range of applications such as simulating

Typing discipline	Static, nominative
Scope	Lexical
Implementation language	ALGOL 60 (primarily; some components Simscript)

What is a C++ coroutine?

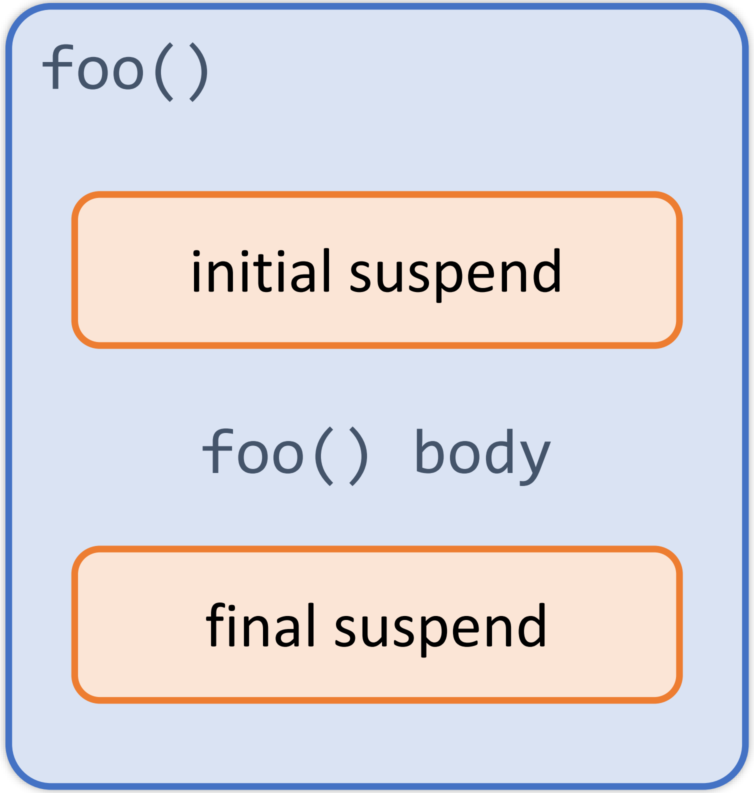
```
Generator<int> foo() {
  co_yield 42;
}
```

A coroutine behaves as if its *function-body* were replaced by:

```
{
  promise-type promise promise-constructor-arguments ;
  try {
    co_await promise.initial_suspend() ;
    function-body
  } catch ( ... ) {
    if (!initial-await-resume-called)
      throw ;
    promise.unhandled_exception() ;
  }
final-suspend :
  co_await promise.final_suspend() ;
}
```

What is a C++ coroutine?

```
Generator<int> foo() {
  co_yield 42;
}
```

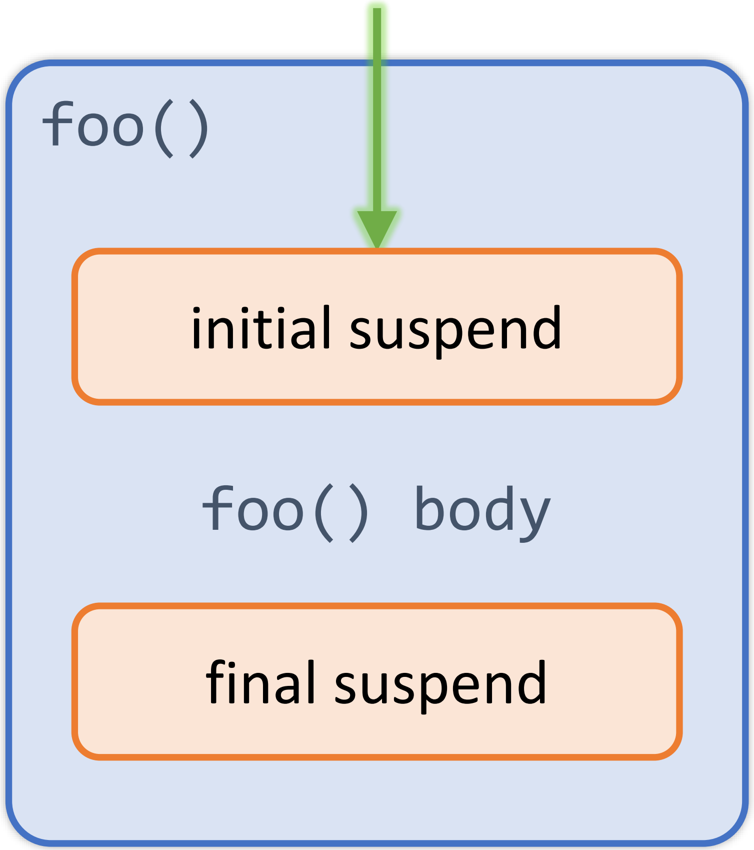


A coroutine behaves as if its *function-body* were replaced by:

```
{
  promise-type promise promise-constructor-arguments ;
  try {
    co_await promise.initial_suspend() ;
    function-body
  } catch ( ... ) {
    if (!initial-await-resume-called)
      throw ;
    promise.unhandled_exception() ;
  }
final-suspend :
  co_await promise.final_suspend() ;
}
```

What is a C++ coroutine?

```
Generator<int> foo() {
  co_yield 42;
}
```

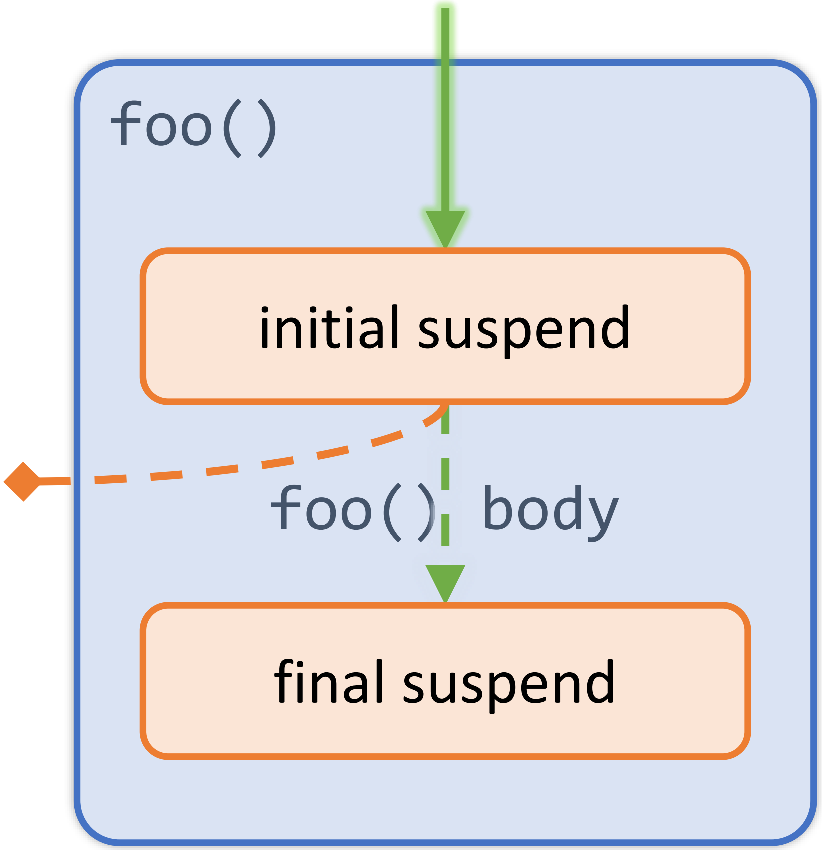


A coroutine behaves as if its *function-body* were replaced by:

```
{
  promise-type promise promise-constructor-arguments ;
  try {
    co_await promise.initial_suspend() ;
    function-body
  } catch ( ... ) {
    if (!initial-await-resume-called)
      throw ;
    promise.unhandled_exception() ;
  }
final-suspend :
  co_await promise.final_suspend() ;
}
```

What is a C++ coroutine?

```
Generator<int> foo() {
  co_yield 42;
}
```

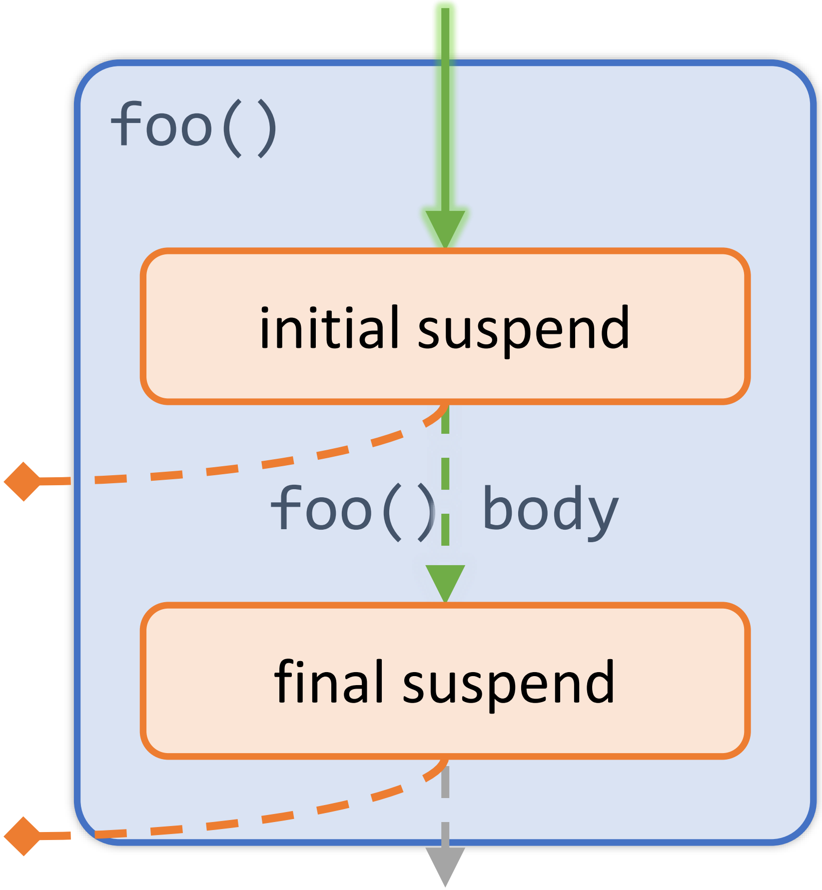


A coroutine behaves as if its *function-body* were replaced by:

```
{
   promise-type promise promise-constructor-arguments ;
   try {
      co_await promise.initial_suspend() ;
      function-body
   } catch ( ... ) {
      if (!initial-await-resume-called)
         throw ;
      promise.unhandled_exception() ;
   }
final-suspend :
   co_await promise.final_suspend() ;
}
```

What is a C++ coroutine?

```
Generator<int> foo() {
  co_yield 42;
}
```



A coroutine behaves as if its *function-body* were replaced by:

```
{
   promise-type promise promise-constructor-arguments ;
   try {
      co_await promise.initial_suspend() ;
      function-body
   } catch ( ... ) {
      if (!initial-await-resume-called)
         throw ;
      promise.unhandled_exception() ;
   }
final-suspend :
   co_await promise.final_suspend() ;
}
```

What is a C++ coroutine?

```
Generator<int> foo() {
  co_yield 42;
}
```

A coroutine behaves as if its *function-body* were replaced by:

```
{
  promise-type promise promise-constructor-arguments ;
  try {
    co_await promise.initial_suspend() ;
    function-body
  } catch ( ... ) {
    if (!initial-await-resume-called)
      throw ;
    promise.unhandled_exception() ;
  }
final-suspend :
  co_await promise.final_suspend() ;
}
```

What is a C++ coroutine?

```
co_yield expression
```

```
co_await promise.yield_value(expression)
```

What is a C++ coroutine?

```
co_return expression_opt;
```

should be of type `void` in our case

```
{
  expression_opt;
  promise.return_void();
  goto final-suspend;
}
```

What is a C++ coroutine?

```
co_return expression_opt;
```

should be of type `void` in our case

```
{
  expression_opt;
  promise.return_void();
  goto final-suspend;
}
```

```
Generator<int> foo() {
  co_yield 42;
  co_return;
}
```

What is a C++ coroutine?

```
co_return expression_opt;
```

should be of type `void` in our case

```
{
  expression_opt;
  promise.return_void();
  goto final-suspend;
}
```

```
Generator<int> foo() {
  co_yield 42;
  // implicit co_return;
}
```

Best practices so far

- "Lazy" asynchronous tasks which do not start immediately, they are suspended at initial suspend point
  (contrast to "eager" tasks)
- Result from asynchronous tasks can be obtained either
  - by `co_await`ing within a coroutine (possibly suspending it), or
  - by synchronously waiting (possibly blocking the thread)

  (unlike `std::future` and co.)

Watch **Lewis Baker**'s talk

"Structured Concurrency: Writing safer concurrent code with coroutines and algorithms"

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}


const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

```
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

```
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

“Naïve” generator

```cpp
Generator<std::string> foo() {
   co_yield "hello";

   const auto s = std::string{ "world" };
   co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```cpp
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
   co_yield "hello";

   const auto s = std::string{ "world" };
   co_yield s;
}

const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {   suspends
    co_yield "hello";

    const auto s = std::string{ "world" };
    co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

“Naïve” generator

```cpp
Generator<std::string> foo() {
   co_yield "hello";

   const auto s = std::string{ "world" };
   co_yield s;
}
```

```cpp
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
    co_yield "hello";

    const auto s = std::string{ "world" };
    co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

resumes

```
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}


const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

coroutine frame
contains

> **promise**
> tells when and how to suspend and resume

and internal state stuff

"Naïve" generator

```
Generator<std::string> foo() {
  co_yield "hello";
                                        suspends

  const auto s = std::string{ "world" };
  co_yield s;
}


const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

coroutine frame
contains

> **promise**
> tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
   co_yield "hello";
   
   const auto s = std::string{ "world" };
   co_yield s;
}
```

resumes

```
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}

const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
    co_yield "hello";

    const auto s = std::string{ "world" };
    co_yield s;
}
```

suspends

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
   co_yield "hello";

   const auto s = std::string{ "world" };
   co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

lives until `f` is destroyed

```
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
   co_yield "hello";

   const auto s = std::string{ "world" };
   co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' ' << f() << '\n';
```

outputs
`hello world`

“Naïve” generator

Why *initial suspend*?

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}

{
  const auto g = bar();  // created but not used
}
```

“Naïve” generator

Why *initial suspend*?

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}

{
  const auto g = bar();  // created but not used
}
```

“Naïve” generator

Why *initial suspend*?

```
Generator<int> bar() {                                    initial suspend
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}

{
  const auto g = bar();  // created but not used
}
```

“Naïve” generator

Why *initial suspend*?

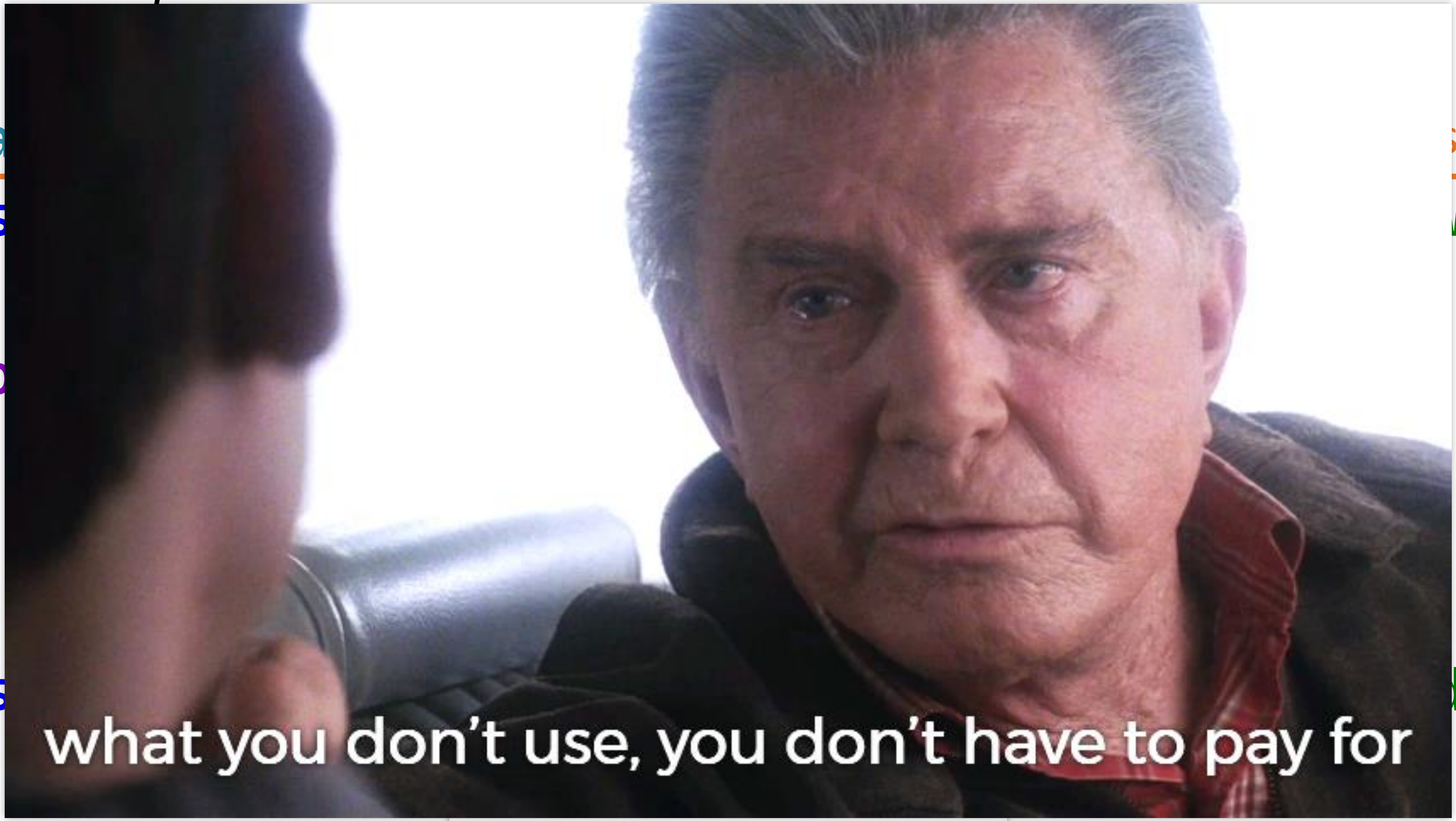
```
Generator<int> bar() {
                                                    initial suspend
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}

{
  const auto g = bar();  // created but not used
}
```

zero overhead principle

“Naïve” generator

Why *initial suspend*?

```
Genera                                   suspend
   cons
   for
      co
}

{
   cons
}
```



zero overhead principle

"Naïve" generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

```
{
  const auto f = foo();
  std::cout << f() << '\n';
} // f is safely destroyed
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

```
{
  const auto f = foo();
  std::cout << f() << '\n';
} // f is safely destroyed
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

```
{
  const auto f = foo();
  std::cout << f() << '\n';
} // f is safely destroyed
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

```
{
  const auto f = foo();
  std::cout << f() << '\n';
} f.~Generator<std::string>()
```

safely* destroys coroutine frame
in suspended state
and frees all associated resources

“Naïve” generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  auto &operator()() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

“Naïve” generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  auto &operator()() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

“Naïve” generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  auto &operator()() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

“Naïve” generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  auto &operator()() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

only coroutine handle is stored

“Naïve” generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  auto &operator()() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

“Naïve” generator

mandatory

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}

  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

“Naïve” generator

mandatory

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}

  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

“Naïve” generator

mandatory

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}

  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

“Naïve” generator

mandatory

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}

  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

“Naïve” generator

mandatory

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value) noexcept(std::is_n

  void return_void() const noexcept {}

  void unhandled_exception() noexcept(std::is_nothrow_copy_constructi

  T &getValue();

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
auto get_return_object() noexcept {
  return Generator{ *this };
}
```

“Naïve” generator

```
auto get_return_object() noexcept {
  return Generator{ *this };
}
```

```
Generator<std::string> foo();


const auto f = foo();
```

“Naïve” generator

```
auto get_return_object() noexcept {
  return Generator{ *this };
}
```

```
Generator<std::string> foo();


const auto f = foo();
```

“Naïve” generator

```
auto get_return_object() noexcept {
  return Generator{ *this };
}
```

```
Generator<std::string> foo();

const auto f = foo();
```

“Naïve” generator

```
std::suspend_always initial_suspend() const noexcept {
  return {};
}
std::suspend_always final_suspend() const noexcept {
  return {};
}
```

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

initial suspend

final suspend

“Naïve” generator

```
std::suspend_always yield_value(const T &value)
  noexcept(std::is_nothrow_copy_constructible_v<T>) {
  result = value;
  return {};
}
```

“Naïve” generator

```
std::suspend_always yield_value(const T &value)
  noexcept(std::is_nothrow_copy_constructible_v<T>) {
  result = value;
  return {};
}
```

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

“Naïve” generator

```
std::suspend_always yield_value(const T &value)
  noexcept(std::is_nothrow_copy_constructible_v<T>) {
  result = value;
  return {};
}
```

```
co_yield expression
```

⇓

```
co_await promise.yield_value(expression)
```

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

“Naïve” generator

```
std::suspend_always yield_value(const T &value)
  noexcept(std::is_nothrow_copy_constructible_v<T>) {
  result = value;
  return {};
}
```

```
Generator<std::string> foo() {
  co_yield "hello";
```

suspends

```
  const auto s = std::string{ "world" };
  co_yield s;
}
```

“Naïve” generator

```
void return_void() const noexcept {}
```

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
  co_return;
}
```

“Naïve” generator

```
void return_void() const noexcept {}
```

```
co_return expression_opt;
```

$\Downarrow$

```
{ expression_opt; promise.return_void(); goto final-suspend; }
```

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
  co_return;
}
```

“Naïve” generator

```
void return_void() const noexcept {}
```

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
  co_return;
}
```

“Naïve” generator

```
void return_void() const noexcept {}
```

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;

}
```

“Naïve” generator

```
void unhandled_exception()
  noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>) {
  result = std::current_exception();
}
```

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

“Naïve” generator

```
struct promise_type {
  //...

  T &getValue() {
    if (std::holds_alternative<std::exception_ptr>(result))
      std::rethrow_exception(std::get<std::exception_ptr>(result));
    return std::get<T>(result);
  }

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

“Naïve” generator

```
struct promise_type {
  //...

  T &getValue() {
    if (std::holds_alternative<std::exception_ptr>(result))
      std::rethrow_exception(std::get<std::exception_ptr>(result));
    return std::get<T>(result);
  }

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

precondition:
we must have result or exception

“Naïve” generator

mandatory

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}

  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

“Naïve” generator

mandatory

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value) noexcept(std::is_n

  void return_void() const noexcept {}

  void unhandled_exception() noexcept(std::is_nothrow_copy_constructi

  T &getValue();

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  auto &operator()() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

“Naïve” generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  auto &operator()() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

only coroutine handle is stored

“Naïve” generator

```
Generator(Generator &&other) noexcept :
  coro{ std::exchange(other.coro, nullptr) }
{}

Generator &operator=(Generator &&other) noexcept {
  if (coro)
    coro.destroy();
  coro = std::exchange(other.coro, nullptr);
}

~Generator() {
  if (coro)
    coro.destroy();
}
```

“Naïve” generator

```
Generator(Generator &&other) noexcept :
  coro{ std::exchange(other.coro, nullptr) }
{}

Generator &operator=(Generator &&other) noexcept {
  if (coro)
    coro.destroy();
  coro = std::exchange(other.coro, nullptr);
}

~Generator() {
  if (coro)
    coro.destroy();
}
```

“Naïve” generator

```
Generator(Generator &&other) noexcept :
  coro{ std::exchange(other.coro, nullptr) }
{}

Generator &operator=(Generator &&other) noexcept {
  if (coro)
    coro.destroy();
  coro = std::exchange(other.coro, nullptr);
}

~Generator() {
  if (coro)
    coro.destroy();
}
```

“Naïve” generator

```
auto &operator()() const {
  coro();  // same as 'coro.resume()'
  return coro.promise().getValue();
}
```

“Naïve” generator

```
auto &operator()() const {
  coro();  // same as 'coro.resume()'
  return coro.promise().getValue();
}
```

```
struct promise_type {
  //...

