How C++20 changes the way we write code

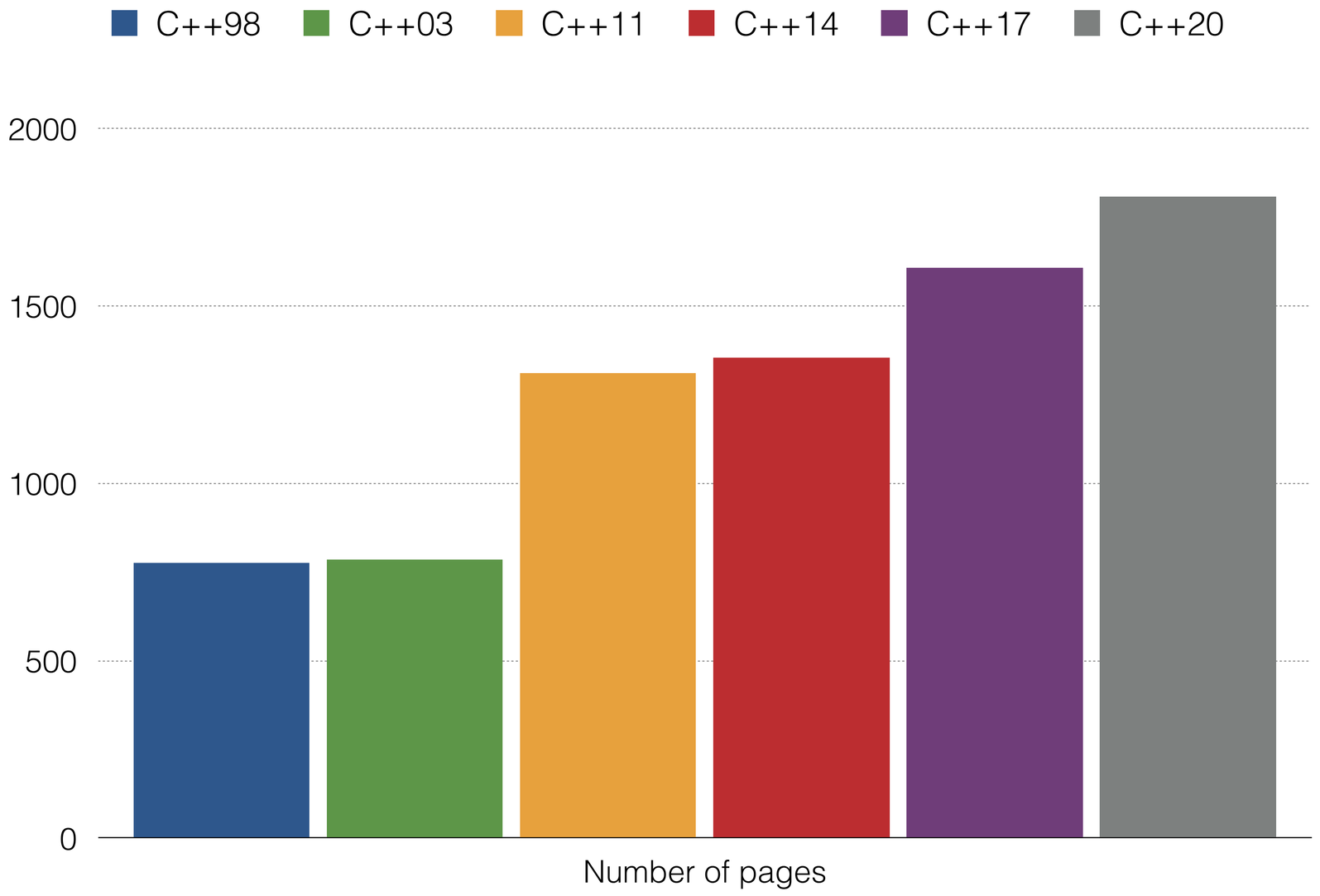
Timur Doumler

@timur_audio

CppCon
18 September 2020


ISO C++ committee meeting
Prague, February 2020

C++20 is done 🎉


C++98
C++03
C++11
C++14
C++17
C++20
2000
1500
1000
500
0
Number of pages

```
std::vector<std::vector<int>> vec = {{1, 0}, {0, 1}};
```

```
std::vector<std::vector<int>> vec = {{1, 0}, {0, 1}};

for (std::vector<std::vector<int>>::iterator i = vec.begin(); i != vec.end(); ++i)
    for (std::vector<int>::const_iterator j = i->begin(); j != i->end(); ++j)
        /* do something */
```