  T &getValue() {
    if (std::holds_alternative<std::exception_ptr>(result))
      std::rethrow_exception(std::get<std::exception_ptr>(result));
    return std::get<T>(result);
  }

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

“Naïve” generator

```
template<typename T>
struct Generator {
  //...

private:
  explicit Generator(promise_type &promise) noexcept :
    coro{ std::coroutine_handle<promise_type>::from_promise(promise) }
  {}

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

```
struct promise_type {
  auto get_return_object() noexcept {
    return Generator{ *this };
  }
  //...
};
```

“Naïve” generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  auto &operator()() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

only coroutine handle is stored

"Naïve" generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}


const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
   co_yield "hello";

   const auto s = std::string{ "world" };
   co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

```
auto get_return_object() noexcept {
  return Generator{ *this };
}
```

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
   co_yield "hello";

   const auto s = std::string{ "world" };
   co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

"Naïve" generator

```
Generator<std::string> foo() {   suspends
  co_yield "hello";
```

coroutine frame
contains

promise
tells when and how to
...me

```
std::suspend_always initial_suspend() const noexcept {
  return {};
}
```

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```cpp
Generator<std::string> foo() {
    co_yield "hello";

    const auto s = std::string{ "world" };
    co_yield s;
}
```

```cpp
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";
```

```
auto &Generator::operator()() const {
  coro();  // same as 'coro.resume()'
  return coro.promise().getValue();
}
```

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";
```

resumes

```
auto &Generator::operator()() const {
  coro();  // same as 'coro.resume()'
  return coro.promise().getValue();
}
```

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
    co_yield "hello";

    const auto s = std::string{ "world" };
    co_yield s;
}
```

resumes

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}


const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";
```

```
std::suspend_always yield_value(const T &value)
  noexcept(std::is_nothrow_copy_constructible_v<T>) {
  result = value;
  return {};
}
```

coroutine frame
contains

promise
…en and how to
…d and resume

…rnal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";
```

suspends

```
std::suspend_always yield_value(const T &value)
  noexcept(std::is_nothrow_copy_constructible_v<T>) {
  result = value;
  return {};
}
```

coroutine frame
contains

promise
en and how to
d and resume

rnal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";
```

```
auto &Generator::operator()() const {
  coro();  // same as 'coro.resume()'
  return coro.promise().getValue();
}
```

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";
```

```
auto &Generator::operator()() const {
  coro();  // same as 'coro.resume()'
  return coro.promise().getValue();
}
```

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";
```

```
auto &Generator::operator()() const {
  coro();   // same as 'coro.resume()'
  return coro.promise().getValue();
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
T &getValue() {
  if (std::holds_alternative<std::exception_ptr>(result))
    std::rethrow_exception(std::get<std::exception_ptr>(result));
  return std::get<T>(result);
}
```

“Naïve” generator

```
Generator<std::string> foo() {
   co_yield "hello";

   const auto s = std::string{ "world" };
   co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
    co_yield "hello";

    const auto s = std::string{ "world" };
    co_yield s;
}
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
   co_yield "hello";
                                    resumes
   const auto s = std::string{ "world" };
   co_yield s;
}
```

coroutine frame

contains

> **promise**
> tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
   co_yield "hello";

   const auto s = std::string{ "world" };
   co_yield s;
}
```

coroutine frame
contains

> **promise**
> tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
   co_yield "hello";

   const auto s = std::string{ "world" };
   co_yield s;
}
```

suspends

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

```
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

“Naïve” generator

```cpp
Generator<std::string> foo() {
    co_yield "hello";

    const auto s = std::string{ "world" };
    co_yield s;
}
```

```cpp
const auto f = foo();
std::cout << f() << ' '
          << f() << '\n';
```

coroutine frame
contains

promise
tells when and how to suspend and resume

and internal state stuff

“Naïve” generator

- can’t know if there are more values
- perfect for infinite sequences
  though

"Naïve" generator

- can't know if there are more values
- perfect for infinite sequences though

```
Generator<int>
```

```
() {
  int prev = 1, next = 1;
  for (;;) {
    co_yield next;
    std::swap(prev, next);
    next += prev;
  }
}

const auto f =
```

```
();
for (size_t i = 0; i != 5; ++i)
  std::cout << f() << '\n';
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}

const auto f = foo();
std::cout << f() << ' '     // yields "hello"
          << f() << '\n';   // yields "world"
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

```
const auto f = foo();
std::cout << f() << ' '    // yields "hello"
          << f() << '\n';  // yields "world"
```

“Naïve” generator

```
Generator<std::string> foo() {
  co_yield "hello";

  const auto s = std::string{ "world" };
  co_yield s;
}
```

```
const auto f = foo();
std::cout << f() << ' '     // yields "hello"
          << f() << '\n';  // yields "world"
std::cout << f();           // ???
```

“Naïve” generator

- unnecessary extra copy

```
Generator<std::string> foo() {
  co_yield "hello";
  //...
}
```

T = std::string

```
std::suspend_always yield_value(const T &value)
  noexcept(std::is_nothrow_copy_constructible_v<T>) {
  result = value;
  return {};
}
```

copies value into `result`

“Naïve” generator

```
const auto s = std::string{ "world" };
co_yield s;
```

```
co_await promise.yield_value(s);
```

“Naïve” generator

```
const auto s = std::string{ "world" };
co_yield s;
```

```
co_await promise.yield_value(s);
```

suspends

“Naïve” generator

```
const auto s = std::string{ "world" };
co_yield s;
```

```
co_await promise.yield_value(s);
```

variable is still accessible during suspension

“Naïve” generator

```
co_yield "hello";
```

```
co_await promise.yield_value("hello");
```

“Naïve” generator

```
co_yield "hello";
```

```
std::string t{ "hello" }
```

```
co_await promise.yield_value( t );
```

“Naïve” generator

```
co_yield "hello";
```

```
std::string t{ "hello" }
```

```
co_await promise.yield_value( t );
```

suspends

“Naïve” generator

```
co_yield "hello";
```

```
std::string t{ "hello" }
```

```
co_await promise.yield_value(   t   );
```

“Naïve” generator

```
co_yield "hello";
```

```
std::string t{ "hello" }
```

```
co_await promise.yield_value(    t    );
```

resumes

“Naïve” generator

```
co_yield "hello";
```

```
std::string t{ "hello" }
co_await promise.yield_value(  t  );
t.~std::string()
```

resumes

“Naïve” generator

```
co_yield "hello";
```

```
std::string t{ "hello" }
```

```
co_await promise.yield_value(   t   );
```

suspends

resumes

```
t.~std::string()
```

```
std::suspend_always yield_value(const T &value)
  noexcept(std::is_nothrow_copy_constructible_v<T>) {
  result = value;
  return {};
}
```

reference is valid during suspension
(until destruction of coroutine frame)

“Naïve” generator

At this point you know almost everything you need to know about how generators work.

The rest is just interface design and making design decisions.

Simple generator

```
const auto g = bar();
while (g.hasValue())
  std::cout << g() << '\n';
```

Simple generator

```cpp
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  bool hasValue() const noexcept;  //has value or exception
  auto &operator()() const;

private:
  explicit Generator(promise_type &promise) noexcept;

  void getNextValue() const noexcept;

  std::coroutine_handle<promise_type> coro;
  mutable bool gotValue = false;
};
```

Simple generator

```
const auto g = bar();
std::cout << g() << '\n'; // we _know_ it has a value
```

compared to

has to get the next value

```
if (g.hasValue())
  std::cout << g() << '\n';
```

Simple generator

```
const auto g = bar();
while (g.hasValue())
  std::cout << g() << '\n';
```

Simple generator

```
const auto g = bar();
while (g.hasValue())
  std::cout << g() << '\n';
```

UGLY

Simple generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  bool advance() const noexcept;
  auto &getValue() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

Simple generator

```
const auto g = bar();
while (g.advance())
  std::cout << g.getValue() << '\n';
```

Simple generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  bool advance() const noexcept;
  auto &getValue() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

Simple generator

```cpp
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}

  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  bool hasException() const noexcept;
  T &getValue();

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

Simple generator

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}

  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  bool hasException() const noexcept;
  T &getValue();

private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

Simple generator

```
struct promise_type {
  //...
  std::suspend_always yield_value(const T &value)
    noexcept(std::is_nothrow_copy_constructible_v<T>) {
    result = value;
    return {};
  }



  //...
private:
  std::variant<std::monostate, T, std::exception_ptr> result;
};
```

copies value into result

Simple generator

```
struct promise_type {
  //...
  std::suspend_always yield_value(const T &value)
    noexcept(std::is_nothrow_copy_constructible_v<T>) {
    result = value;
    return {};
  }
  std::suspend_always yield_value(T &&value) noexcept {
    result = std::addressof(value);
    return {};
  }
  //...
private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
};
```

copies value into result

Simple generator

```
struct promise_type {
  //...
  bool hasException() const noexcept {
    return std::holds_alternative<std::exception_ptr>(result);
  }
  T &getValue() {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));

    return std::holds_alternative<T>(result) ? std::get<T>(result) :
                                               *std::get<T*>(result);
  }
  //...
private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
};
```

Simple generator

```
struct promise_type {
  //...
  bool hasException() const noexcept {
    return std::holds_alternative<std::exception_ptr>(result);
  }
  T &getValue() {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));

    return std::holds_alternative<T>(result) ? std::get<T>(result) :
                                              *std::get<T*>(result);
  }
  //...
private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
};
```

Simple generator

```cpp
struct promise_type {
  //...
  bool hasException() const noexcept {
    return std::holds_alternative<std::exception_ptr>(result);
  }
  T &getValue() {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));

    return std::holds_alternative<T>(result) ? std::get<T>(result) :
                                               *std::get<T*>(result);
  }
  //...
private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
};
```

precondition:
we must have result or exception

Simple generator

```
const auto f = foo();
std::vector<std::string> values;
while (f.advance())
  values.push_back(std::move(f.getValue()));
```

Simple generator

```
const auto f = foo();
std::vector<std::string> values;
while (f.advance())
  values.push_back(std::move(f.getValue()));
```

value is returned by non-const reference and can be moved from

Simple generator

```
const auto f = foo();
std::vector<std::string> values;
while (f.advance())
  values.push_back(std::move(f.getValue()));
```

value is returned by non-const reference and can be moved from

What if we want to yield values only by `const` reference?

Simple generator

```
const
std::
while
   val
```

What if we want to yield values only by `const` reference?

```cpp
struct promise_type {
  //...
  std::suspend_always yield_value(const T &value) noexcept {
    result = std::addressof(value);
    return {};
  }
  //...
  const T &getValue() {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));

    return *std::get<const T*>(result);
  }
  //...
private:
  std::variant<std::monostate, const T*, std::exception_ptr> result;
};
```

```
struct promise_type {
  //...
  std::suspend_always yield_value(const T &value) noexcept {
    result = std::addressof(value);
    return {};
  }
  //...
  const T &getValue() {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));

    return *std::get<const T*>(result);
  }
  //...
private:
  std::variant<std::monostate, const T*, std::exception_ptr> result;
};
```

```cpp
struct promise_type {
  //...
  std::suspend_always yield_value(const T &value) noexcept {
    result = std::addressof(value);
    return {};
  }
  //...
  const T &getValue() {
    if (hasException())
      std::rethrow_exception(std::get<std::

    return *std::get<const T*>(result);
  }
  //...
private:
  std::variant<std::monostate, const T*, std::exception_ptr> result;
};
```

```cpp
template<typename T>
struct Generator<const T> {
  struct promise_type;

  //...

  auto &getValue() const;

  //...
};
```

Simple generator

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);
  std::suspend_always yield_value(T &&value) noexcept;

  void return_void() const noexcept {}

  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  bool hasException() const noexcept;
  T &getValue();

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
};
```

Simple generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  bool advance() const noexcept;
  auto &getValue() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

Simple generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  bool advance() const noexcept;
  auto &getValue() const;

private:
  explicit Generator(promise_type &promise) noexcept;

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

Simple generator

```
template<typename T>
struct Generator {
  //...
  bool advance() const noexcept {
    coro();
    return !coro.done() || coro.promise().hasException();
  }
  auto &getValue() const {
    return coro.promise().getValue();
  }

  //...
};
```

Simple generator

```
template<typename T>
struct Generator {
  //...
  bool advance() const noexcept {
    coro();
    return !coro.done() || coro.promise().hasException();
  }
  auto &getValue() const {
    return coro.promise().getValue();
  }

  //...
};
```

Simple generator

```
template<typename T>
struct Generator {
  //...
  bool advance() const noexcept {
    coro();
    return !coro.done() || coro.promise().hasException();
  }
  auto &getValue() const {
    return coro.promise().getValue();
  }

  //...
};
```

precondition:
`advance()` must be called and return `true`

Simple generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}

const auto g = bar();
try {
  while (g.advance())
    std::cout << g.getValue() << '\n';
}
catch (const std::exception &e) {
  std::cout << "exception: " << e.what() << '\n';
}
```

Simple generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}

const auto g = bar();
try {
  while (g.advance())
    std::cout << g.getValue() << '\n';
}
catch (const std::exception &e) {
  std::cout << "exception: " << e.what() << '\n';
}
```

may throw exception from within the coroutine

Simple generator

```
const auto g = bar();
//...
try {
  if (g.advance())                  // has exception, not value
    std::cout << g.getValue() << '\n';  // throws
}
catch (const std::exception &e) {
  std::cout << "exception: " << e.what() << '\n';
}

std::cout << g.getValue() << '\n'; // still has exception
                                   // and throws
```

Simple generator

```
const auto g = bar();
//...
try {
  if (g.advance())                   // has exception, not value
    std::cout << g.getValue() << '\n';  // throws
}
catch (const std::exception &e) {
  std::cout << "exception: " << e.what() << '\n';
}

std::cout << g.getValue() << '\n'; // still has exception
                                   // and throws
```

Simple generator

```
const auto g = bar();
//...
try {
  if (g.advance())                 // has exception, not value
    std::cout << g.getValue() << '\n';  // throws
}
catch (const std::exception &e) {
  std::cout << "exception: " << e.what() << '\n';
}

std::cout << g.getValue() << '\n'; // still has exception
                                   // and throws
```

Simple generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  bool advance() const noexcept;
  auto &getValue() const;
  bool hasException() const noexcept;

private:
  explicit Generator(promise_type &promise) noexcept;

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

Simple generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  bool advance() const noexcept;
  auto &getValue() const;
  bool hasException() const n

private:
  explicit Generator(promise_type &promise) noexcept;

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

```
bool hasException() const noexcept {
  return coro.promise().hasException();
}
```

Simple generator

```
const auto g = bar();
while (g.advance()) {
  if (g.hasException()) {
    // handle exception
  }

  std::cout << g.getValue() << '\n';
}
```

Simple generator

```
const auto g = bar();
while (g.advance()) {
  if (g.hasException()) {
    // handle exception
  }

  std::cout << g.getValue() << '\n';
}
```

UGLY

What we want to have

Required operations:

- check if there are values
- get a value
- advance to the next value

What we want to have

Required operations:

- check if there are values
- get a value
- advance to the next value

Iterators and ranges:

```
auto i = r.begin()
if (i != r.end()) {  // check
  auto x = *i;       // get
  ++i;               // advance
}
```

What we want to have

```
const auto f = foo();
auto i = f.begin();
std::cout << *i << ' ' << *(++i) << '\n';
```

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

What we want to have

```
const auto f = foo();
auto i = f.begin();
std::cout << *i << ' ' << *(++i) << '\n';
```

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
template<typename T>
struct Generator {
  struct promise_type;
  struct Iterator;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  Iterator begin() const;
  Iterator end() const noexcept;

private:
  explicit Generator(promise_type &promise) noexcept;

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

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value)
    noexcept(std::is_nothrow_copy_constructible_v<T>);
  std::suspend_always yield_value(T &&value) noexcept;

  void return_void() const noexcept {}

  void unhandled_exception()
    noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  bool isValueInitialized() const noexcept;
  T &getValue() noexcept;
  bool hasException() const noexcept;
  void throwIfException() const;

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
};
```

```cpp
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  std::suspend_always final_suspend() const noexcept;

  std::suspend_always yield_value(const T &value)
    noexcept(std::is_nothrow_copy_constructible_v<T>);
  std::suspend_always yield_value(T &&value) noexcept;

  void return_void() const noexcept {}

  void unhandled_exception()
    noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  bool isValueInitialized() const noexcept;
  T &getValue() noexcept;
  bool hasException() const noexcept;
  void throwIfException() const;

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
};
```

```cpp
struct promise_type {
//...
  bool isValueInitialized() const noexcept {
    return !std::holds_alternative<std::monostate>(result);
  }
  T &getValue() noexcept {
    return std::holds_alternative<T>(result) ? std::get<T>(result) :
                                               *std::get<T*>(result);
  }
  bool hasException() const noexcept {
    return std::holds_alternative<std::exception_ptr>(result);
  }
  void throwIfException() const {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));
  }
//...
};
```

```cpp
struct promise_type {
//...
  bool isValueInitialized() const noexcept {
    return !std::holds_alternative<std::monostate>(result);
  }
  T &getValue() noexcept {
    return std::holds_alternative<T>(result) ? std::get<T>(result) :
                                               *std::get<T*>(result);
  }
  bool hasException() const noexcept {
    return std::holds_alternative<std::exception_ptr>(result);
  }
  void throwIfException() const {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));
  }
//...
};
```

```cpp
struct promise_type {
//...
  bool isValueInitialized() const noexcept {
    return !std::holds_alternative<std::monostate>(result);
  }
  T &getValue() noexcept {
    return std::holds_alternative<T>(result) ? std::get<T>(result) :
                                                *std::get<T*>(result);
  }
  bool hasException() const noexcept {
    return std::holds_alternative<std::exception_ptr>(result);
  }
  void throwIfException() const {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));
  }
//...
};
```

```cpp
struct promise_type {
//...
  bool isValueInitialized() const noexcept {
    return !std::holds_alternative<std::monostate>(result);
  }
  T &getValue() noexcept {
    return std::holds_alternative<T>(result) ? std::get<T>(result) :
                                               *std::get<T*>(result);
  }
  bool hasException() const noexcept {
    return std::holds_alternative<std::exception_ptr>(result);
  }
  void throwIfException() const {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));
  }
//...
};
```

Range generator

```
struct Iterator {
  // iterator boilerplate

  Iterator() noexcept = default;
  explicit Iterator(const std::coroutine_handle<promise_type> &coro) noexcept;

  friend bool operator==(const Iterator&, const Iterator&) noexcept = default;
  friend bool operator!=(const Iterator&, const Iterator&) noexcept = default;

  Iterator &operator++();
  auto &operator*() const noexcept;

private:
  const std::coroutine_handle<promise_type> *coro = nullptr;
};
```

Range generator

```
struct Iterator {
  using iterator_category = std::input_iterator_tag;
  using difference_type = std::ptrdiff_t;  // doesn't make sense for input iterator
  using value_type = T;
  using reference = T&;
  using pointer = T*;

  Iterator() noexcept = default;
  explicit Iterator(const std::coroutine_handle<promise_type> &coro) noexcept :
    coro{ &coro }
  {}

  friend bool operator==(const Iterator&, const Iterator&) noexcept = default;
  friend bool operator!=(const Iterator&, const Iterator&) noexcept = default;
  //...
};
```

Range generator

```
struct Iterator {
  using iterator_category = std::input_iterator_tag;
  using difference_type = std::ptrdiff_t;  // doesn't make sense for input iterator
  using value_type = T;
  using reference = T&;
  using pointer = T*;

  Iterator() noexcept = default;
  explicit Iterator(const std::coroutine_handle<promise_type> &coro) noexcept :
    coro{ &coro }
  {}

  friend bool operator==(const Iterator&, const Iterator&) noexcept = default;
  friend bool operator!=(const Iterator&, const Iterator&) noexcept = default;
  //...
};
```

Range generator

```
struct Iterator {
  using iterator_category = std::input_iterator_tag;
  using difference_type = std::ptrdiff_t;  // doesn't make sense for input iterator
  using value_type = T;
  using reference = T&;
  using pointer = T*;

  Iterator() noexcept = default;
  explicit Iterator(const std::coroutine_handle<promise_type> &coro) noexcept :
    coro{ &coro }
  {}

  friend bool operator==(const Iterator&, const Iterator&) noexcept = default;
  friend bool operator!=(const Iterator&, const Iterator&) noexcept = default;
  //...
};
```

Range generator

```
struct Iterator {
  //...
  Iterator &operator++() {
    assert(coro != nullptr);
    assert(!coro->done());

    coro->resume();
    if (coro->done()) {
      auto coroHandle = std::exchange(coro, nullptr);
      coroHandle->promise().throwIfException();
    }
    return *this;
  }
  //...
};
```

precondition:

can increment

`++i`

only if

`i != end()`

and coroutine is not finished

Range generator

```
struct Iterator {
  //...
  Iterator &operator++() {
    assert(coro != nullptr);
    assert(!coro->done());

    coro->resume();
    if (coro->done()) {
      auto coroHandle = std::exchange(coro, nullptr);
      coroHandle->promise().throwIfException();
    }
    return *this;
  }
  //...
};
```

Range generator

```
struct Iterator {
  //...
  Iterator &operator++() {
    assert(coro != nullptr);
    assert(!coro->done());

    coro->resume();
    if (coro->done()) {
      auto coroHandle = std::exchange(coro, nullptr);
      coroHandle->promise().throwIfException();
    }
    return *this;
  }
  //...
};
```

Range generator

```
struct Iterator {
  //...
  Iterator &operator++() {
    assert(coro != nullptr);
    assert(!coro->done());

    coro->resume();
    if (coro->done()) {
      auto coroHandle = std::exchange(coro, nullptr);
      coroHandle->promise().throwIfException();
    }
    return *this;
  }
  //...
};
```

Range generator

```
struct Iterator {
  //...
  Iterator &operator++() {
    assert(coro != nullptr);
    assert(!coro->done());

    coro->resume();
    if (coro->done()) {
      auto coroHandle = std::exchange(coro, nullptr);
      coroHandle->promise().throwIfException();
    }
    return *this;
  }
  //...
};
```

Range generator

```
struct Iterator {
  //...
  Iterator &operator++() {
    assert(coro != nullptr);
    assert(!coro->done());

    coro->resume();
    if (coro->done()) {
      auto coroHandle = std::exchange(coro, nullptr);
      coroHandle->promise().throwIfException();
    }
    return *this;
  }
  //...
};
```

Range generator

```
const auto g = bar();
auto k = g.begin();

auto i = k;  // 'i' and 'k' both refer to 'g.begin()'
while (i != g.end())
  ++i;

assert(i == g.end());
// 'k' is invalid
```

Iterators are invalidated when generator coroutine finishes.

(Except the `end()` sentinel iterator.)

Range generator

```
struct Iterator {
  //...
  auto &operator*() const noexcept {
    assert(coro != nullptr);
    assert(!coro->done());

    return coro->promise().getValue();
  }

private:
  const std::coroutine_handle<promise_type> *coro = nullptr;
};
```

Range generator

```
template<typename T>
struct Generator {
  struct promise_type;
  struct Iterator;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  Iterator begin() const;
  Iterator end() const noexcept;

private:
  explicit Generator(promise_type &promise) noexcept;

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

Range generator

```
template<typename T>
struct Generator {
  struct promise_type;
  struct Iterator;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  Iterator begin() const;
  Iterator end() const noexcept;

private:
  explicit Generator(promise_type &promise) noexcept;

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

Range generator

```
Iterator begin() const {
  if (coro.done())
    return end();

  auto i = Iterator{ coro };
  if (!coro.promise().isValueInitialized())
    ++i;  // can throw, or become '*this == end()'
  return i;
}

Iterator end() const noexcept {
  return {};
}
```

Range generator

```
Iterator begin() const {
  if (coro.done())
    return end();

  auto i = Iterator{ coro };
  if (!coro.promise().isValueInitialized())
    ++i;  // can throw, or become '*this == end()'
  return i;
}

Iterator end() const noexcept {
  return {};
}
```

Range generator

```
Iterator begin() const {
  if (coro.done())
    return end();

  auto i = Iterator{ coro };
  if (!coro.promise().isValueInitialized())
    ++i;  // can throw, or become '*this == end()'
  return i;
}

Iterator end() const noexcept {
  return {};
}
```

Range generator

```
Iterator begin() const
  if (coro.done())
    return end();

  auto i = Iterator{ coro };
  if (!coro.promise().isValueInitialized())
    ++i;  // can throw, or become '*this == end()'
  return i;
}

Iterator end() const noexcept {
  return {};
}
```

```
const auto g = bar();
if (g.begin() == g.end())
  //...

for (auto &i : g)  // 'g.begin()' called
  std::cout << i << '\n';
```

Range generator

```cpp
Iterator begin() const {
  if (coro.done())
    return end();

  auto i = Iterator{ coro };
  if (!coro.promise().isValueInitialized())
    ++i;  // can throw, or become '*this == end()'
  return i;
}

Iterator end() const noexcept {
  return {};
}
```

Range generator

```
Iterator begin() const {
  if (coro.done())
    return end();

  auto i = Iterator{ coro };
  if (!coro.promise().isValueInitialized())
    ++i;  // can throw, or become '*this == end()'
  return i;
}

Iterator end() const noexcept {
  return {};
}
```

Range generator

```
template<typename T>
struct Generator {
  struct promise_type;
  struct Iterator;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  Iterator begin() const;
  Iterator end() const noexcept;

private:
  explicit Generator(promise_type &promise) noexcept;

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

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}
```

Range generator

```
const auto g = bar();
try {
  for (auto &i : g)
    std::cout << i << '\n';
}
catch (const std::exception &e) {
  std::cout << "exception: " << e.what() << '\n';
}
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}
```

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}


const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}


const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}


const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}


const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}


const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}


const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n
}   co_return;
```

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n
}   co_return;
```

```
{
  promise.return_void();
  goto final-suspend;
}
```

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}
```

```
struct promise_type {
//...
  std::suspend_always final_suspend() const noexcept;
//...
};
```

Range generator

```
Generator<int> bar() {
   const auto values = getValues();  // may throw
   for (auto n : values)
     co_yield n;
                                            final suspend
}
```

```
struct promise_type {
//...
  std::suspend_always final_suspend() const noexcept;
//...
};
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
                                              final suspend
}
```

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}
```

final suspend

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
                                          final suspend
}
```

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // throws
  for (auto n : values)
    co_yield n;
}


const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // throws
  for (au
    co_yi
}


const aut
for (auto
  std::co
```

```
{
  //...
  try {
    //...
    function-body
  }
  catch (...) {
    //...
    promise.unhandled_exception();
  }
final-suspend:
  //...
}
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // throws
  for (auto n : values)
    co_yield n;
}



const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // throws
  for (auto n : values)
    co_yield n;
}
```

```
struct promise_type {
//...
  std::suspend_always final_suspend() const noexcept;
//...
};
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // throws
  for (auto n : values)
    co_yield n;
}
```

final suspend

```
struct promise_type {
//...
  std::suspend_always final_suspend() const noexcept;
//...
};
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // throws
  for (auto n : values)
    co_yield n;
}
```

final suspend

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // throws
  for (auto n : values)
    co_yield n;
}
```

final suspend

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

Range generator

```
Generator<int> bar() {
  const auto values = getValues();  // throws
  for (auto n : values)
    co_yield n;
}
```

final suspend

```
const auto g = bar();
for (auto &i : g)
  std::cout << i << '\n';
```

`g.begin()` or `++it`
rethrow the exception

Range generator

At *this* point you know *almost everything* you need to know about how generators work.

For real this time.

Range generator

```
const auto g = bar();
for (auto i = g.begin();   // may throw
      i != g.end();
      ++i) {                // may throw
    std::cout <<
      *i                    // never throws
      << '\n';
}
```

```
struct LazyIterator {
  // iterator boilerplate

  LazyIterator() noexcept = default;
  explicit LazyIterator(const std::coroutine_handle<promise_type> &coro) noexcept;

  friend bool operator==(const LazyIterator&, const LazyIterator&) noexcept = default;
  friend bool operator!=(const LazyIterator&, const LazyIterator&) noexcept = default;

  LazyIterator &operator++() noexcept;
  auto &operator*() const;
  friend bool hasException(const LazyIterator &i) noexcept;

private:
  const std::coroutine_handle<promise_type> *coro = nullptr;
};
```

```cpp
struct LazyIterator {
  // iterator boilerplate

  LazyIterator() noexcept = default;
  explicit LazyIterator(const std::coroutine_handle<promise_type> &coro) noexcept;

  friend bool operator==(const LazyIterator&, const LazyIterator&) noexcept = default;
  friend bool operator!=(const LazyIterator&, const LazyIterator&) noexcept = default;

  LazyIterator &operator++() noexcept;
  auto &operator*() const;
  friend bool hasException(const LazyIterator &i) noexcept;

private:
  const std::coroutine_handle<promise_type> *coro = nullptr;
};
```

```
struct LazyIterator {
  // iterator boilerplate

  LazyIterator() noexcept = default;
  explicit LazyIterator(const std::coroutine_handle<promise_type> &coro) noexcept;

  friend bool operator==(const LazyIterator&, const LazyIterator&) noexcept = default;
  friend bool operator!=(const LazyIterator&, const LazyIterator&) noexcept = default;

  LazyIterator &operator++() noexcept;
  auto &operator*() const;
  friend bool hasException(const LazyIterator &i) noexcept;

private:
  const std::coroutine_handle<promise_type> *coro = nullptr;
};
```

Range generator

```
template<typename T>
struct Generator {
  struct promise_type;
  struct LazyIterator;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  LazyIterator begin() const noexcept;
  LazyIterator end() const noexcept;

private:
  explicit Generator(promise_type &promise) noexcept;

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

does not throw

Range generator

```
const auto g = bar();
for (auto i = g.begin();   // does
     i != g.end();         // not
     ++i) {                // throw
  if (hasException(i))
    break;
  std::cout << *i << '\n';
}
```

Range generator

```
const auto g = bar();
for (auto i = g.begin();   // does
     i != g.end();         // not
     ++i) {                // throw
  if (hasException(i))
    break;
  std::cout << *i << '\n';
}
```

```
const auto g = bar();
for (auto i = g.begin(); i != g.end(); ++i) {
  if (hasException(i)) {
    try {
      *i;  // throws
    }
    catch (...) {
    }
    try {
      *i;  // throws
    }
    catch (...) {
    }
    break;
  }
  std::cout << *i << '\n';
}
```

Yielding from nested generators

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}

Generator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  for (auto n : bar())  // 'bar()' is resumed and suspended
    co_yield n;  // yields and suspends, then resumes
}
```

Yielding from nested generators

```
Generator<int> bar() {
  const auto values = getValues();  // may throw
  for (auto n : values)
    co_yield n;
}

Generator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  for (auto n : bar())  // 'bar()' is resumed and suspended
    co_yield n;  // yields and suspends, then resumes
}
```

Yielding from nested generators

```
Generator<int> qux() {
  const auto g
  if (auto i =                              ()) {
    co_yield *
    ++i;
  }
  for (auto i
    co_yield i;  // yield the rest
}
```

Yielding from nested generators

```
Generator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  for (auto i : g)
    co_yield i;  // yield the rest
}
```

Yielding from nested generators

```
Generator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  for (auto i : g)
    co_yield i;  // yield
}
```

up to three resumes & suspends per yielded value!

```
const auto h = qux();
for (auto &i : h)
  std::cout << i << '\n';
```

Recursive generator

```
RecursiveGenerator<int> bar() {
  for (auto n : getValues())
    co_yield n;
}

RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_ thing
}
```

Recursive generator

```
RecursiveGenerator<int> bar() {
  for (auto n : getValues())
    co_yield n;
}

RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_ thing
}
```

Recursive generator

```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```

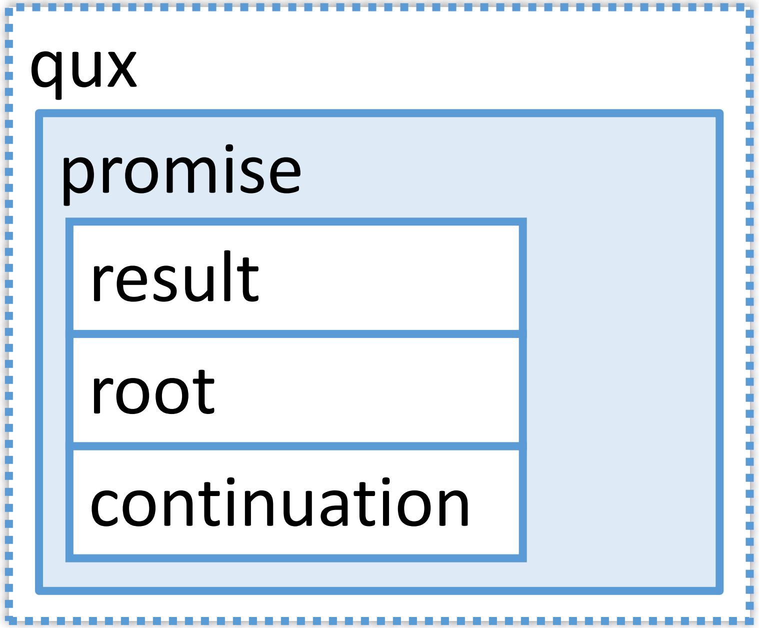
Recursive generator

```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```

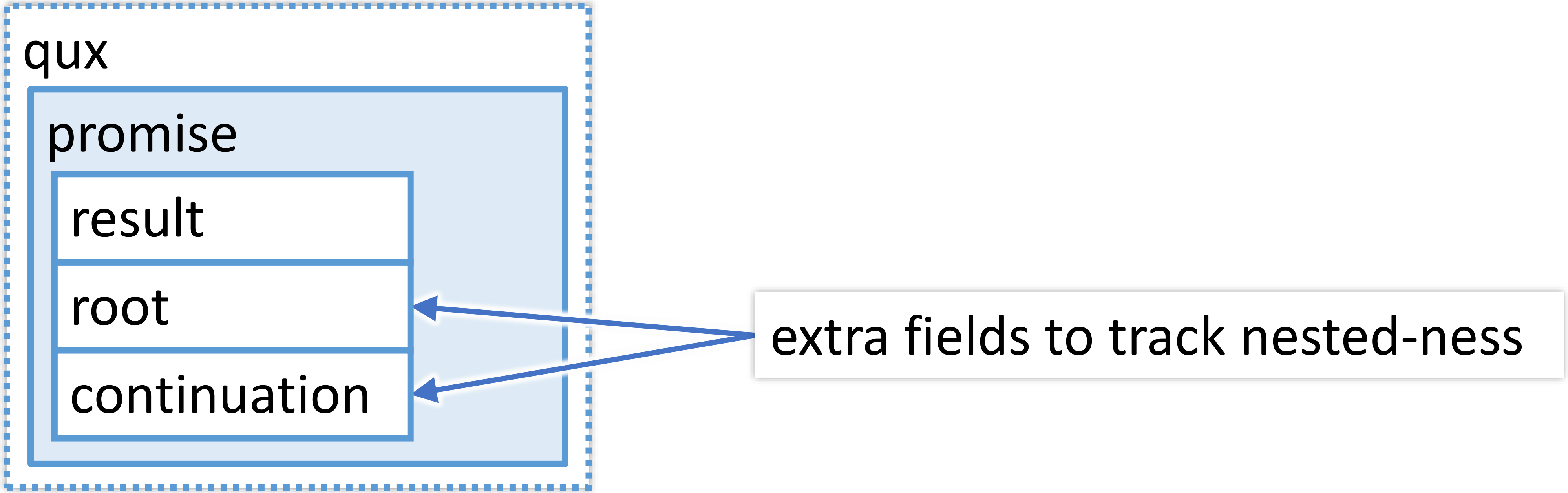
```cpp
const auto h = qux();
for (auto &i : h)
  std::cout << i << '\n';
```

```cpp
const auto h = qux();
for (auto &i : h)
  std::cout << i << '\n';
```

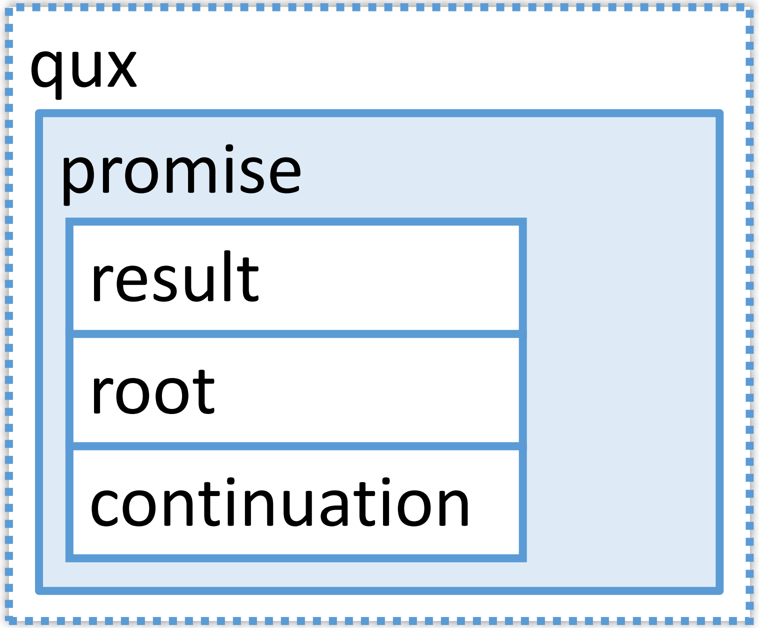
```
const auto h = qux();
for (auto &i : h)
  std::cout << i << '\n';
```



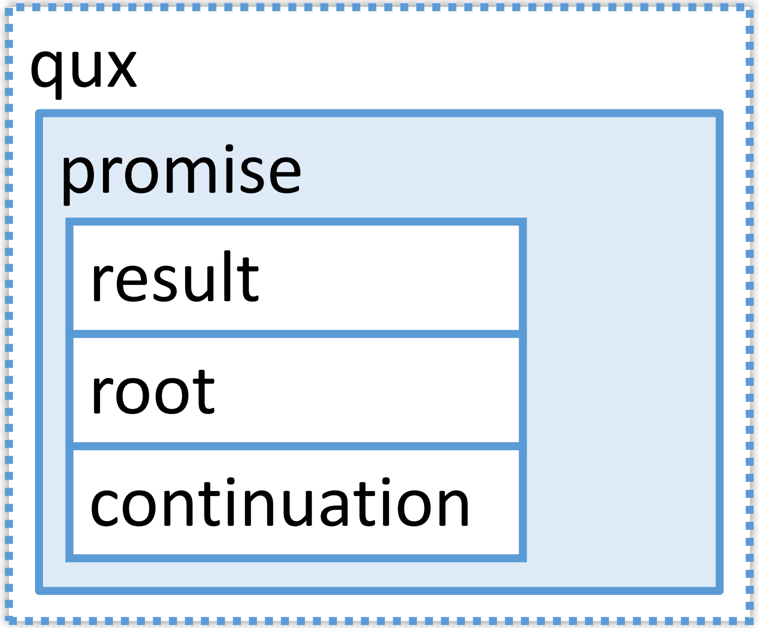
```cpp
const auto h = qux();
for (auto &i : h)
  std::cout << i << '\n';
```



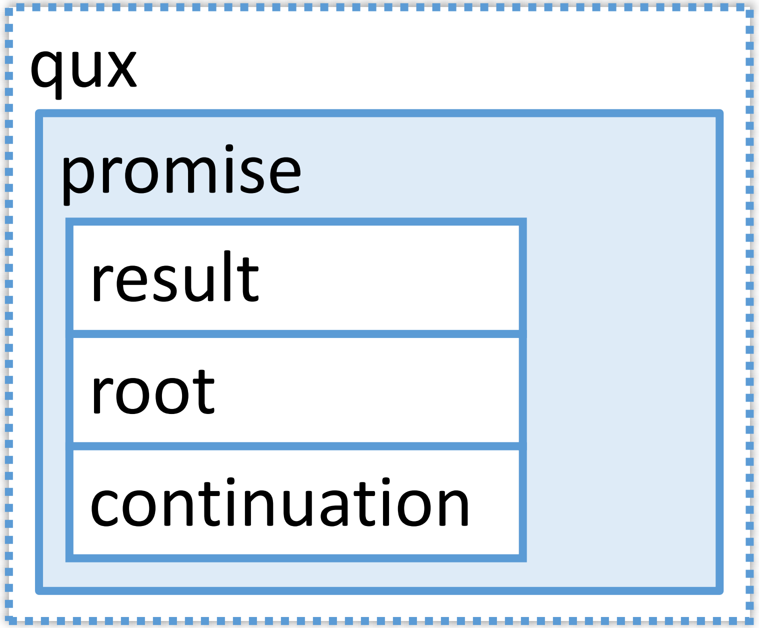
```cpp
const auto h = qux();
for (auto &i : h)
  std::cout << i << '\n';
```



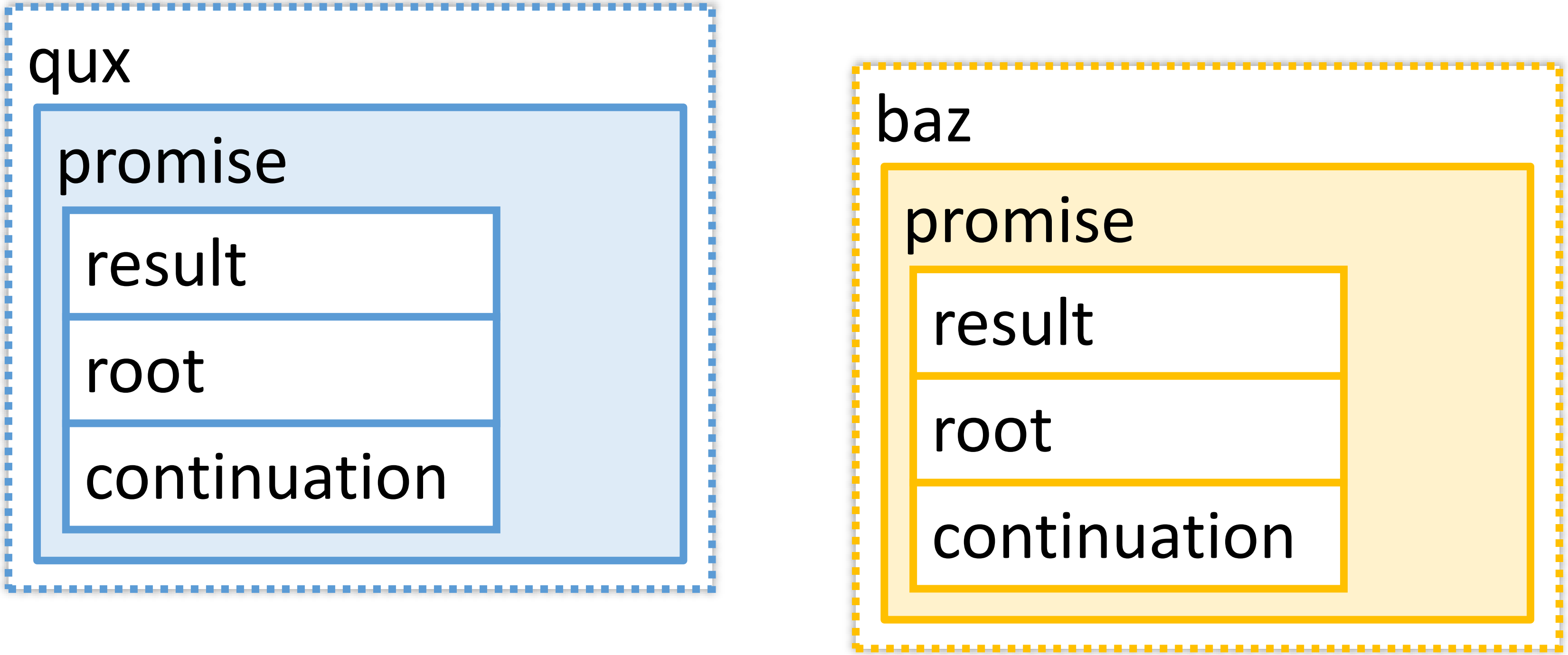
```cpp
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



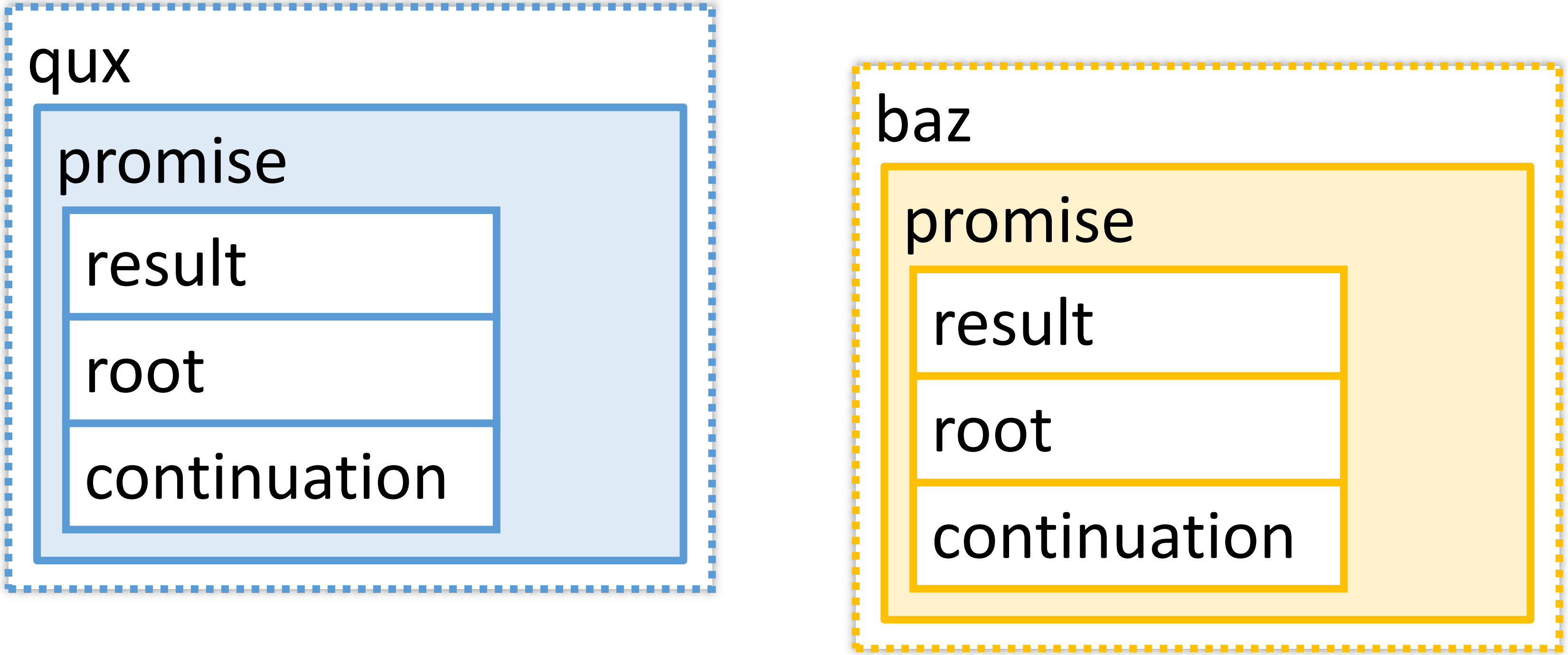
```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



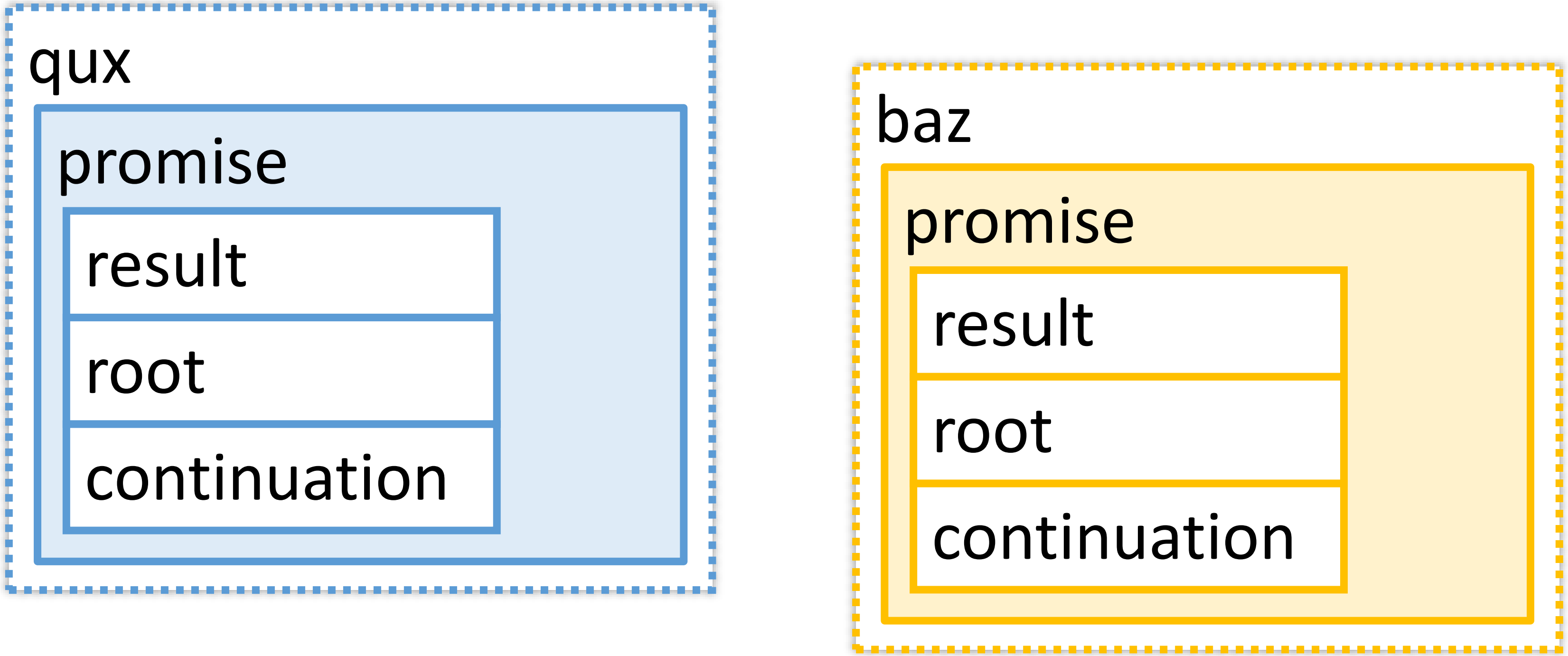
```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