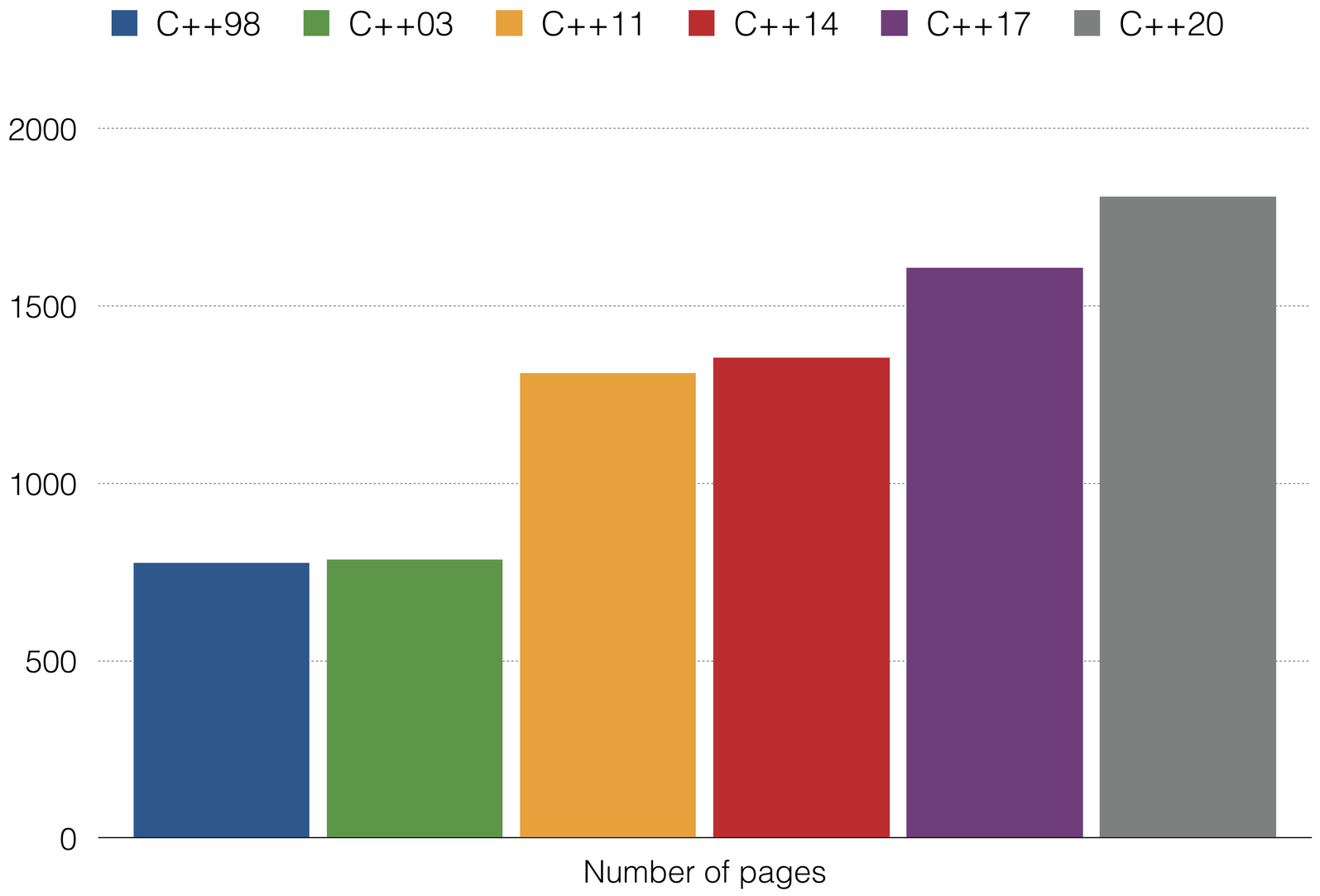
```
std::vector<std::vector<int>> vec = {{1, 0}, {0, 1}};

for (std::vector<std::vector<int>>::iterator i = vec.begin(); i != vec.end(); ++i)
    for (std::vector<int>::const_iterator j = i->begin(); j != i->end(); ++j)
        /* do something */
```