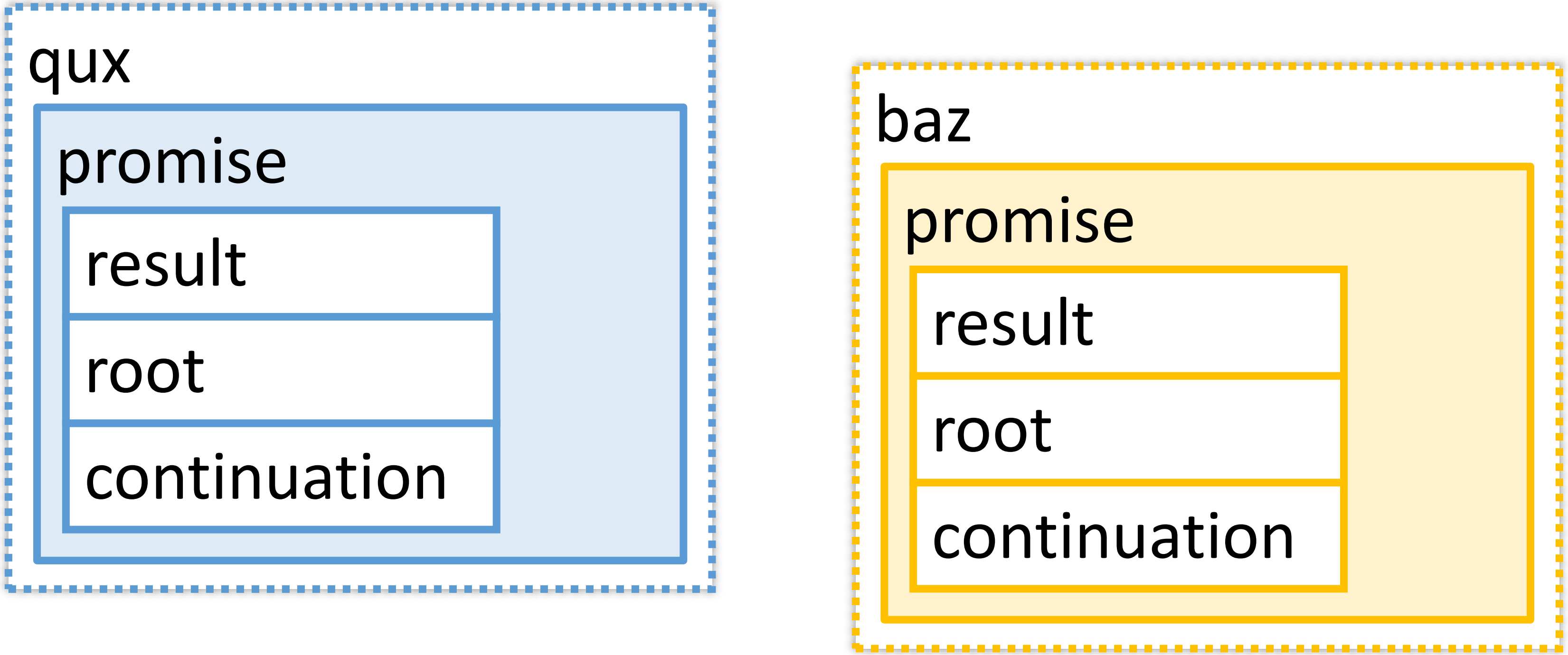
```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



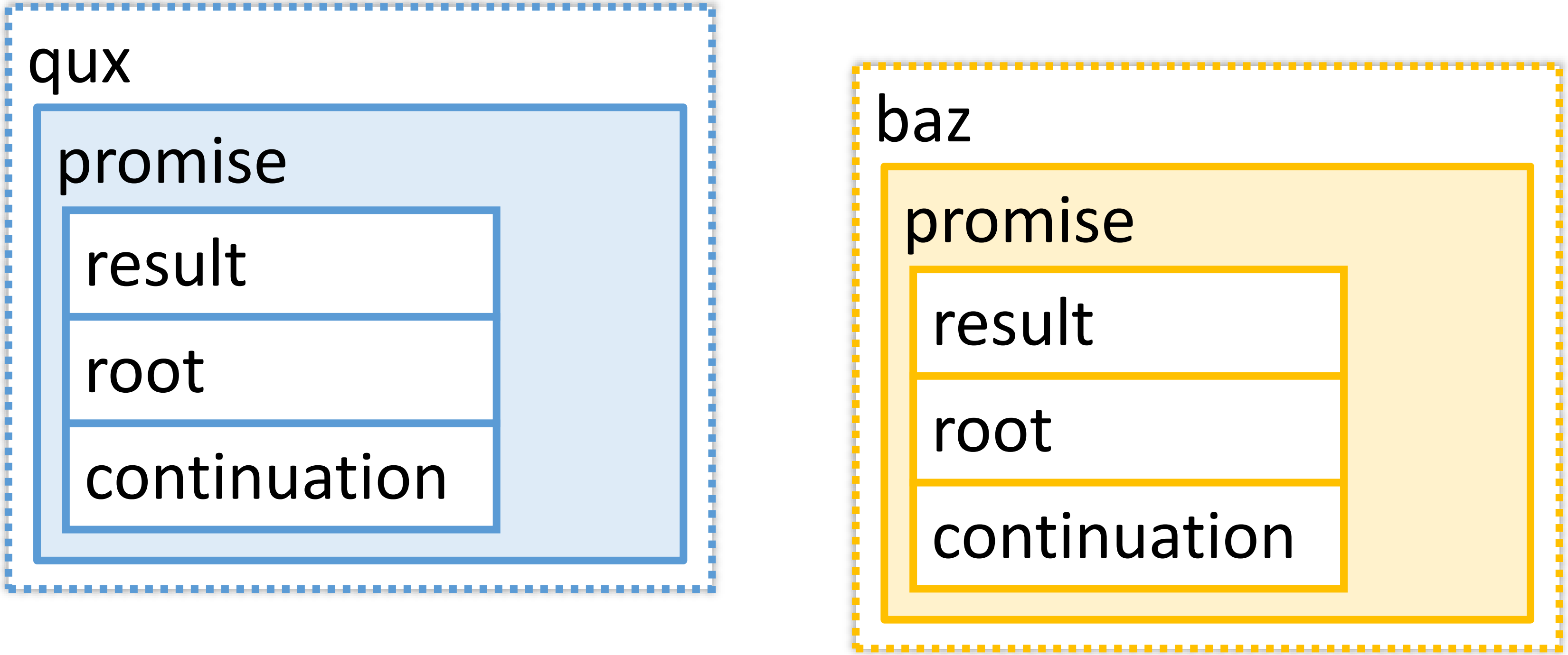
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_ thing
}
```



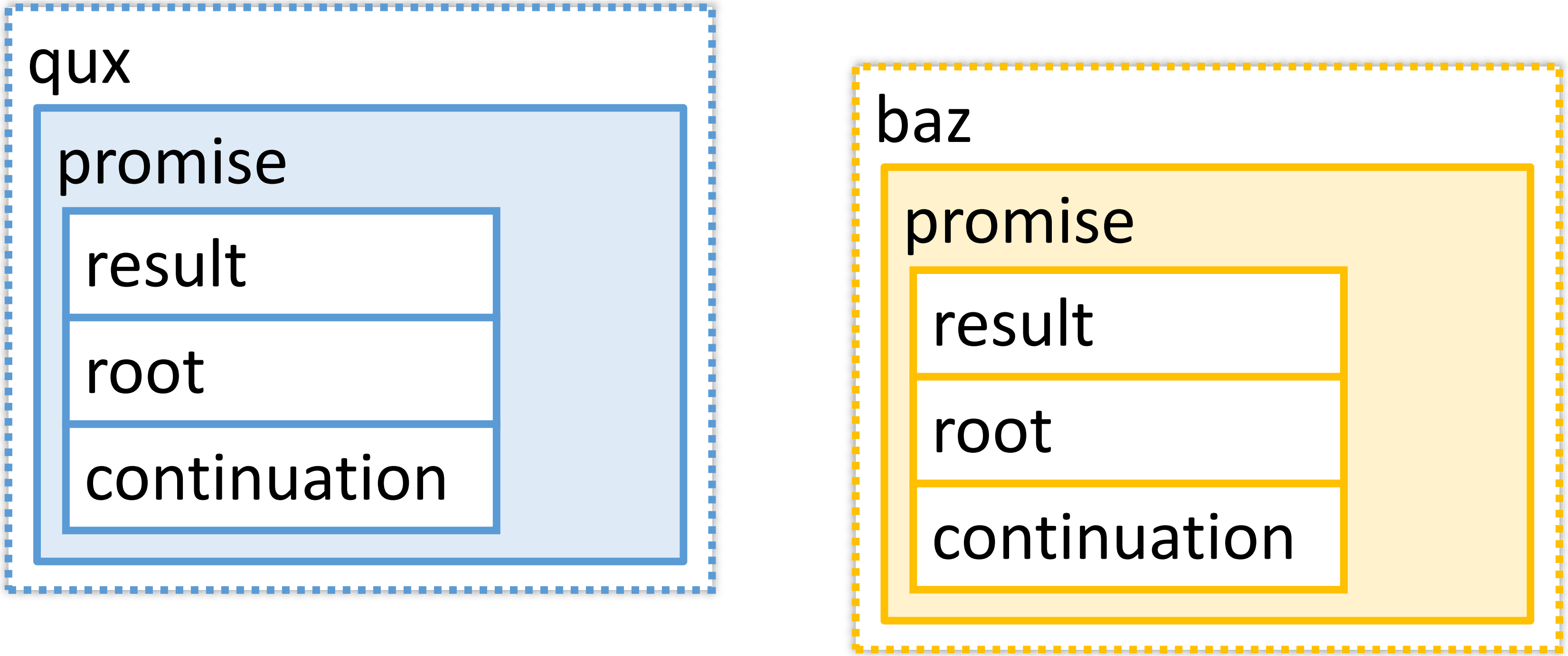
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_ thing
}
```



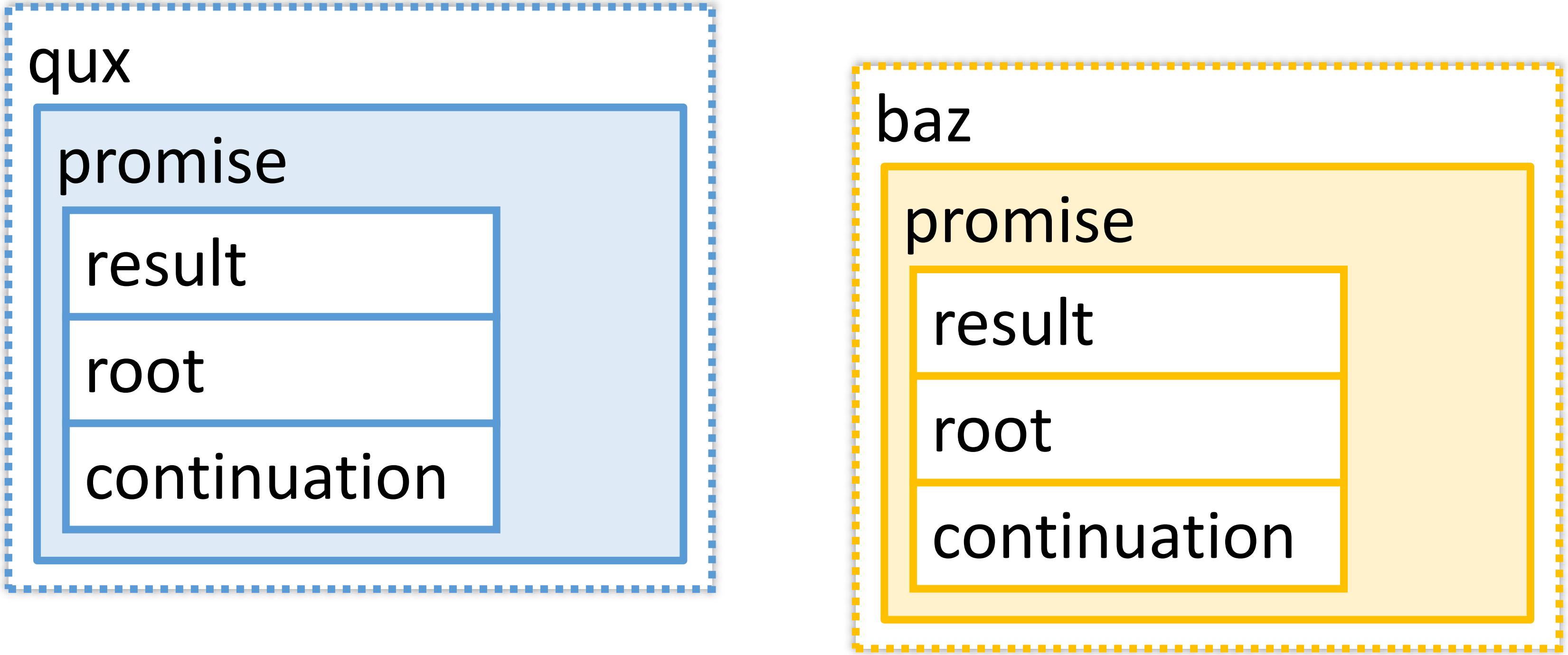
```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



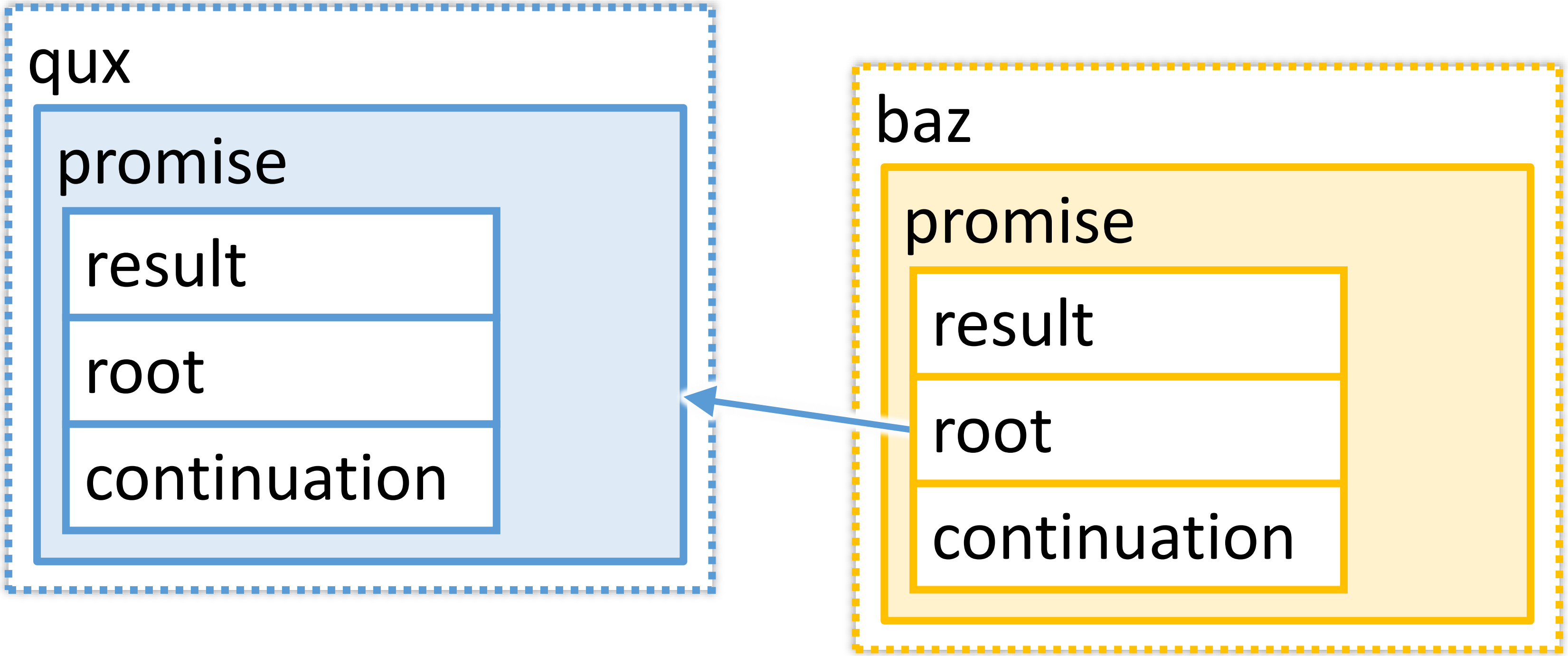
```
const auto h = qux();
for (auto &i : h)
  std::cout << i << '\n';
```



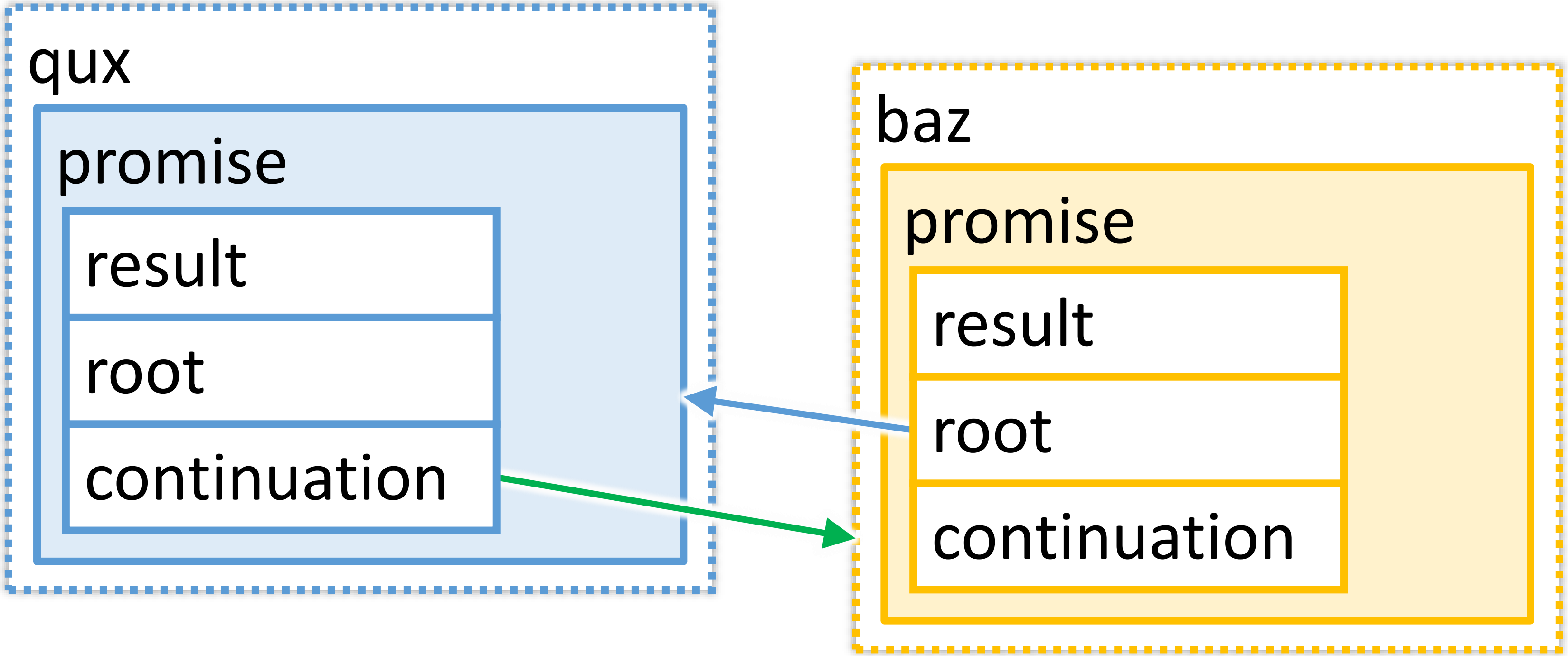
```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



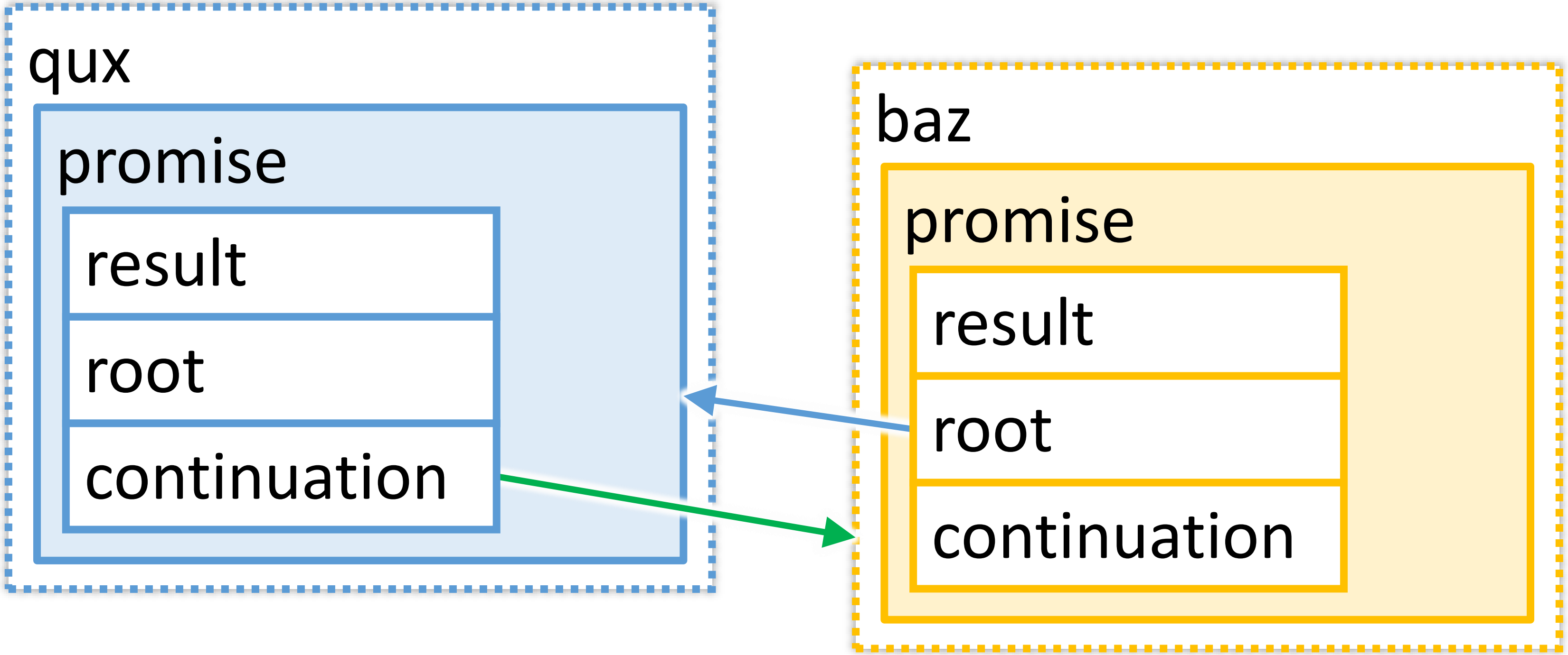
```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