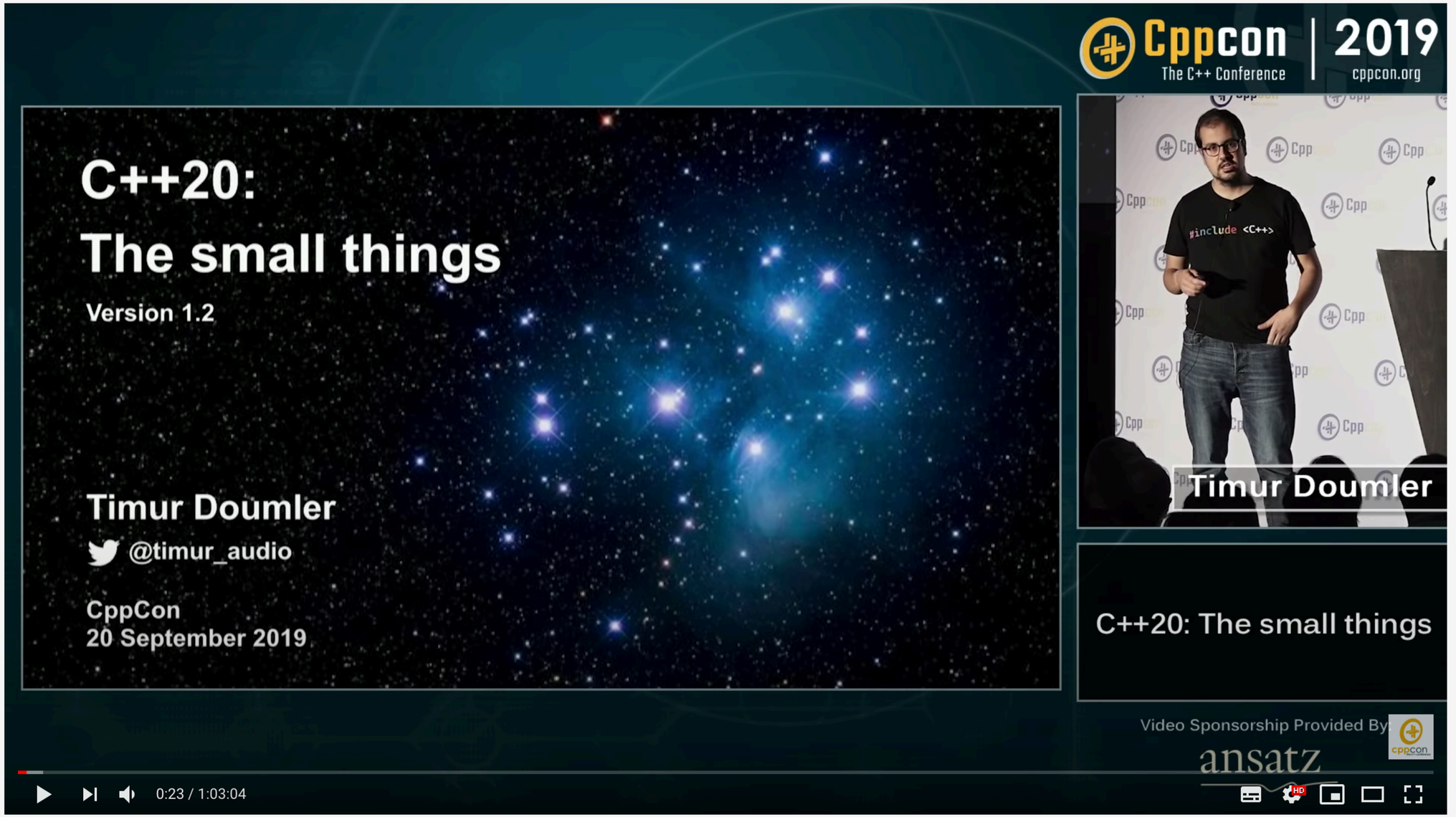
```
struct sorter_type {
    bool operator()(const std::vector<int>& lhs, const std::vector<int>& rhs) {
        return /* something */
    }
} sorter;

std::sort(vec.begin(), vec.end(), sorter);
```

```
std::vector<std::vector<int> > vec = {{1, 0}, {0, 1}};

for (std::vector<std::vector<int> >::iterator i = vec.begin(); i != vec.end(); ++i)
    for (std::vector<int>::const_iterator j = i->begin(); j != i->end(); ++j)
        /* do something */

struct sorter_type {
    bool operator()(const std::vector<int>& lhs, const std::vector<int>& rhs) {
        return /* something */
    }
} sorter;

std::sort(vec.begin(), vec.end(), sorter);
```

```
std::vector<std::vector<int> > vec = {{1, 0}, {0, 1}};

for (std::vector<std::vector<int> >::iterator i = vec.begin(); i != vec.end(); ++i)
    for (std::vector<int>::const_iterator j = i->begin(); j != i->end(); ++j)
        /* do something */

struct sorter_type {
    bool operator()(const std::vector<int>& lhs, const std::vector<int>& rhs) {
        return /* something */
    }
} sorter;

std::sort(vec.begin(), vec.end(), sorter);
```

```
std::vector<std::vector<int>> vec = {{1, 0}, {0, 1}};

for (auto&& vi : vec)
    for (auto&& i : vi)
        /* do something */

std::sort(vec.begin(),
          vec.end(),
          [](const auto& lhs, const auto& rhs) { return /* something */ });
```


C++98
C++03
C++11
C++14
C++17
C++20
2000
1500
1000
500
0
Number of pages



AURORA

CppCon 2019: Timur Doumler “C++20: The small things”