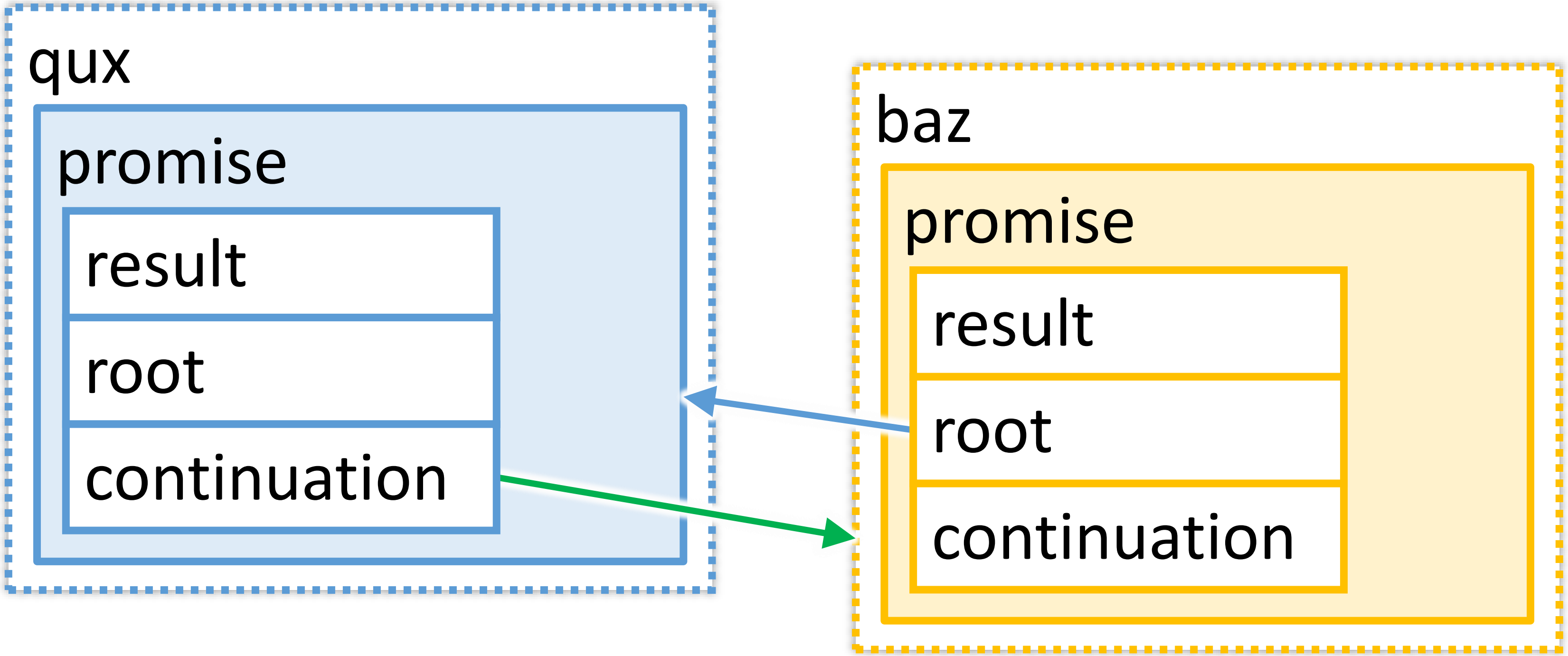
```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



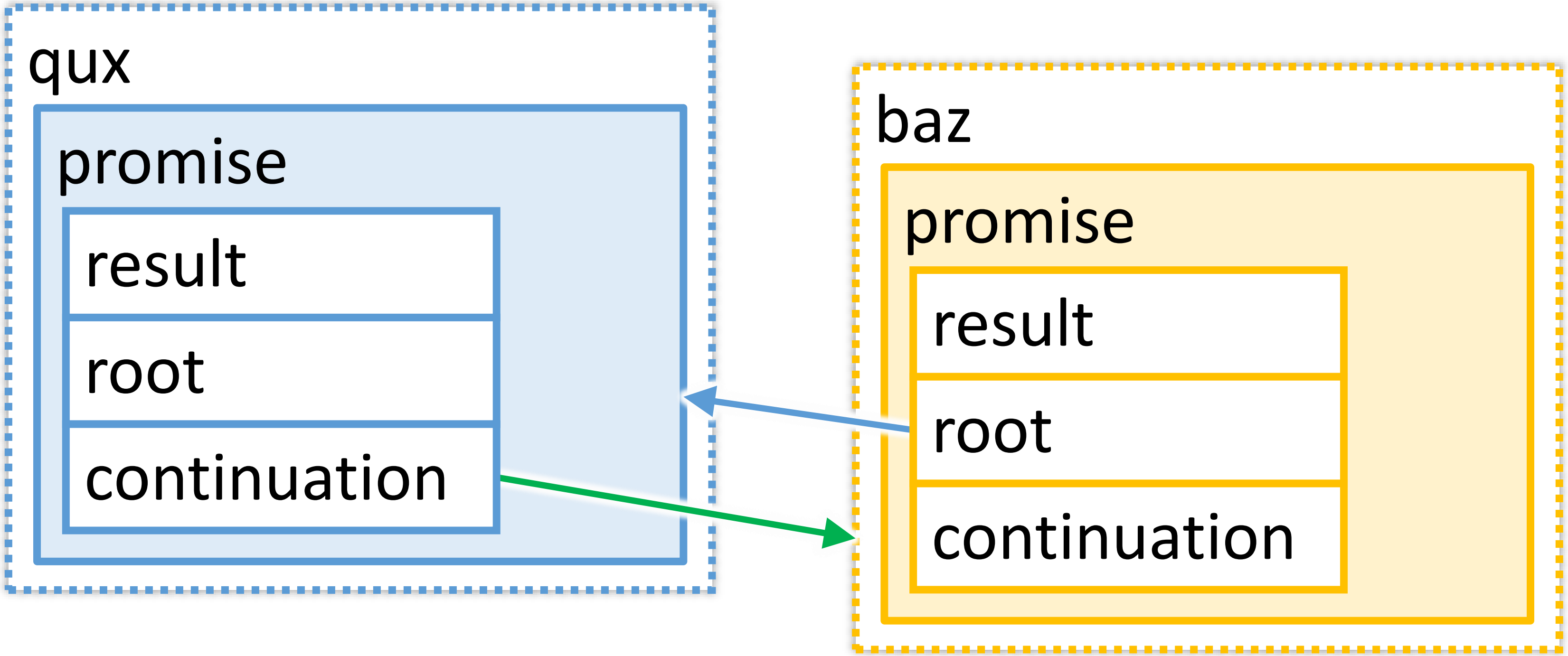
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_ thing
}
```



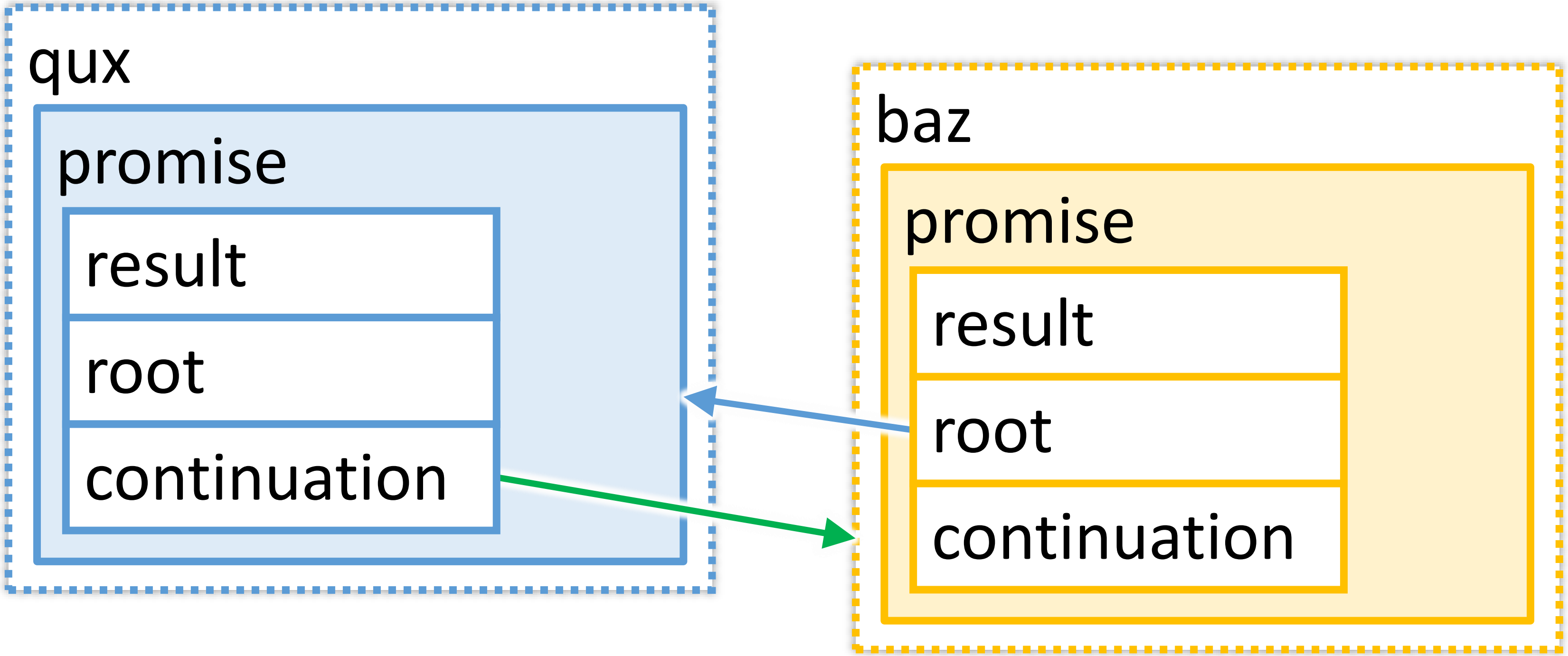
```
const auto h = qux();
for (auto &i : h)
  std::cout << i << '\n';
```



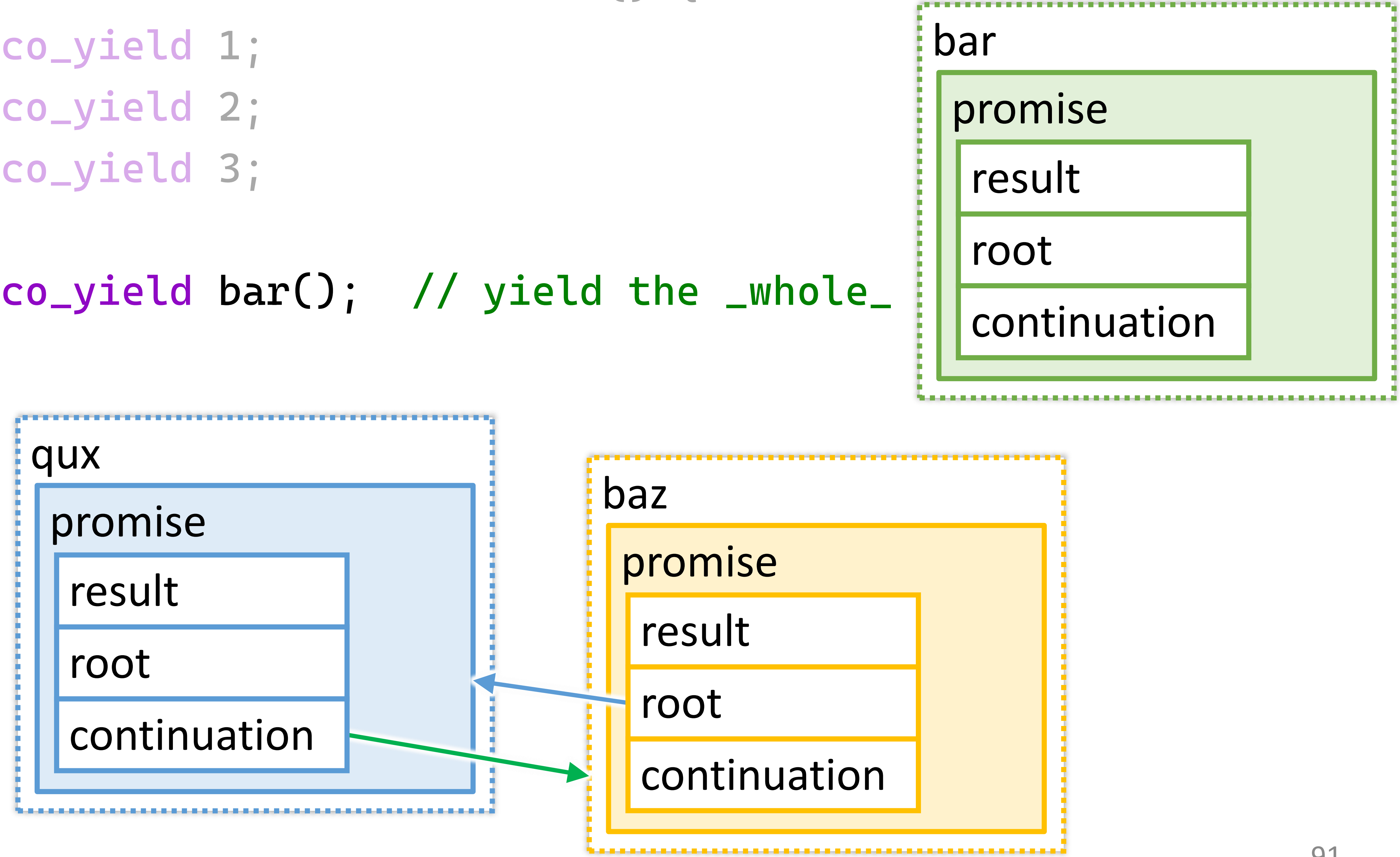
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_ thing
}
```



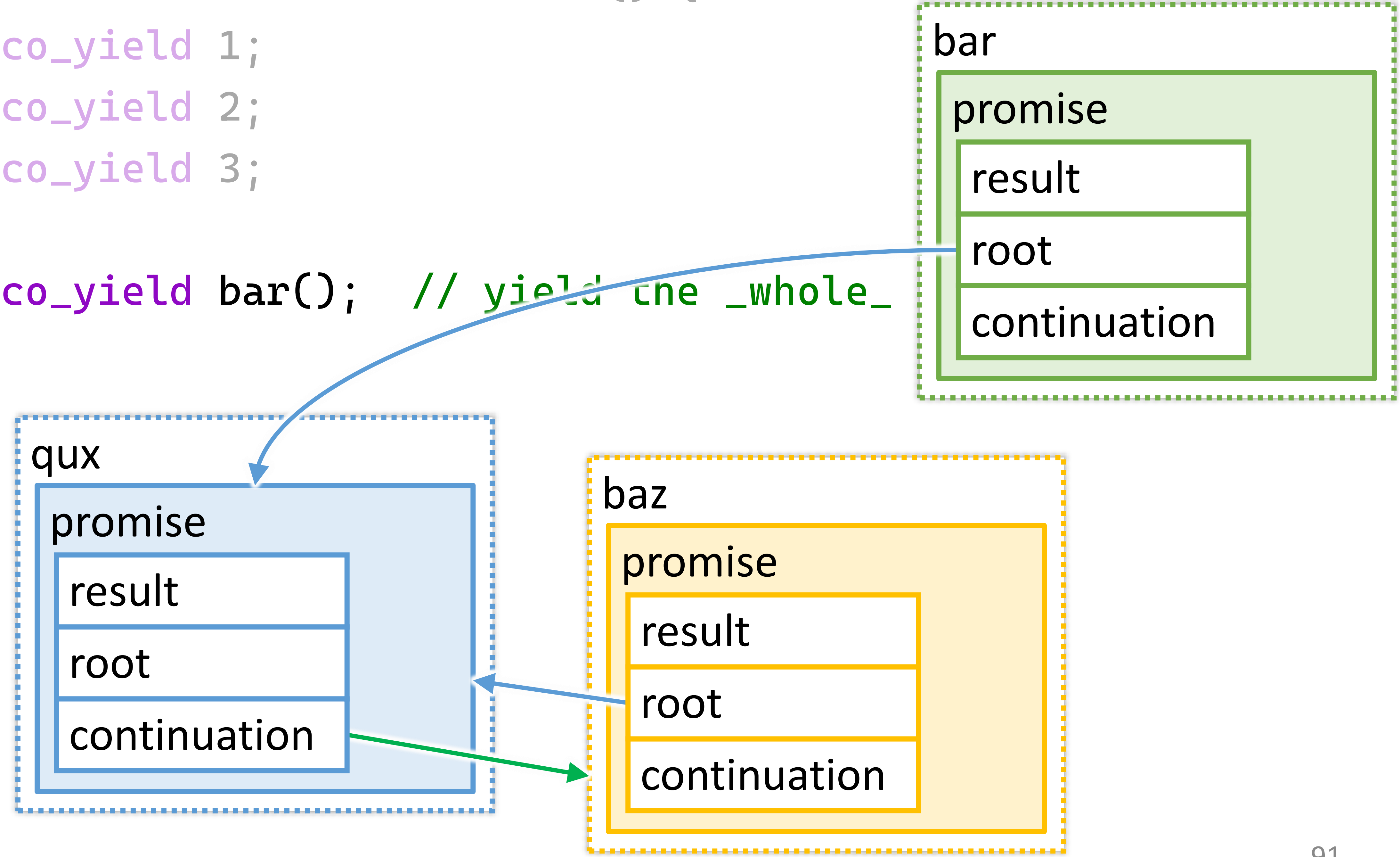
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_ thing
}
```