12,071 views • 9 Oct 2019

230 2 SHARE SAVE

coroutines
concepts
ranges
modules

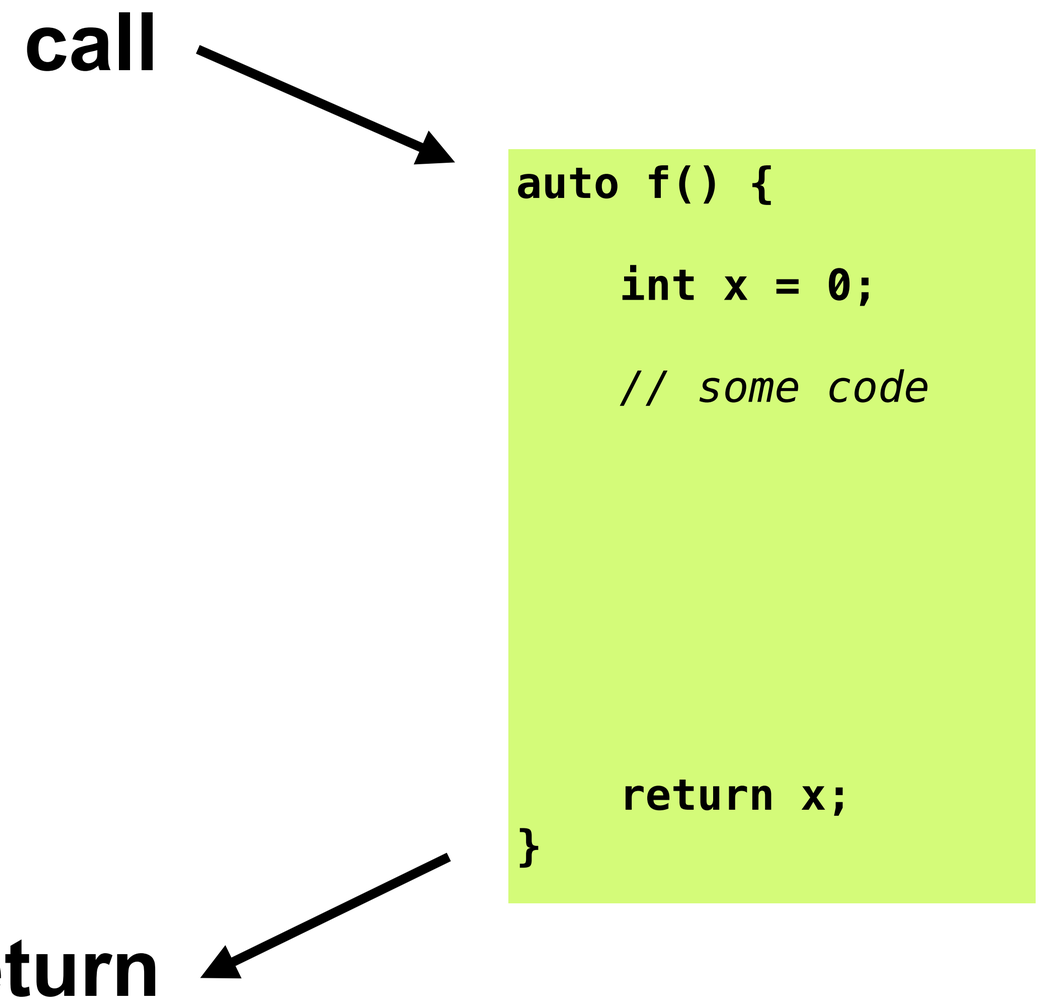
coroutines

concepts

ranges

modules

functions


call
auto f() {
int x = 0;
// some code
return x;
}
return

1952

THE USE OF SUB-ROUTINES IN PROGRAMMES

D. J. Wheeler

Cambridge & Illinois Universities

A sub-routine may perhaps best be described as a self-contained part of a programme, which is capable of being used in different programmes. It is an entity of its own within a programme. There is no necessity to compose a programme of a set of distinct sub-routines; for the programme can be written as a complete unit, with no divisions into smaller parts. However it is usually advantageous to arrange that a programme is comprised of a set of sub-routines, some of which have been made specially for the particular programme while others are easier to use a sub-routine which will meet the specifications with a small amount of manipulation than to make one specially for the purpose.

It should be pointed out that the preparation of a library sub-routine requires a considerable amount of work. This is much greater than the effort merely required to code the sub-routine in its simplest possible form. It will usually be necessary to code it in the library standard form and this may detract from its efficiency in time and space. It may be desirable to code it in such a manner that the operation is

```
int f();

int main() {
    std::cout << f() << '\n';
}
```

```
int f() {
    return 0;
}

int main() {
    std::cout << f() << '\n';
}

// Output:
// 0
```

```
int f() {
    return 0;
}

int main() {
    std::cout << f() << '\n';
    std::cout << f() << '\n';
    std::cout << f() << '\n';
}

// Output:
// 0
// 0
// 0
```

```
int main() {
    for (int i = 0; i < 3; ++i)
        std::cout << i << ‘\n';
}

// Output:
// 0
// 1
// 2
```

```cpp
int main() {
    std::vector seq = {0, 1, 2};

    for (auto i : seq)
        std::cout << i << '\n';
}

// Output:
// 0
// 1
// 2
```

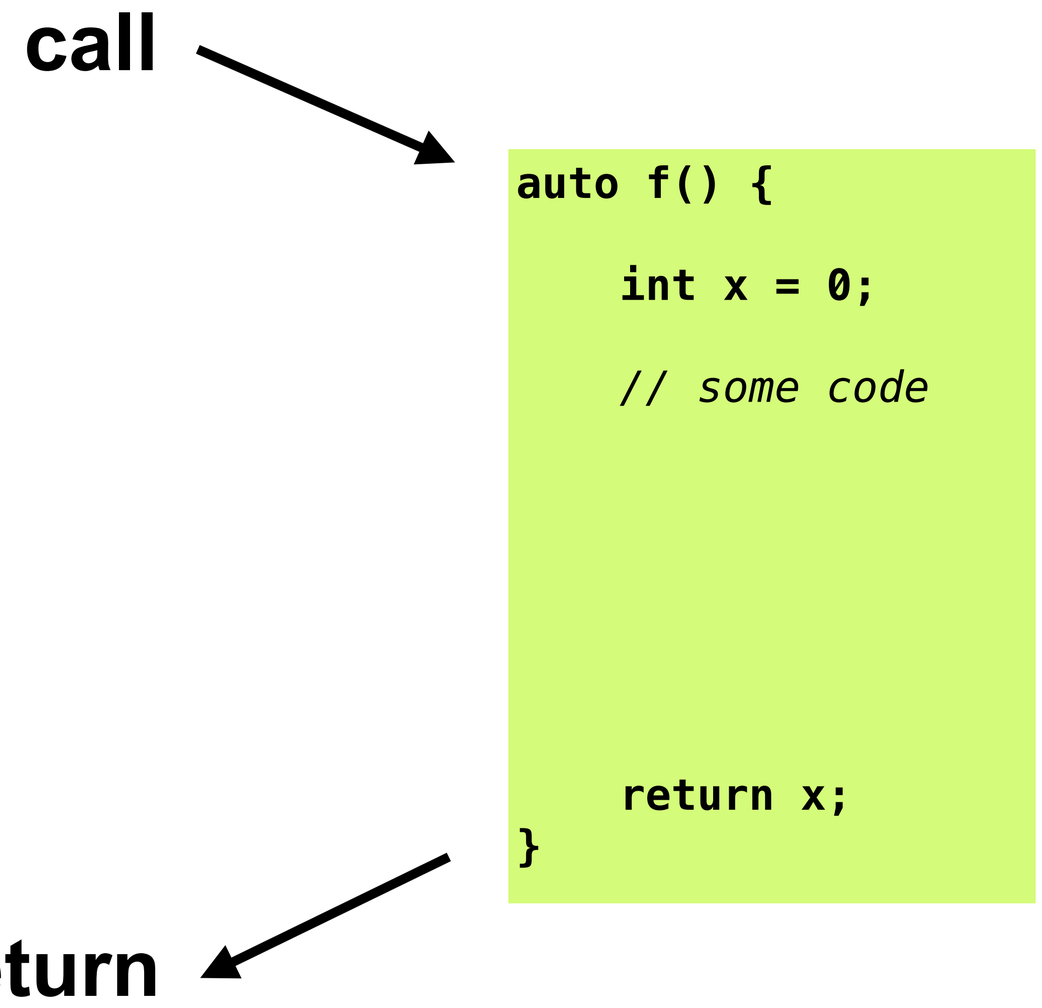
```
int f() {
    // ???
}

int main() {
    std::cout << f() << '\n';
    std::cout << f() << '\n';
    std::cout << f() << '\n';
}

// Output:
// 0
// 1
// 2
```

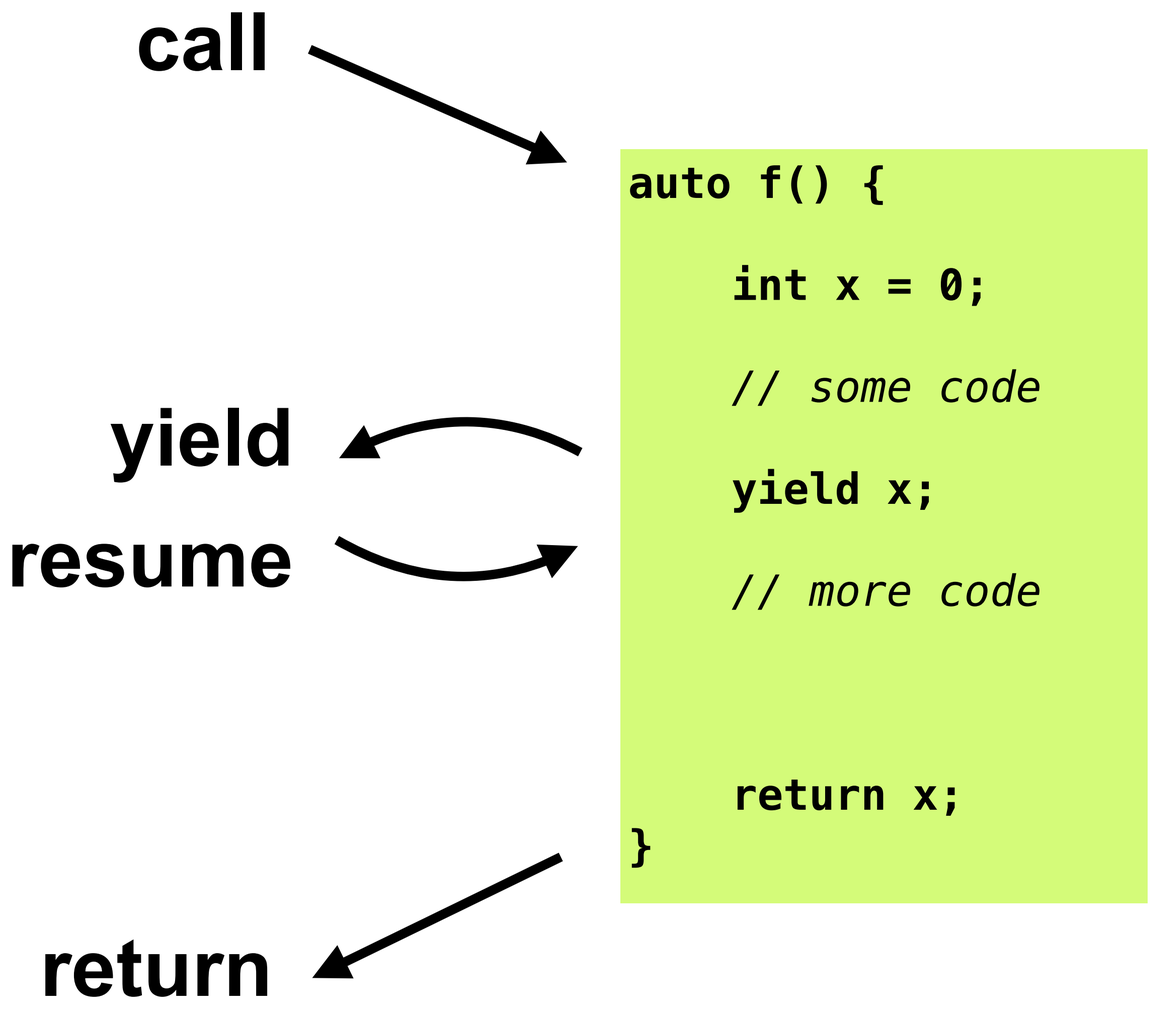
```
int f() {
    static int i = 0;
    return i++;
}

int main() {
    std::cout << f() << '\n';
    std::cout << f() << '\n';
    std::cout << f() << '\n';
}

// Output:
// 0
// 1
// 2
```

```
int f() {
    static int i = 0;  // global state :(
    return i++;
}

int main() {
    std::cout << f() << '\n';
    std::cout << f() << '\n';
    std::cout << f() << '\n';
}

// Output:
// 0
// 1
// 2
```

```
class my_generator {
    int i = 0;
public:
    int operator()() {
        return i++;
    }
};

int main() {
    my_generator g;
    std::cout << g() << '\n';
    std::cout << g() << '\n';
    std::cout << g() << '\n';
}

// Output:
// 0
// 1
// 2
```

```
int main() {
    auto g = [i = 0]() mutable {
        return i++;
    };

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

// Output:
// 0
// 1
// 2
```