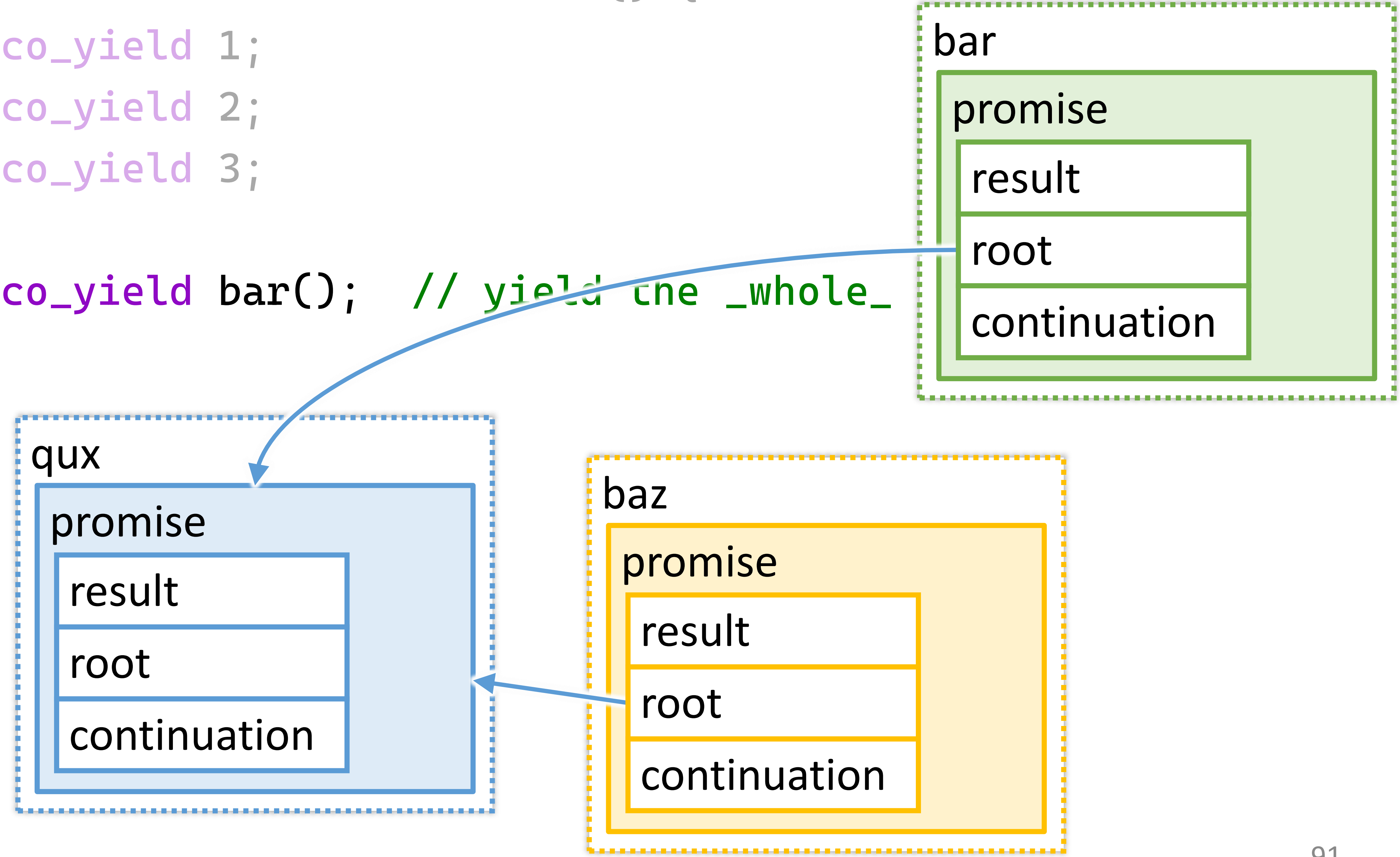
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_
}
```



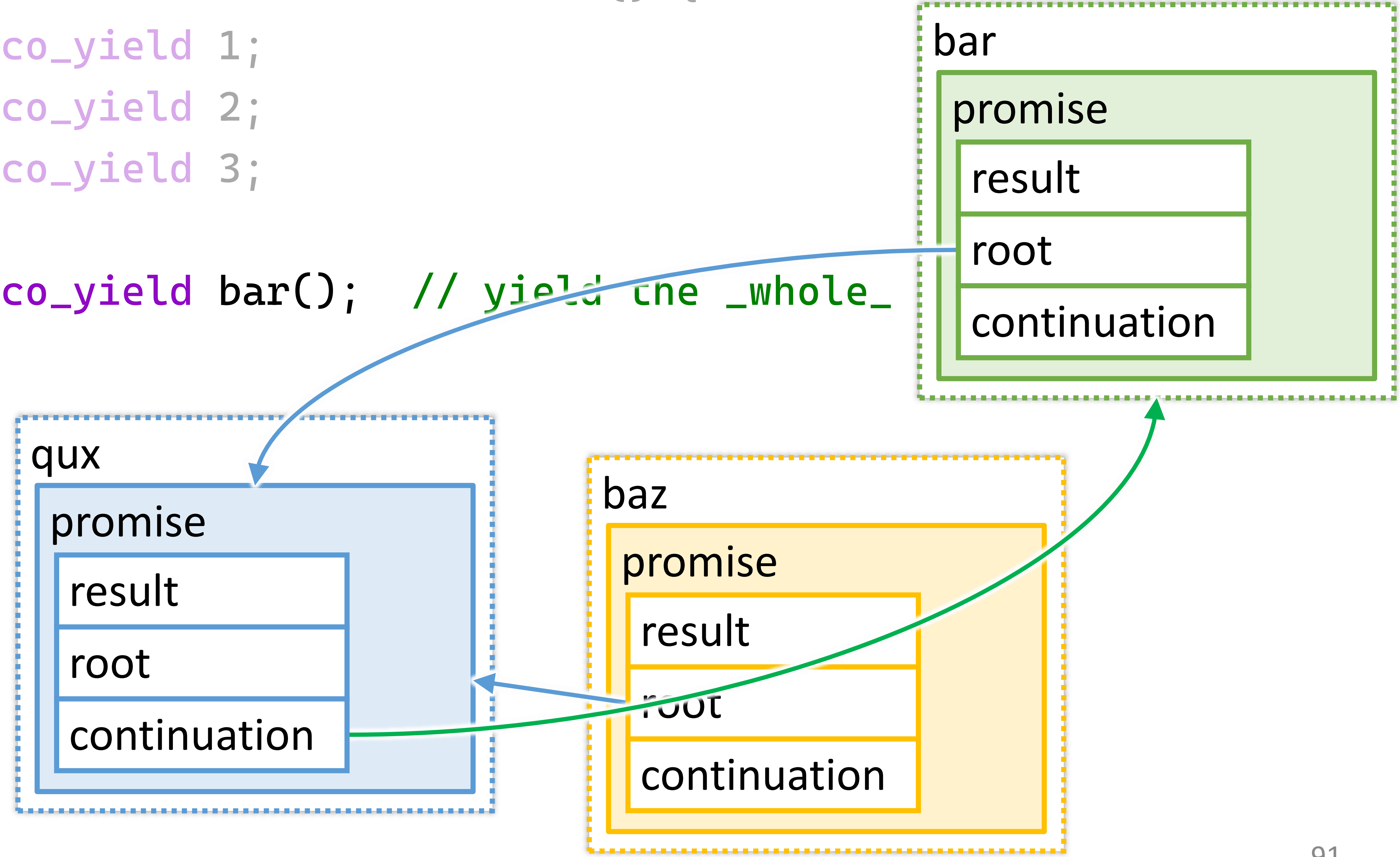
```cpp
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_
}
```



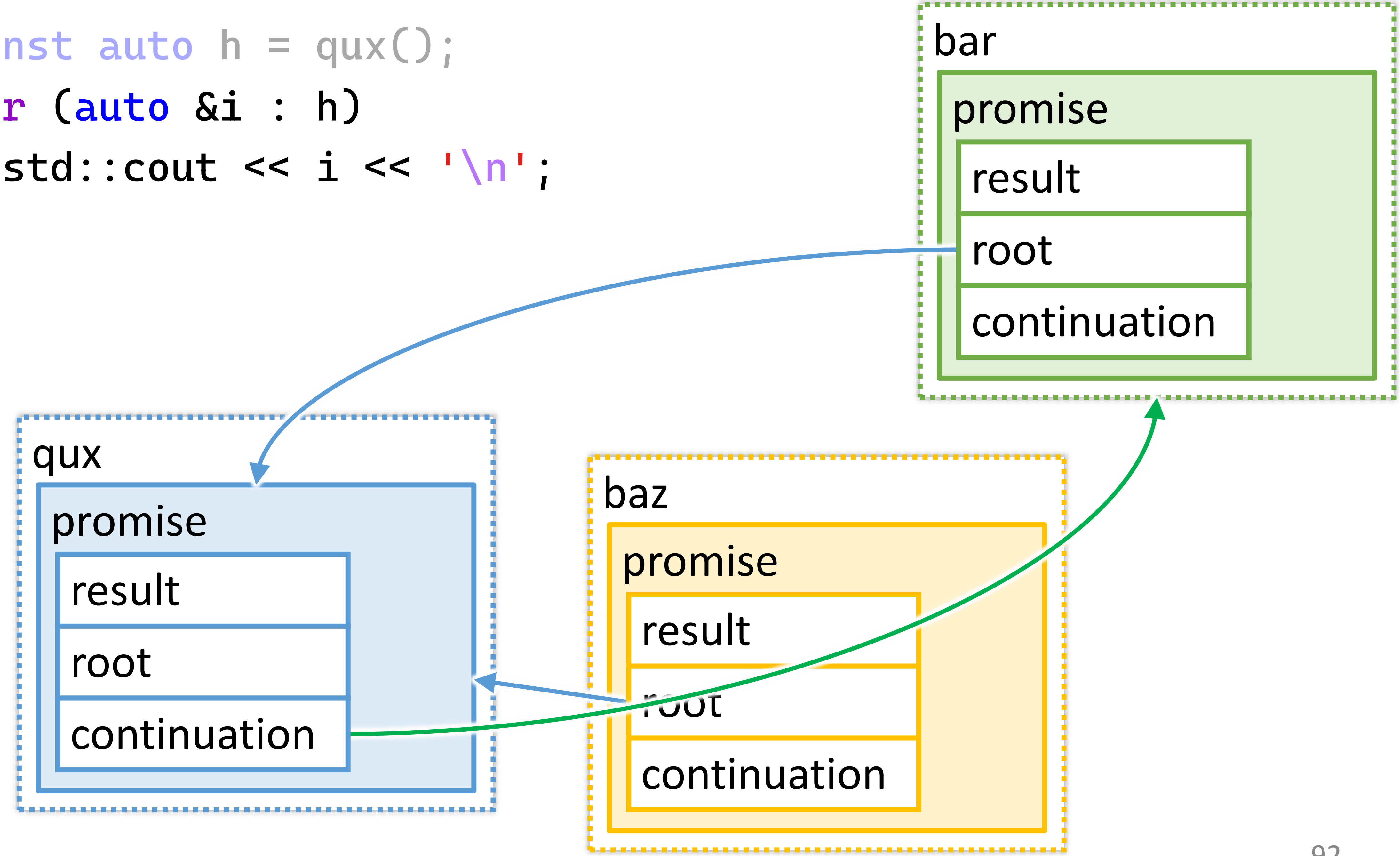
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield _the_ _whole_
}
```



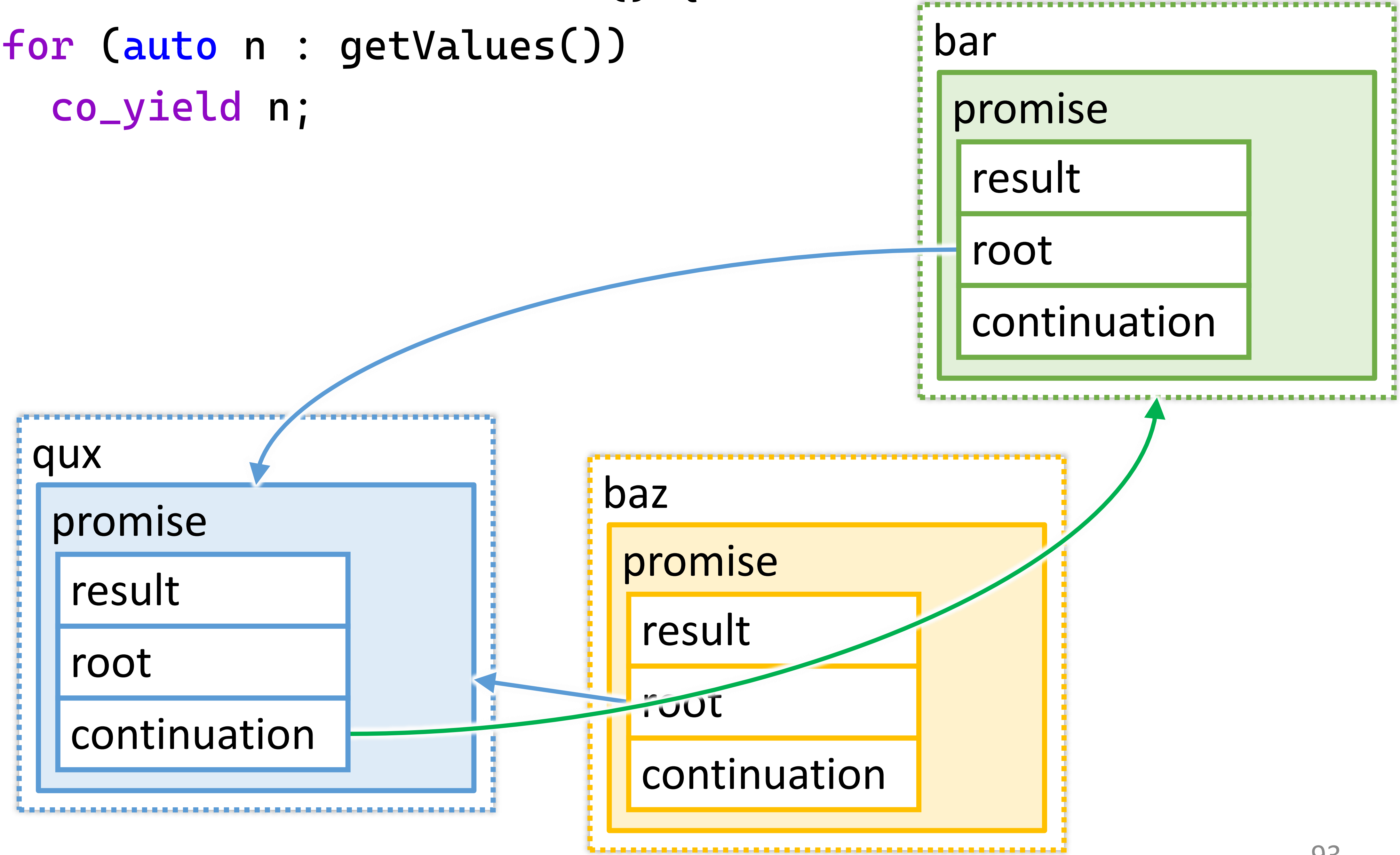
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_
}
```



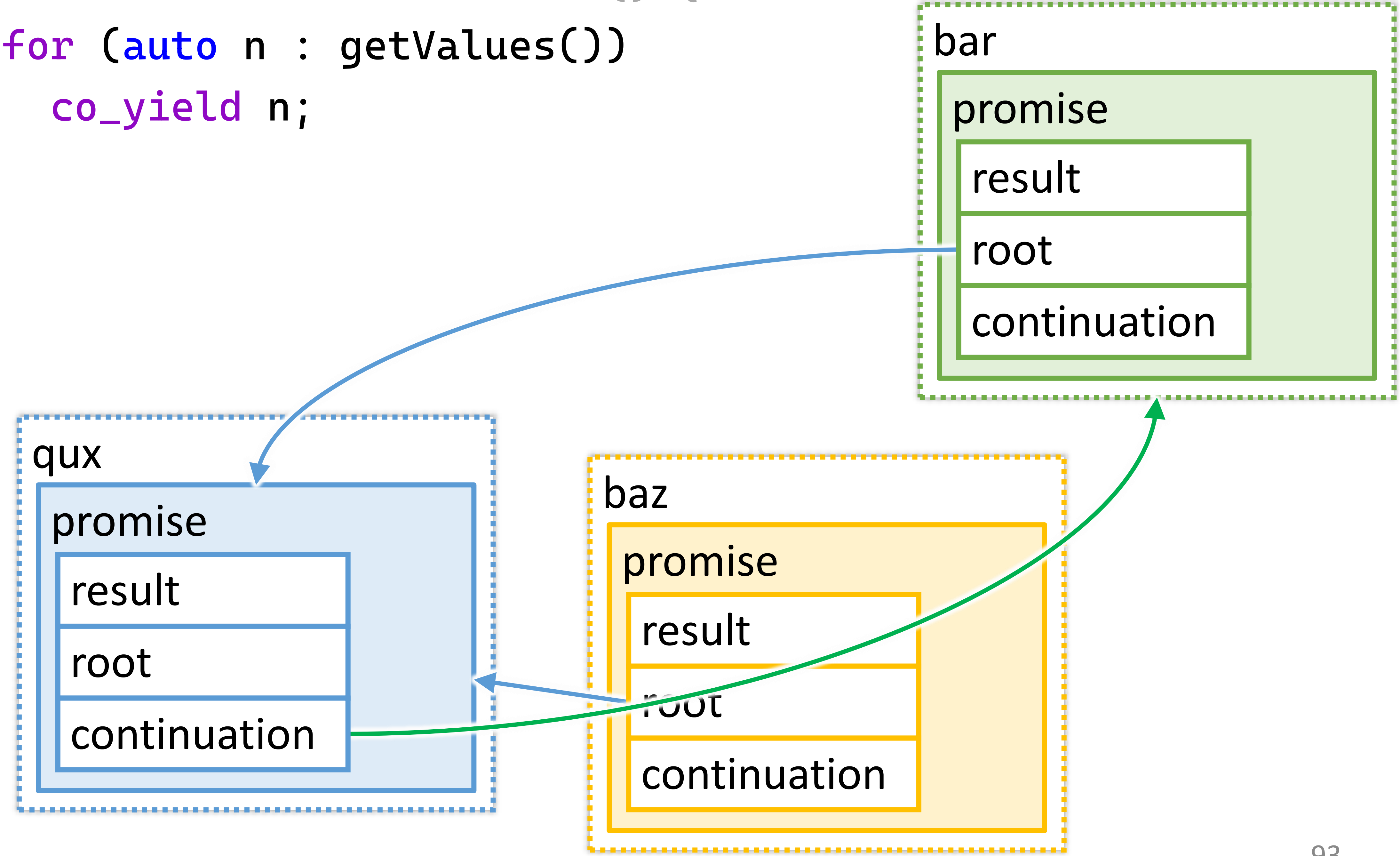
```
const auto h = qux();
for (auto &i : h)
  std::cout << i << '\n';
```



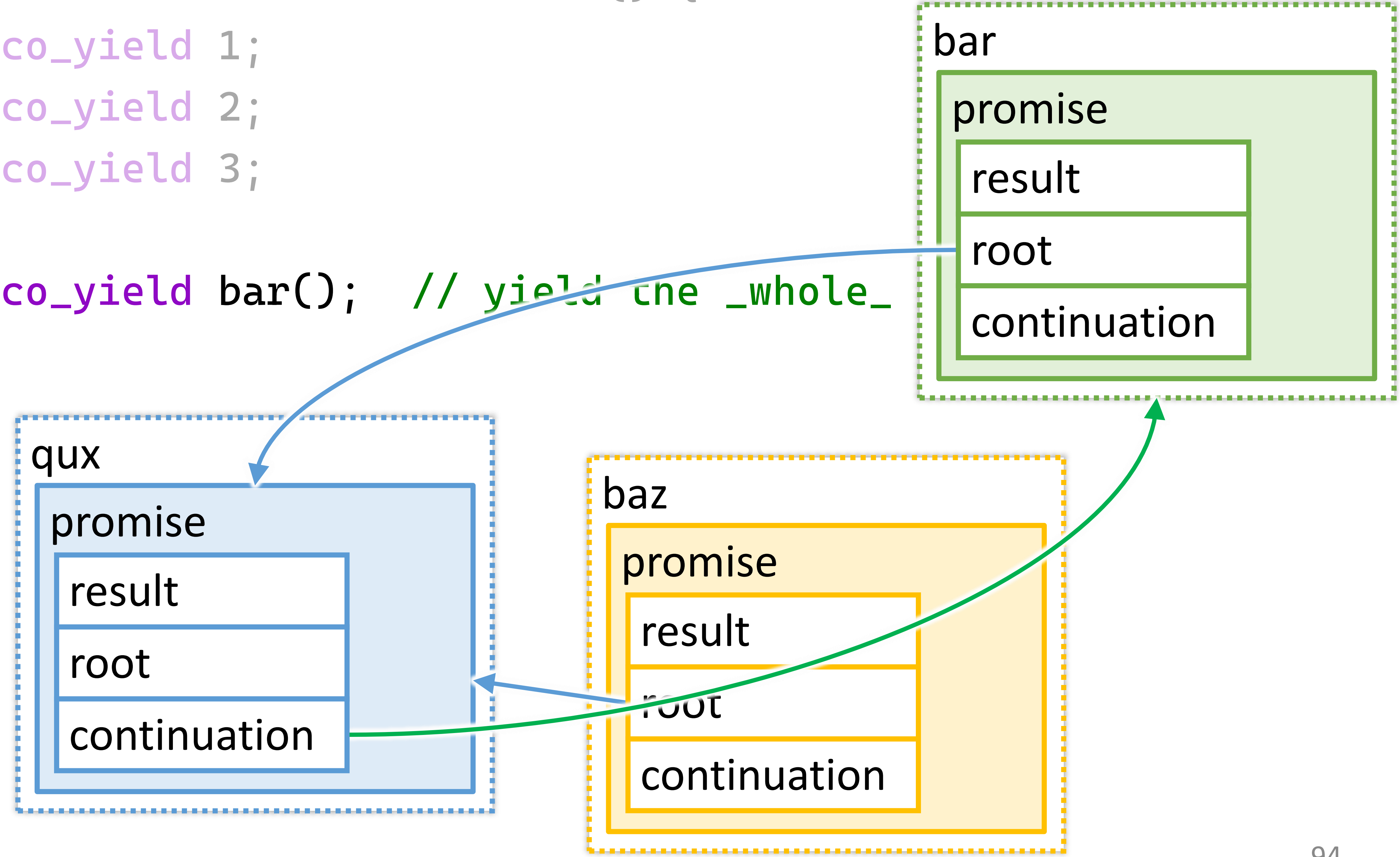
```
RecursiveGenerator<int> bar() {
  for (auto n : getValues())
    co_yield n;
}
```



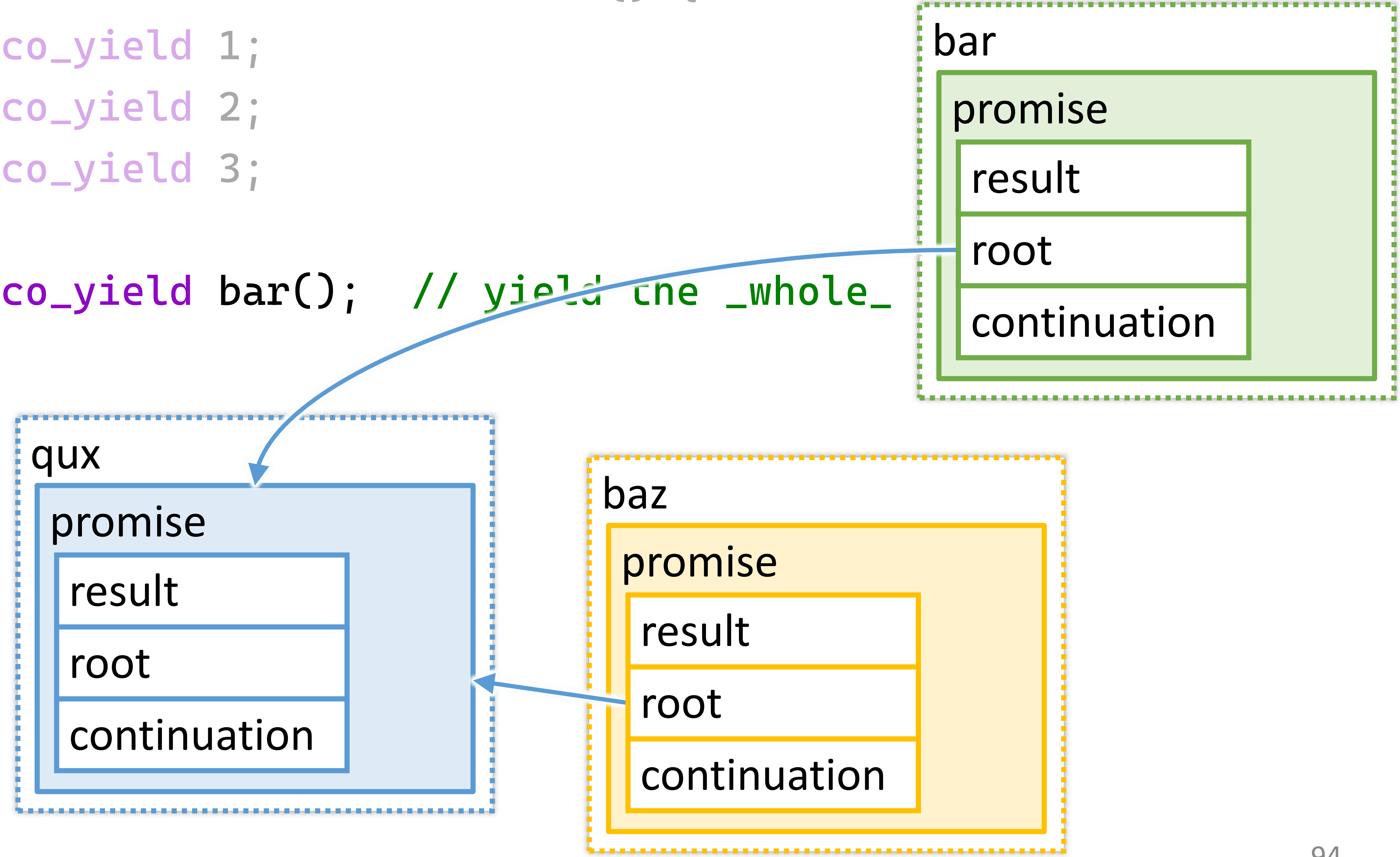
```
RecursiveGenerator<int> bar() {
  for (auto n : getValues())
    co_yield n;
}
```



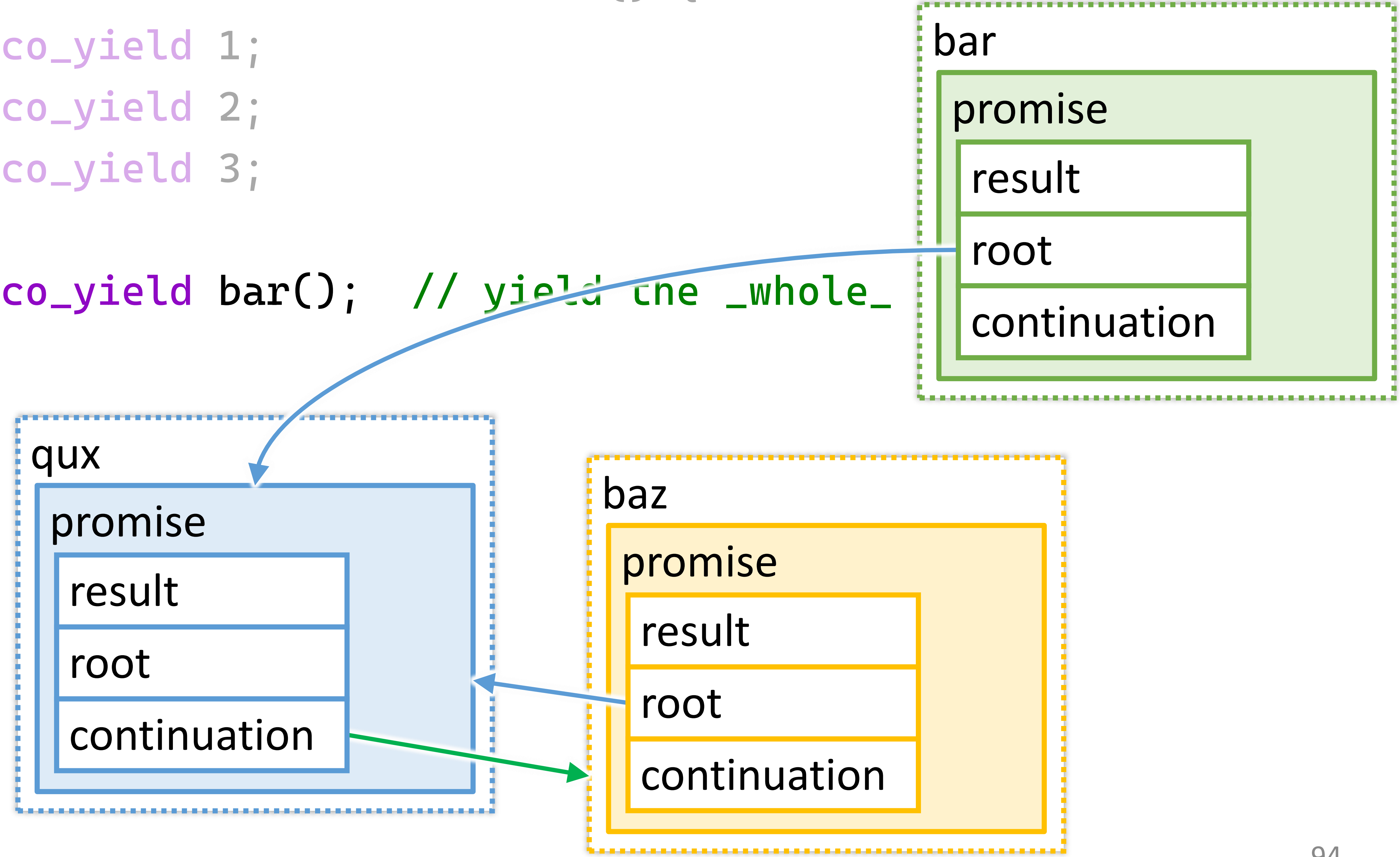
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_
}
```