call
auto f() {
int x = 0;
// some code
return x;
}
return




```
auto f() {
    int x = 0;
    // some code
    yield x;
    // more code

    return x;
}
```

1963

Design of a Separable Transition-Diagram Compiler*

MELVIN E. CONWAY
Directorate of Computers, USAF
L. G. Hanscom Field, Bedford, Mass.

A COBOL compiler design is presented which is compact enough to permit rapid, one-pass compilation of a large subset of COBOL on a moderately large computer. Versions of the same compiler for smaller machines require only two working tapes plus a compiler tape. The methods given are largely applicable to the construction of ALGOL compilers.

Introduction

This paper is written in rebuttal of three propositions widely held among compiler writers, to wit: (1) syntax-directed compilers [1] suffer practical disadvantages over other types of compilers, chiefly in speed; (2) compilers should be written with compilers; (3) COBOL [2] compilers must be complicated. The form of the rebuttal is to describe a high-speed, one-pass, syntax-directed COBOL compiler to make this design (in which all tables are accessed while stored in memory) practical on contemporary computers. None of these techniques is limited in application to COBOL compilers. The following specific techniques are discussed: the coroutine method of separating programs, transition diagrams in syntactical analysis, data name qualification analysis, and instruction generation for conditional statements.

The algorithms described were verified on the 5000-word Burroughs 220 at the Case Institute of Technology Computing Center. A two-pass configuration was planned for that machine, and first-pass code was checked out through the syntactical analysis. At the time the project was discontinued a complete COBOL syntax checker was operating at 140 fully-punched source cards per minute. (The Case 220 had a typical single-address instruction time of 100 microseconds.) Remarks presented later suggest that a complete one-pass version of the compiler, which would be feasible on a 10,000-word machine, would run at well over 100 source cards per minute.

Coroutines and Separable Programs

That property of the design which makes it amenable to many segment configurations is its *separability*. A program organization is separable if it is broken up into processing

```
int f() {
    int i = 0;
    while (true)
        yield i++;
}

int main() {
    std::cout << f() << '\n';
    std::cout << f() << '\n';
    std::cout << f() << '\n';
}

// Output:
// 0
// 1
// 2
```

```
int f() {
    int i = 0;
    while (true)
        co_yield i++;
}

int main() {
    std::cout << f() << '\n';
    std::cout << f() << '\n';
    std::cout << f() << '\n';
}

// Output:
// 0
// 1
// 2
```

```
my_generator<int> f() {
    int i = 0;
    while (true)
        co_yield i++;
}

int main() {
    auto g = f();
    std::cout << g() << '\n';
    std::cout << g() << '\n';
    std::cout << g() << '\n';
}

// Output:
// 0
// 1
// 2
```

User code

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

User code

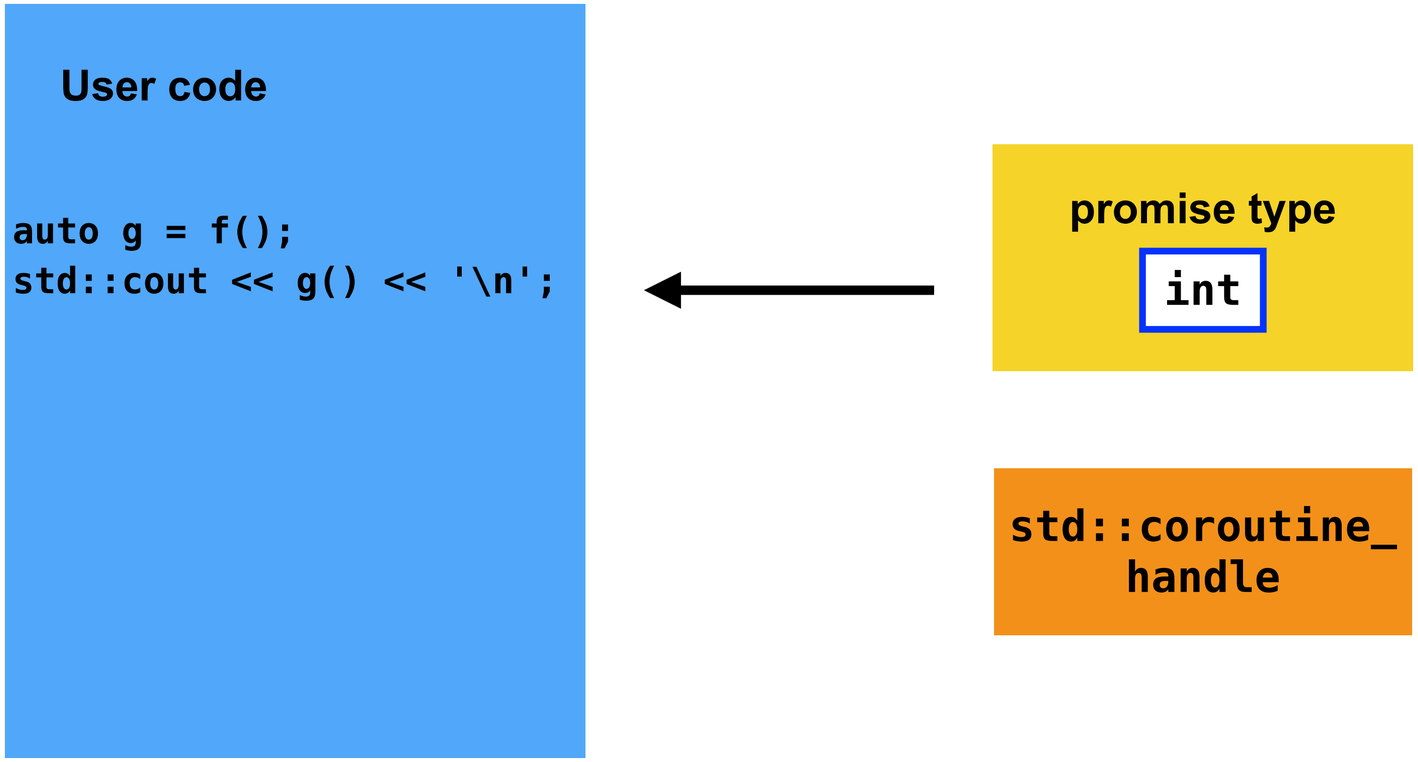
```
auto g = f();
std::cout << g() << '\n';
```