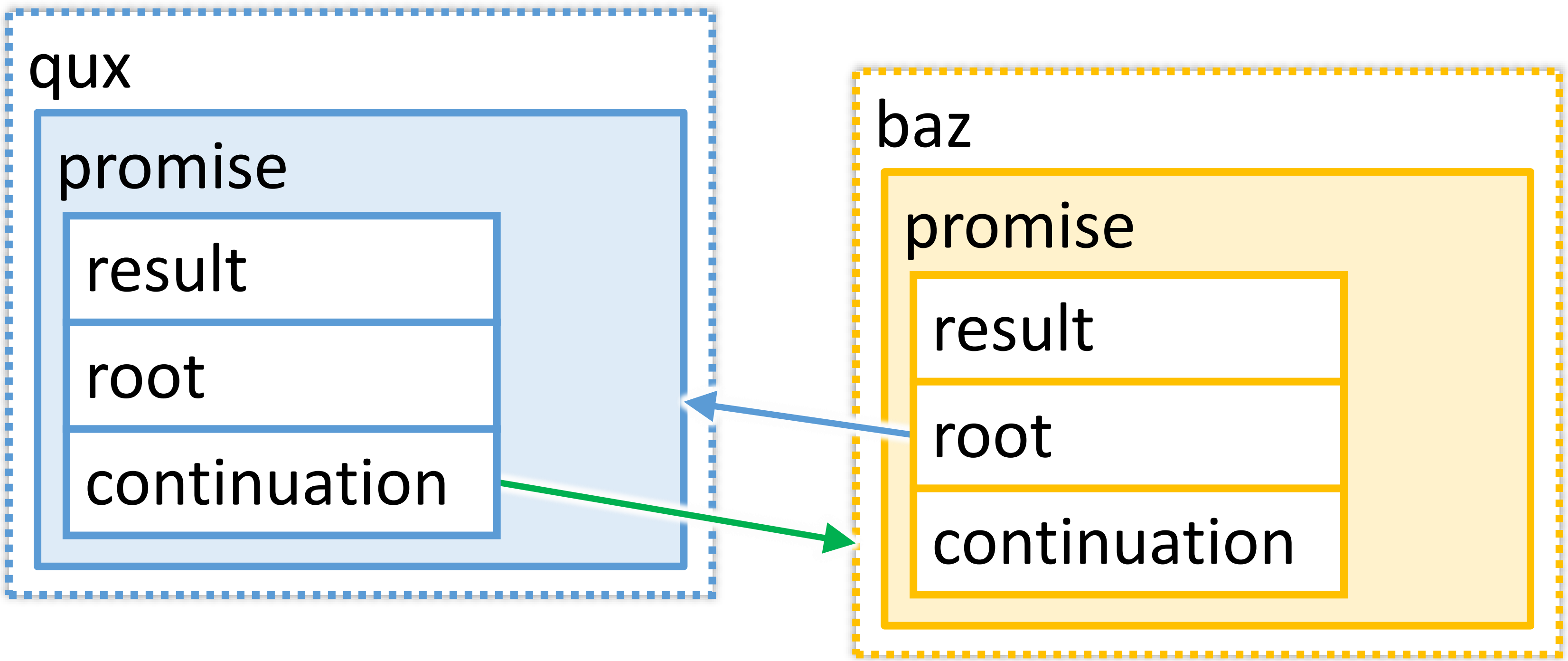
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_
}
```



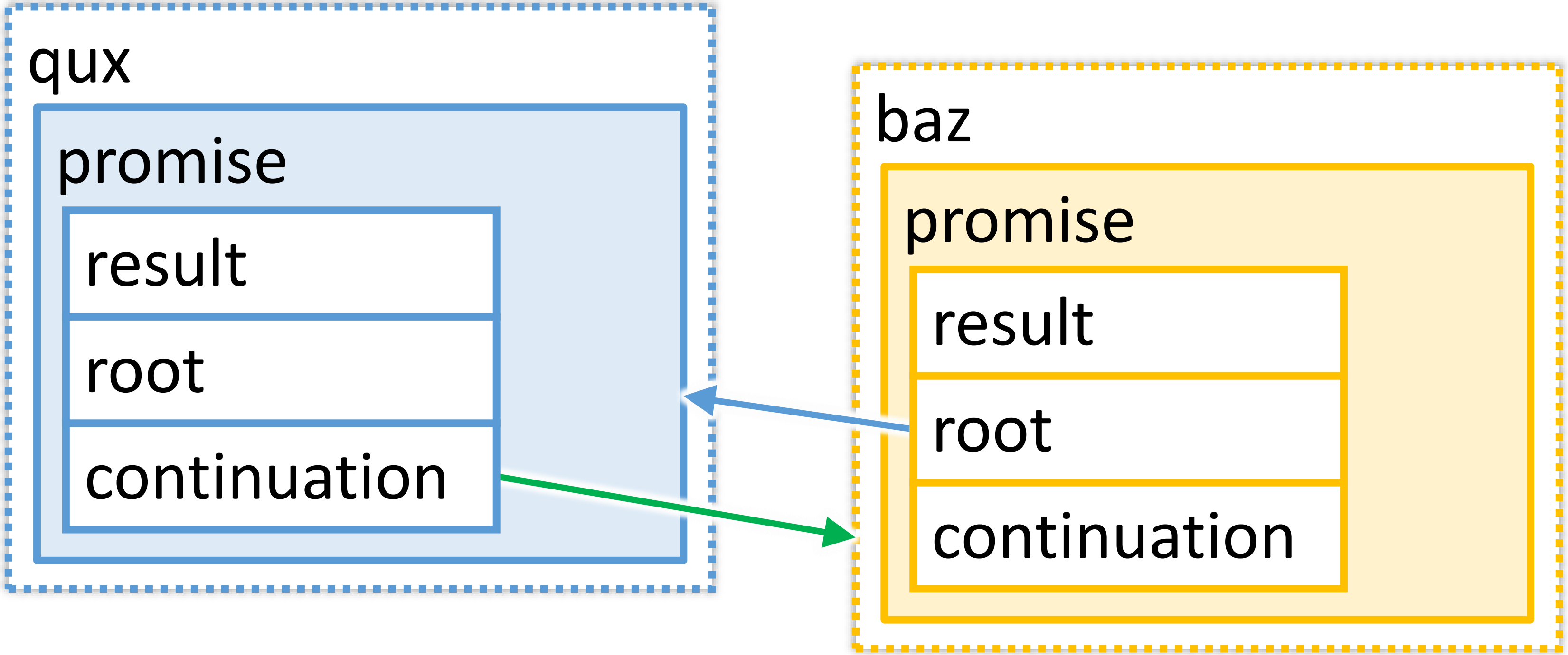
```cpp
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_
}
```



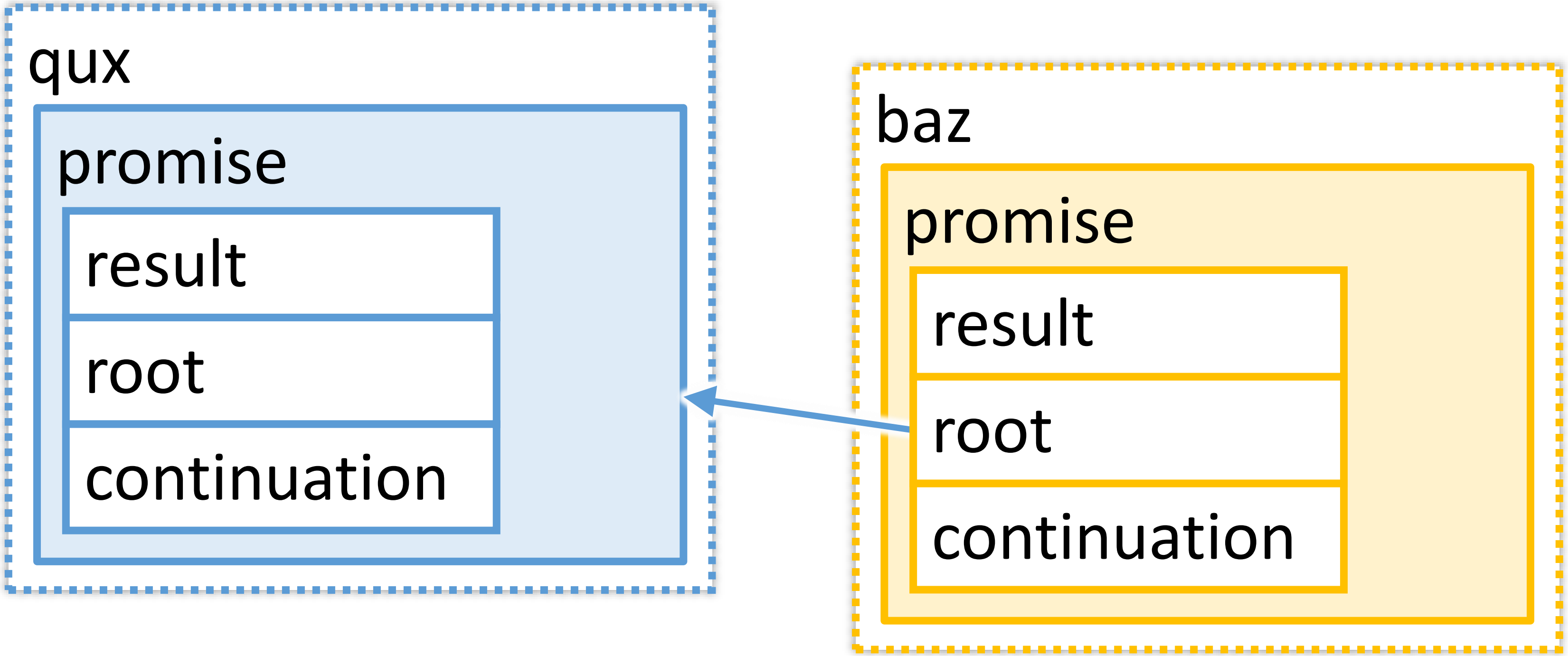
```
RecursiveGenerator<int> baz() {
  co_yield 1;
  co_yield 2;
  co_yield 3;

  co_yield bar();  // yield the _whole_ thing
}
```



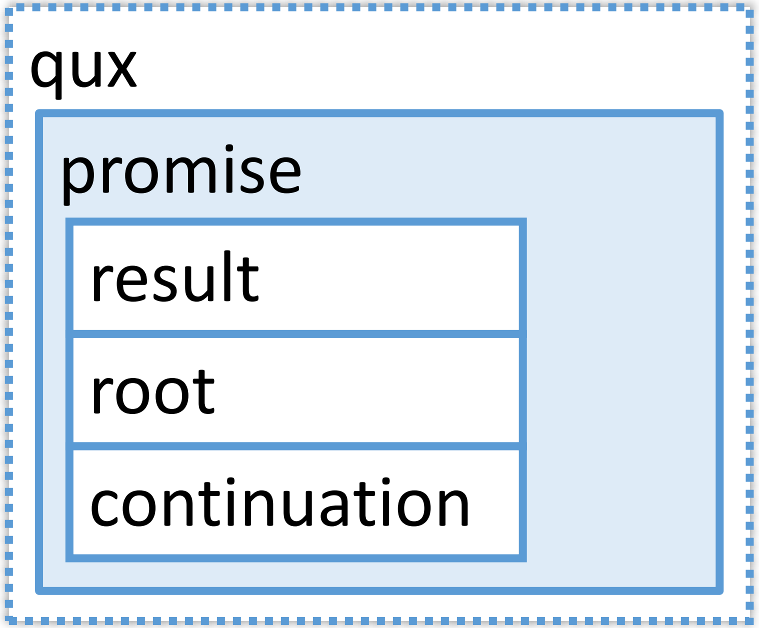
```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



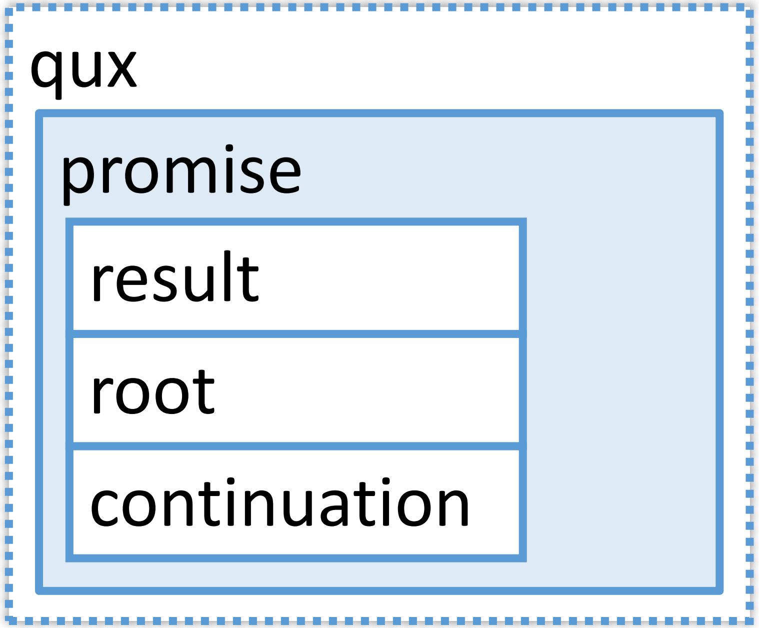
```
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



```cpp
RecursiveGenerator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  co_yield g;  // yield the rest
}
```



```
const auto h = qux();
for (auto &i : h)
  std::cout << i << '\n';
```

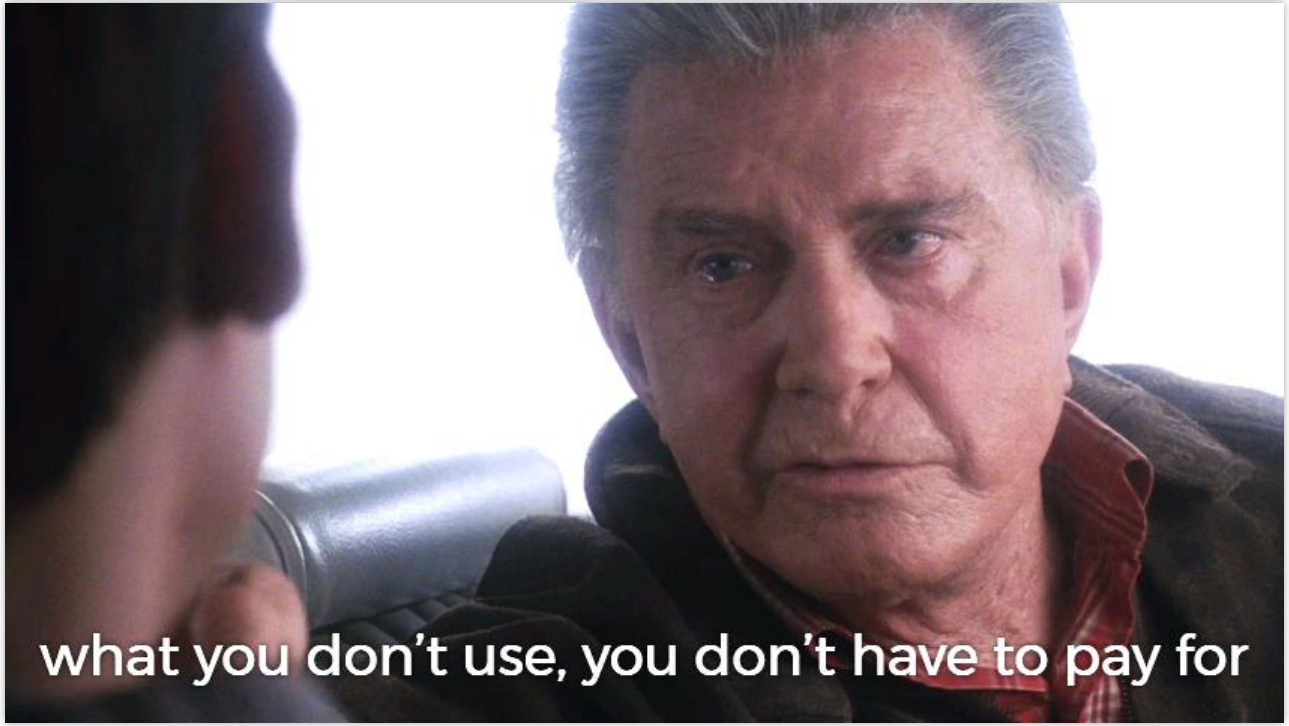


Recursive generator

In this presentation `Generator` and `RecursiveGenerator` are different because the latter needs additional fields to track nested-ness, which is an overhead (though a reasonably small one).

Recursive generator

In this presentation `Generator` and `RecursiveGenerator` are different because the latter needs additional fields to track nested-ness, which is an overhead (though a reasonably small one).



`std::generator`

- **P2502**: `std::generator`: Synchronous Coroutine Generator for Ranges by Casey Carter http://wg21.link/p2502
- accepted into C++23
- works very much like `RecursiveGenerator`

```
namespace std {
  template<class R, class V = void, class Allocator = void>
  class generator {
  public:
    using yielded =
      conditional_t<is_reference_v<reference>, reference, const reference&>;
    class promise_type;

    generator(const generator&) = delete;
    generator(generator&&) noexcept;
    ~generator();
    generator &operator=(const generator&) = delete;
    generator &operator=(generator&&) noexcept;
    iterator begin();
    default_sentinel_t end() const noexcept;
  };

  template<class R, class V, class Allocator>
  constexpr bool ranges::enable_view<generator<R, V, Allocator>> = true;
}
```

std::generator

Members [generator.members]

```
iterator begin();
```

Preconditions: *`coroutine_`* refers to a coroutine suspended at its initial suspend-point.

Effects: Equivalent to:

```
coroutine_.resume();
return iterator(coroutine_);
```

Remarks: This function pushes *`coroutine_`* onto the `generator`'s empty stack of associated coroutines.

[*Note:* A program that calls `begin` more than once on the same generator has undefined behavior. — *end note*]

std::generator

Members [generator.members]

`iterator begin();`

Preconditions: *`coroutine_`* refers to a coroutine suspended at its initial suspend-point.

Effects: Equivalent to:

```
coroutine_.resume();
return iterator(coroutine_);
```

Remarks: This function pushes *`coroutine_`* onto the `generator`'s empty stack of associated coroutines.

[*Note:* A program that calls `begin` more than once on the same generator has undefined behavior. — *end note*]

std::generator

```
std::generator<int> qux() {
  const auto g = baz();
  if (auto i = g.begin(); i != g.end()) {
    co_yield *i * 33;
    ++i;
  }
  for (auto i : g)  // UB: 'g.begin()' is called
    co_yield i;  // yield the rest
}
```

changes observable state
(that we can't observe)

std::generator

```
std::generator<int> qux() {
  const auto g = baz();
  for (auto &i : g | std::views::take(1))
    co_yield i * 33;


  for (auto i : g)  // UB: 'g.begin()' is called
    co_yield i;  // yield the rest
}
```

std::generator

```
std::generator<int> qux() {
  const auto g = baz();
  for (auto &i : g | std::views::take(1))
    co_yield i * 33;


  co_yield std::ranges::elements_of{ g };  // UB?
}
```

std::generator

```
std::generator<int> qux() {
  auto g = baz();
  for (auto &i : g | std::views::take(1))
    co_yield i * 33;

  // pointless: calling 'begin()' is UB
  processValues(std::move(g));
}
```

std::generator

```
std::istringstream s{ "hello world ..." };
for (auto i = std::istream_iterator<std::string>{ s };
     i != std::istream_iterator<std::string>{};
     ++i) {
  std::cout << *i << '\n';
  //...
  if (someCondition)
    break;
}
for (auto i = std::istream_iterator<std::string>{ s };
     i != std::istream_iterator<std::string>{};
     ++i) {
  // do something else with the rest of the data
}
```

std::generator

```
std::istringstream s{ "hello world ..." };
for (auto i = std::istream_iterator<std::string>{ s };
      i != std::istream_iterator<std::string>{};
      ++i) {
  std::cout << *i << '\n';
  //...
  if (someCondition)
    break;
}
for (auto i = std::istream_iterator<std::string>{ s };
      i != std::istream_iterator<std::string>{};
      ++i) {
  // do something else with the rest of the data
}
```

`istream_iterator`'s ctor reads from `s`, i.e. changes its observable state

iteration can be safely restarted/continued

`std::generator`

- generator type in the standard library

  Yay!

- recursive — always has (reasonably small) overhead when you don't yield nested generators

- can't restart/continue iteration after `begin()` is already called once

`std::generator`

- generator type in the standard library

  Yay!

- recursive — always has (reasonably small) overhead when you don't yield nested generators

  whatever…

- can't restart/continue iteration after `begin()` is already called once

`std::generator`

- generator type in the standard library

  Yay!

- recursive — always has (reasonably small) overhead when you don't yield nested generators

  whatever…

- can't restart/continue iteration after `begin()` is already called once

  (╯°□°）╯︵ ┻━┻

Async generator

```
Task<std::vector<int>> getValuesAsync();

AsyncGenerator<int> generateValuesAsync() {
  const auto values = co_await getValuesAsync();
  for (auto &v : values) {
    if (isValueValid(v))
      co_yield v;
  }
}
```

Async generator

```
Task<std::vector<int>> getValuesAsync();

AsyncGenerator<int> generateValuesAsync() {
  const auto values = co_await getValuesAsync();
  for (auto &v : values) {
    if (isValueValid(v))
      co_yield v;
  }
}
```

Async generator

```
Task<std::vector<int>> getValuesAsync();

AsyncGenerator<int> generateValuesAsync() {
  const auto values = co_await getValuesAsync();
  for (auto &v : values) {
    if (isValueValid(v))
      co_yield v;
  }
}
```

Async generator

```
Task<std::vector<int>> getValuesAsync();

AsyncGenerator<int> generateValuesAsync() {
  const auto values = co_await getValuesAsync();
  for (auto &v : values) {
    if (isValueValid(v))
      co_yield v;
  }
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  for (auto i = g.begin();
       i != g.end();
       i = co_await i.next()) {
    //...
  }
  //...
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  for (auto i = g.begin();
       i != g.end();
       i = co_await i.next()) {
    //...
  }
  //...
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  int prettiest = -37;
  int prettinessLevel = 0;

  while (auto next = co_await g.next()) {
    const auto p = getPrettinessLevel(*next);
    if (prettinessLevel < p ||
        prettinessLevel == p && prettiest < *next) {
      prettiest = *next;
      prettinessLevel = p;
    }
  }

  co_return prettiest;
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  int prettiest = -37;
  int prettinessLevel = 0;

  while (auto next = co_await g.next()) {
    const auto p = getPrettinessLevel(*next);
    if (prettinessLevel < p ||
        prettinessLevel == p && prettiest < *next) {
      prettiest = *next;
      prettinessLevel = p;
    }
  }

  co_return prettiest;
}
```

Async generator

```
try {
  const auto value = syncWait(getPrettiestValue());
  std::cout << value << '\n';
}
catch (const std::exception &e) {
  std::cout << "exception: " << e.what() << '\n';
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  while (auto next = co_await g.next()) {
    //...
  }
  //...
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  while (auto next = co_await g.next()) {
    //...
  }
  //...
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  while (auto next = co_await g.next()) {
    //...
  }
  //...
}
```

this coroutine is suspended and set as continuation for g

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  while (auto next = co_await g.next()) {
    //...
  }
  //...
}
```

Async generator

```
Task<int> getPrettiestValue() {
   auto g = generateValuesAsync();
   //...
   while (auto next = co_await g.next()) {
      //...
   }
   //...
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  while (auto next = co_await g.next()) {
    //...
  }
  //...
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  while (auto next = co_await g.next()) {
    //...
  }
  //...
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  while (auto next = co_await g.next()) {
    //...
  }
  //...
}
```

this coroutine is resumed
as a continuation of g
and optional result is returned

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  while (auto next = co_await g.next()) {
    //...
  }
  //...
}
```

this coroutine is resumed
as a continuation of g
and optional result is returned

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  //...
  while (auto next = co_await g.next()) {
    //...
  }
  //...
}
```

this coroutine is resumed
as a continuation of g
and optional result is returned

Async generator

What lies ahead

is separated by an even thinner veil

from nonsense.

Async generator

```
template<typename T>
struct AsyncGenerator {
  struct promise_type;

  AsyncGenerator(AsyncGenerator &&other) noexcept;
  AsyncGenerator &operator=(AsyncGenerator &&other) noexcept;
  ~AsyncGenerator();

  auto next();

private:
  explicit AsyncGenerator(promise_type &promise) noexcept;

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

Async generator

```
template<typename T>
struct AsyncGenerator {
  struct promise_type;

  AsyncGenerator(AsyncGenerator &&other) noexcept;
  AsyncGenerator &operator=(AsyncGenerator &&other) noexcept;
  ~AsyncGenerator();

  auto next();

private:
  explicit AsyncGenerator(promise_type &promise) noexcept;

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

Async generator

```
template<typename T>
struct AsyncGenerator {
  //...
  AsyncGenerator(AsyncGenerator &&other) noexcept :
    coro{ std::exchange(other.coro, nullptr) }
  {}
  AsyncGenerator &operator=(AsyncGenerator &&other) noexcept {
    if (coro)
      coro.destroy();
    coro = std::exchange(other.coro, nullptr);
  }
  ~AsyncGenerator() {
    if (coro)
      coro.destroy();
  }
  //...
};
```

Async generator

```
template<typename T>
struct AsyncGenerator {
  //...
  AsyncGenerator(AsyncGenerator &&other) noexcept :
    coro{ std::exchange(other.coro, nullptr) }
  {}
  AsyncGenerator &operator=(AsyncGenerator &&other) noexcept {
    if (coro)
      coro.destroy();
    coro = std::exchange(other.coro, nullptr);
  }
  ~AsyncGenerator() {
    if (coro)
      coro.destroy();
  }
  //...
};
```

Async generator

```
template<typename T>
struct AsyncGenerator {
  //...
  AsyncGenerator(AsyncGenerator &&other) noexcept :
    coro{ std::exchange(other.coro, nullptr) }
  {}
  AsyncGenerator &operator=(AsyncGenerator &&other) noexcept {
    if (coro)
      coro.destroy();
    coro = std::exchange(other.coro, nullptr);
  }
  ~AsyncGenerator() {
    if (coro)
      coro.destroy();
  }
  //...
};
```

Async generator

```
template<typename T>
struct AsyncGenerator {
  //...
  auto next() {
    return typename promise_type::NextAwaitable{ coro };
  }

private:
  explicit AsyncGenerator(promise_type &promise) noexcept :
    coro{ std::coroutine_handle<promise_type>::from_promise(promise) }
  {}

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

Async generator

```
template<typename T>
struct AsyncGenerator {
  //...
  auto next() {
    return typename promise_type::NextAwaitable{ coro };
  }

private:
  explicit AsyncGenerator(promise_type &promise) noexcept :
    coro{ std::coroutine_handle<promise_type>::from_promise(promise) }
  {}

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

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  auto final_suspend() const noexcept;

  struct YieldAwaitable;
  YieldAwaitable yield_value(T &&value) noexcept;
  YieldAwaitable yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}
  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();
  bool hasException() const noexcept;
  void throwIfException() const;

  struct NextAwaitable;

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  auto final_suspend() const noexcept;

  struct YieldAwaitable;
  YieldAwaitable yield_value(T &&value) noexcept;
  YieldAwaitable yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}
  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();
  bool hasException() const noexcept;
  void throwIfException() const;

  struct NextAwaitable;

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  auto final_suspend() const noexcept;

  struct YieldAwaitable;
  YieldAwaitable yield_value(T &&value) noexcept;
  YieldAwaitable yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}
  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();
  bool hasException() const noexcept;
  void throwIfException() const;

  struct NextAwaitable;

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  auto final_suspend() const noexcept;

  struct YieldAwaitable;
  YieldAwaitable yield_value(T &&value) noexcept;
  YieldAwaitable yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}
  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();
  bool hasException() const noexcept;
  void throwIfException() const;

  struct NextAwaitable;

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

```
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  auto final_suspend() const noexcept;

  struct YieldAwaitable;
  YieldAwaitable yield_value(T &&value) noexcept;
  YieldAwaitable yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}
  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();
  bool hasException() const noexcept;
  void throwIfException() const;

  struct NextAwaitable;

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

```cpp
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  auto final_suspend() const noexcept;

  struct YieldAwaitable;
  YieldAwaitable yield_value(T &&value) noexcept;
  YieldAwaitable yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}
  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();
  bool hasException() const noexcept;
  void throwIfException() const;

  struct NextAwaitable;

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

Async generator

```
struct NextAwaitable {
  NextAwaitable(std::coroutine_handle<promise_type> coro) noexcept :
    coro{ coro } {}
  bool await_ready() const noexcept {
    return false;
  }
  auto await_suspend(std::coroutine_handle<> thatCoro) const noexcept {
    coro.promise().continuation = thatCoro;
    return coro;
  }
  std::optional<T> await_resume() const {
    //...
  }
private:
  std::coroutine_handle<promise_type> coro;
};
```

Async generator

```
struct NextAwaitable {
  NextAwaitable(std::coroutine_handle<promise_type> coro) noexcept :
    coro{ coro } {}
  bool await_ready() const noexcept {
    return false;
  }
  auto await_suspend(std::coroutine_handle<> thatCoro) const noexcept {
    coro.promise().continuation = thatCoro;
    return coro;
  }
  std::optional<T> await_resume() const {
    //...
  }
private:
  std::coroutine_handle<promise_type> coro;
};
```

handle of generator's coroutine

Async generator

```
struct NextAwaitable {
  NextAwaitable(std::coroutine_handle<promise_type> coro) noexcept :
    coro{ coro } {}
  bool await_ready() const noexcept {
    return false;
  }
  auto await_suspend(std::coroutine_handle<> thatCoro) const noexcept {
    coro.promise().continuation = thatCoro;
    return coro;
  }
  std::optional<T> await_resume() const {
    //...
  }
private:
  std::coroutine_handle<promise_type> coro;
};
```

Async generator

```
struct NextAwaitable {
  NextAwaitable(std::coroutine_handle<promise_type> coro) noexcept :
    coro{ coro } {}
  bool await_ready() const noexcept {
    return false;
  }
  auto await_suspend(std::coroutine_handle<> thatCoro) const noexcept {
    coro.promise().continuation = thatCoro;
    return coro;
  }
  std::optional<T> await_resume() const {
    //...
  }
private:
  std::coroutine_handle<promise_type> coro;
};
```

symmetric transfer of control

Async generator

```
struct NextAwaitable {
  //...
  std::optional<T> await_resume() const {
    auto &promise = coro.promise();
    if (coro.done()) {
      promise.throwIfException();
      return {};
    }
    return std::move(promise.getValue());
  }

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

Async generator

```
struct NextAwaitable {
  //...
  std::optional<T> await_resume() const {
    auto &promise = coro.promise();
    if (coro.done()) {
      promise.throwIfException();
      return {};
    }
    return std::move(promise.getValue());
  }

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

Async generator

```
struct NextAwaitable {
  //...
  std::optional<T> await_resume() const {
    auto &promise = coro.promise();
    if (coro.done()) {
      promise.throwIfException();
      return {};
    }
    return std::move(promise.getValue());
  }

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

Async generator

```
struct NextAwaitable {
  //...
  std::optional<T> await_resume() const {
    auto &promise = coro.promise();
    if (coro.done()) {
      promise.throwIfException();
      return {};
    }
    return std::move(promise.getValue());
  }

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

```cpp
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  auto final_suspend() const noexcept;

  struct YieldAwaitable;
  YieldAwaitable yield_value(T &&value) noexcept;
  YieldAwaitable yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}
  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();
  bool hasException() const noexcept;
  void throwIfException() const;

  struct NextAwaitable;

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

```cpp
struct promise_type {
  auto get_return_object() noexcept;
  std::suspend_always initial_suspend() const noexcept;
  auto final_suspend() const noexcept;

  struct YieldAwaitable;
  YieldAwaitable yield_value(T &&value) noexcept;
  YieldAwaitable yield_value(const T &value) noexcept(std::is_nothrow_copy_constructible_v<T>);

  void return_void() const noexcept {}
  void unhandled_exception() noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>);

  T &getValue();
  bool hasException() const noexcept;
  void throwIfException() const;

  struct NextAwaitable;

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

Async generator

```
struct promise_type {
  auto get_return_object() noexcept {
    return AsyncGenerator{ *this };
  }
  std::suspend_always initial_suspend() const noexcept {
    return {};
  }
  //...
};
```

Async generator

```
struct promise_type {
  auto get_return_object() noexcept {
    return AsyncGenerator{ *this };
  }
  std::suspend_always initial_suspend() const noexcept {
    return {};
  }
  //...
};
```

Async generator

```
struct promise_type {
  auto get_return_object() noexcept {
    return AsyncGenerator{ *this };
  }
  std::suspend_always initial_suspend() const noexcept {
    return {};
  }
  //...
};
```

Async generator

```
struct promise_type {
  //...
  auto final_suspend() const noexcept {
    struct FinalAwaitable {
      //...
    };
    return FinalAwaitable{};
  }
  //...
};
```

Async generator

```
struct FinalAwaitable {
  bool await_ready() const noexcept {
    return false;
  }
  std::coroutine_handle<>
    await_suspend(std::coroutine_handle<promise_type> thisCoro)
      noexcept {
    auto &promise = thisCoro.promise();

    assert(promise.continuation);

    return promise.continuation;
  }
  void await_resume() const noexcept {}
};
```

Async generator

```
struct FinalAwaitable {
  bool await_ready() const noexcept {
    return false;
  }
  std::coroutine_handle<>
    await_suspend(std::coroutine_handle<promise_type> thisCoro)
      noexcept {
    auto &promise = thisCoro.promise();

    assert(promise.continuation);

    return promise.continuation;
  }
  void await_resume() const noexcept {}
};
```

Async generator

```
struct FinalAwaitable {
  bool await_ready() const noexcept {
    return false;
  }
  std::coroutine_handle<>
    await_suspend(std::coroutine_handle<promise_type> thisCoro)
      noexcept {
    auto &promise = thisCoro.promise();

    assert(promise.continuation);

    return promise.continuation;
  }
  void await_resume() const noexcept {}
};
```

Async generator

```
struct FinalAwaitable {
  bool await_ready() const noexcept {
    return false;
  }
  std::coroutine_handle<>
    await_suspend(std::coroutine_handle<promise_type> thisCoro)
      noexcept {
    auto &promise = thisCoro.promise();

    assert(promise.continuation);

    return promise.continuation;
  }
  void await_resume() const noexcept {}
};
```

Async generator

```
struct FinalAwaitable {
  bool await_ready() const noexcept {
    return false;
  }
  std::coroutine_handle<>
    await_suspend(std::coroutine_handle<promise_type> thisCoro)
      noexcept {
    auto &promise = thisCoro.promise();

    assert(promise.continuation);

    return promise.continuation;
  }
  void await_resume() const noexcept {}
};
```

symmetric transfer of control

Async generator

```
struct FinalAwaitable {
  bool await_ready() const noexcept {
    return false;
  }
  std::coroutine_handle<>
    await_suspend(std::coroutine_handle<promise_type> thisCoro)
      noexcept {
    auto &promise = thisCoro.promise();

    assert(promise.continuation);

    return promise.continuation;
  }
  void await_resume() const noexcept {}
};
```

```cpp
struct promise_type {
  //...
  struct YieldAwaitable {
    //...
  };

  YieldAwaitable yield_value(T &&value) noexcept {
    result = std::addressof(value);
    return {};
  }
  YieldAwaitable yield_value(const T &value)
    noexcept(std::is_nothrow_copy_constructible_v<T>) {
    result = value;
    return {};
  }
  //...
};
```

```cpp
struct promise_type {
  //...
  struct YieldAwaitable {
    //...
  };

  YieldAwaitable yield_value(T &&value) noexcept {
    result = std::addressof(value);
    return {};
  }
  YieldAwaitable yield_value(const T &value)
    noexcept(std::is_nothrow_copy_constructible_v<T>) {
    result = value;
    return {};
  }
  //...
};
```

```
struct promise_type {
  //...
  struct YieldAwaitable {
    //...
  };

  YieldAwaitable yield_value(T &&value) noexcept {
    result = std::addressof(value);
    return {};
  }
  YieldAwaitable yield_value(const T &value)
    noexcept(std::is_nothrow_copy_constructible_v<T>) {
    result = value;
    return {};
  }
  //...
};
```

Async generator

```
struct YieldAwaitable {
  bool await_ready() const noexcept {
    return false;
  }
  std::coroutine_handle<>
    await_suspend(std::coroutine_handle<promise_type> thisCoro)
      const noexcept {
    auto &promise = thisCoro.promise();

    assert(promise.continuation);

    return std::exchange(promise.continuation, nullptr);
  }
  void await_resume() const noexcept {}
};
```

Async generator

```
struct YieldAwaitable {
  bool await_ready() const noexcept {
    return false;
  }
  std::coroutine_handle<>
    await_suspend(std::coroutine_handle<promise_type> thisCoro)
      const noexcept {
    auto &promise = thisCoro.promise();

    assert(promise.continuation);

    return std::exchange(promise.continuation, nullptr);
  }
  void await_resume() const noexcept {}
};
```

Async generator

```
struct YieldAwaitable {
  bool await_ready() const noexcept {
    return false;
  }
  std::coroutine_handle<>
    await_suspend(std::coroutine_handle<promise_type> thisCoro)
      const noexcept {
    auto &promise = thisCoro.promise();

    assert(promise.continuation);

    return std::exchange(promise.continuation, nullptr);
  }
  void await_resume() const noexcept {}
};
```

symmetric transfer of control

Async generator

```
struct YieldAwaitable {
  bool await_ready() const noexcept {
    return false;
  }
  std::coroutine_handle<>
    await_suspend(std::coroutine_handle<promise_type> thisCoro)
      const noexcept {
    auto &promise = thisCoro.promise();

    assert(promise.continuation);

    return std::exchange(promise.continuation, nullptr);
  }
  void await_resume() const noexcept {}
};
```

Async generator

```
AsyncGenerator<int> generateValuesAsync() {
  const auto values = co_await getValuesAsync();
  for (auto &v : values) {
    if (isValueValid(v))
      co_yield v;
  }
}
```

Async generator

```
AsyncGenerator<int> generateValuesAsync() {
  const auto values = co_await getValuesAsync();
  for (auto &v : values) {
    if (isValueValid(v))
      co_yield v;
  }
}
```

Async generator

```
AsyncGenerator<int> generateValuesAsync() {
  const auto values = co_await getValuesAsync();
  for (auto &v : values) {
    if (isValueValid(v))
      co_yield v;
  }
}
```

suspend

value is yielded,
this coroutine is suspended
and continuation is resumed
via symmetric transfer of control

Async generator

```
struct promise_type {
  //...
  void return_void() const noexcept {}

  void unhandled_exception()
    noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>) {
    result = std::current_exception();
  }
  //...
};
```

Async generator

```
struct promise_type {
  //...
  void return_void() const noexcept {}

  void unhandled_exception()
    noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>) {
    result = std::current_exception();
  }
  //...
};
```

Async generator

```
struct promise_type {
  //...
  void return_void() const noexcept {}

  void unhandled_exception()
    noexcept(std::is_nothrow_copy_constructible_v<std::exception_ptr>) {
    result = std::current_exception();
  }
  //...
};
```

Async generator

```
struct promise_type {
  //...
  T &getValue() {
    return std::holds_alternative<T>(result) ? std::get<T>(result) :
                                               *std::get<T*>(result);
  }
  bool hasException() const noexcept {
    return std::holds_alternative<std::exception_ptr>(result);
  }
  void throwIfException() const {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));
  }
  //...
};
```

Async generator

```
struct promise_type {
  //...
  T &getValue() {
    return std::holds_alternative<T>(result) ? std::get<T>(result) :
                                               *std::get<T*>(result);
  }
  bool hasException() const noexcept {
    return std::holds_alternative<std::exception_ptr>(result);
  }
  void throwIfException() const {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));
  }
  //...
};
```

Async generator

```
struct promise_type {
  //...
  T &getValue() {
    return std::holds_alternative<T>(result) ? std::get<T>(result) :
                                               *std::get<T*>(result);
  }
  bool hasException() const noexcept {
    return std::holds_alternative<std::exception_ptr>(result);
  }
  void throwIfException() const {
    if (hasException())
      std::rethrow_exception(std::get<std::exception_ptr>(result));
  }
  //...
};
```

Async generator

```
struct promise_type {
  //...
  struct NextAwaitable {
    //...
  };

private:
  std::variant<std::monostate, T, T*, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

Async generator

```
Task<std::vector<int>> getValuesAsync();

AsyncGenerator<int> generateValuesAsync() {
  const auto values = co_await getValuesAsync();
  for (auto &v : values) {
    if (isValueValid(v))
      co_yield v;
  }
}
```

Async generator

```
Task<std::vector<int>> getValuesAsync();

AsyncGenerator<int> generateValuesAsync() {
  const auto values = co_await getValuesAsync();
  for (auto &v : values) {
    if (isValueValid(v))
      co_yield v;
  }
}
```

Async generator

```
Task<std::vector<int>> getValuesAsync();

AsyncGenerator<int> generateValuesAsync() {
  const auto values = co_await getValuesAsync();
  for (auto &v : values) {
    if (isValueValid(v))
      co_yield v;
  }
}
```

Async generator

```
Task<int> getPrettiestValue() {
  auto g = generateValuesAsync();
  int prettiest = -37;
  int prettinessLevel = 0;

  while (auto next = co_await g.next()) {
    const auto p = getPrettinessLevel(*next);
    if (prettinessLevel < p ||
        prettinessLevel == p && prettiest < *next) {
      prettiest = *next;
      prettinessLevel = p;
    }
  }

  co_return prettiest;
}
```

https://youtu.be/OE45F3iKtv4

Thanks for listening!

Understanding C++ coroutines by example part 2: generators

Pavel Novikov

@cpp_ape

Thanks to Phil Nash for feedback.

Slides: https://bit.ly/3Q716se

References

- Lewis Baker "Structured Concurrency: Writing safer concurrent code with coroutines and algorithms" https://youtu.be/1Wy5sq3s2rg
- P2502: `std::generator`: Synchronous Coroutine Generator for Ranges http://wg21.link/p2502
- Павел Новиков "Учимся готовить C++ корутины на практике" (часть 1) https://youtu.be/OE45F3iKtv4

Bonus slides

Ugly simple generator

```
template<typename T>
struct Generator {
  struct promise_type;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  bool hasValue() const noexcept;  //has value or exception
  auto &operator()() const;

private:
  explicit Generator(promise_type &promise) noexcept;

  void getNextValue() const noexcept;

  std::coroutine_handle<promise_type> coro;
  mutable bool gotValue = false;
};
```

Ugly simple generator

```
template<typename T>
struct Generator {
  //...
private:
  //...
  void getNextValue() const noexcept {
    if (!gotValue && !coro.done()) {
      coro();
      gotValue = !coro.done() || coro.promise().hasException();
    }
  }

  std::coroutine_handle<promise_type> coro;
  mutable bool gotValue = false;
};
```

Ugly simple generator

```
template<typename T>
struct Generator {
  //...
  bool hasValue() const noexcept {  //has value or exception
    getNextValue();
    return gotValue;
  }

  auto &operator()() const {
    getNextValue();
    gotValue = false;
    return coro.promise().getValue();
  }
  //...
};
```

precondition:

```
  hasValue() == true
```

or, more precisely:

```
  !coro.done() or
  coro.promise().hasException()
```

Range generator with lazy iterator

```
template<typename T>
struct Generator {
  struct promise_type;
  struct LazyIterator;

  Generator(Generator &&other) noexcept;
  Generator &operator=(Generator &&other) noexcept;
  ~Generator();

  LazyIterator begin() const noexcept;
  LazyIterator end() const noexcept;

private:
  explicit Generator(promise_type &promise) noexcept;

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

```cpp
struct LazyIterator {
  // iterator boilerplate

  LazyIterator() noexcept = default;
  explicit LazyIterator(const std::coroutine_handle<promise_type> &coro) noexcept;

  friend bool operator==(const LazyIterator&, const LazyIterator&) noexcept = default;
  friend bool operator!=(const LazyIterator&, const LazyIterator&) noexcept = default;

  LazyIterator &operator++() noexcept;
  auto &operator*() const;
  friend bool hasException(const LazyIterator &i) noexcept;

private:
  const std::coroutine_handle<promise_type> *coro = nullptr;
};
```

Range generator with lazy iterator

```
struct LazyIterator {
  //...
  LazyIterator &operator++() noexcept {
    assert(coro != nullptr);
    assert(!coro->done());

    coro->resume();
    if (coro->done() && !coro->promise().hasException())
      coro = nullptr;
    return *this;
  }
  //...
};
```

Range generator with lazy iterator

```
struct LazyIterator {
  //...
  auto &operator*() const {
    assert(coro != nullptr);
    coro->promise().throwIfException();
    return coro->promise().getValue();
  }

  friend bool hasException(const LazyIterator &i) noexcept {
    return i.coro && i.coro->promise().hasException();
  }
  //...
};
```

Range generator with lazy iterator

```
LazyIterator begin() const noexcept {
  if (coro.done())
    return end();

  auto i = LazyIterator{ coro };
  if (!coro.promise().isValueInitialized())
    ++i;
  return i;
}

LazyIterator end() const noexcept {
  return {};
}
```