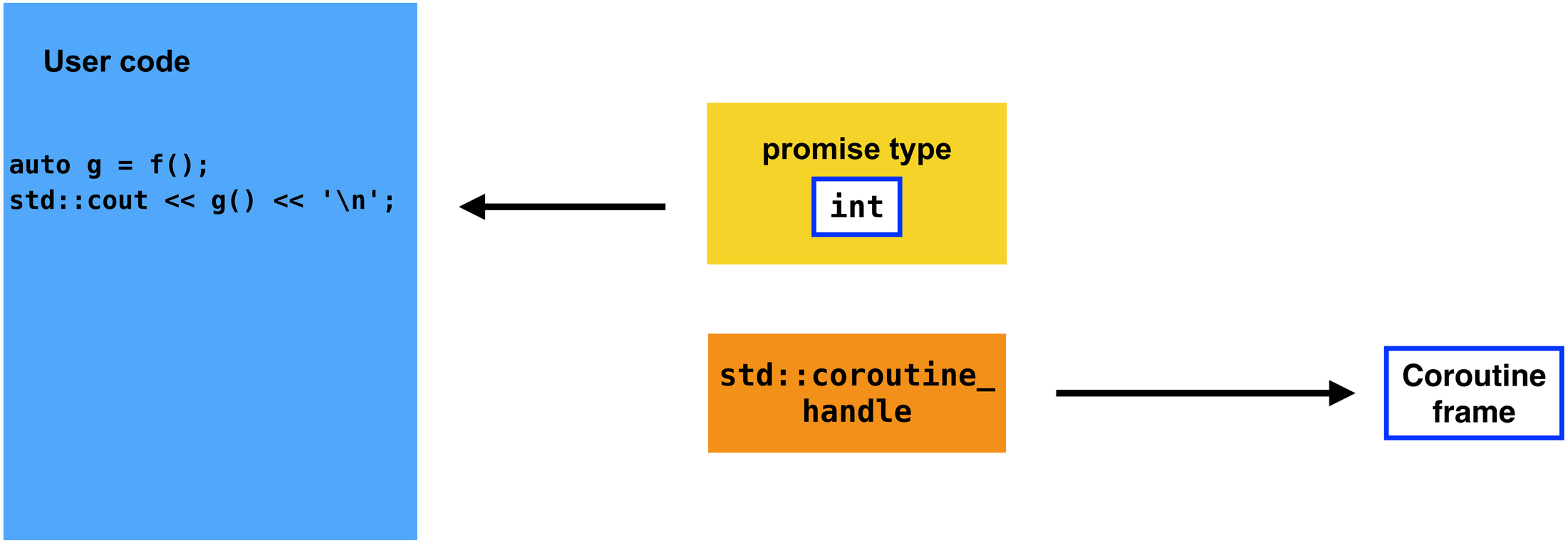
User code

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

promise type

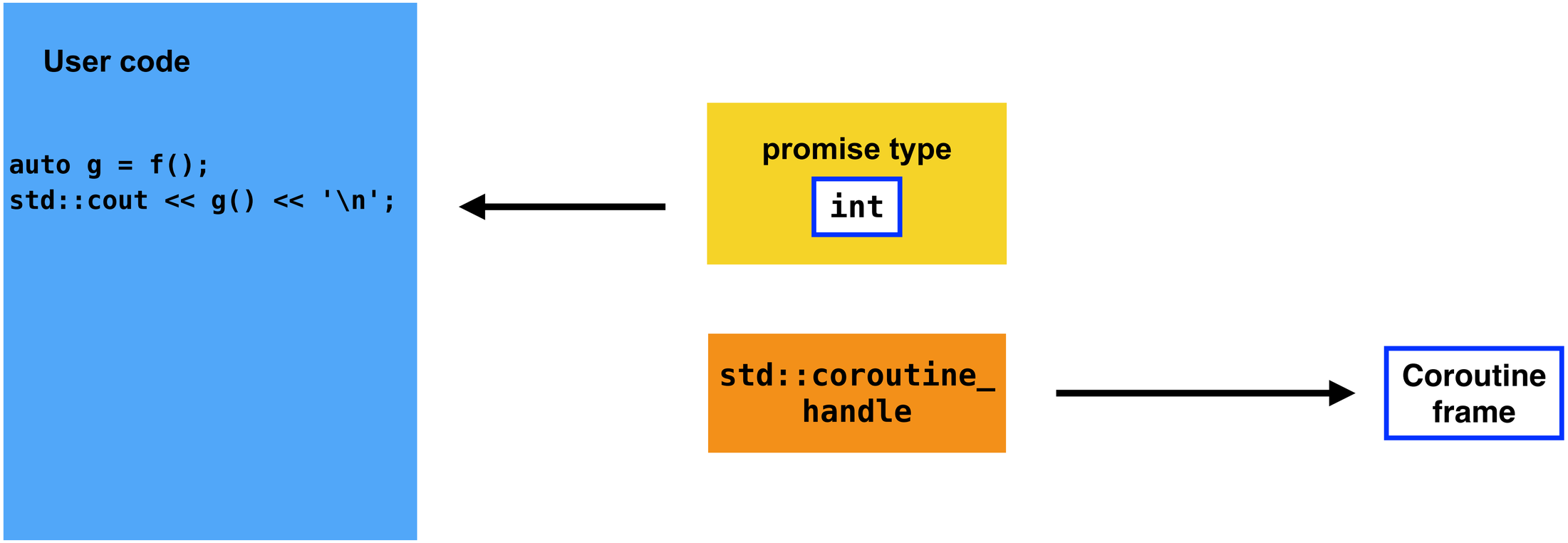
`int`

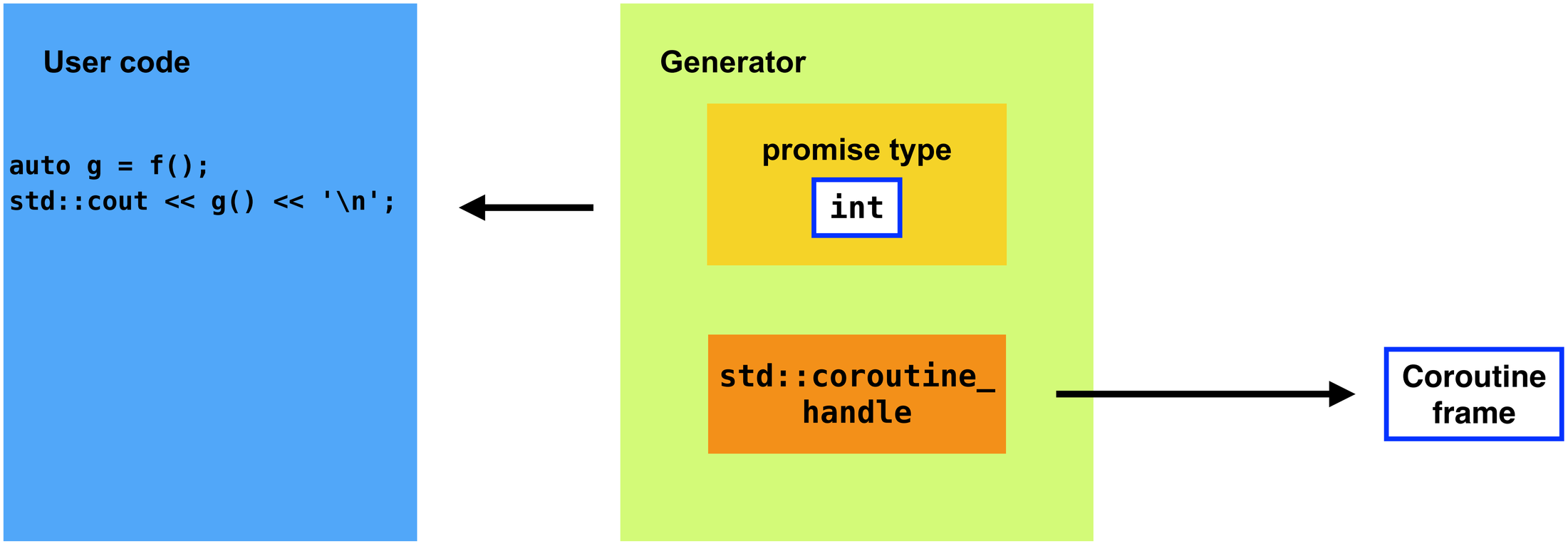

User code
auto g = f();
std::cout << g() << '\n';
promise type
int
std::coroutine_
handle

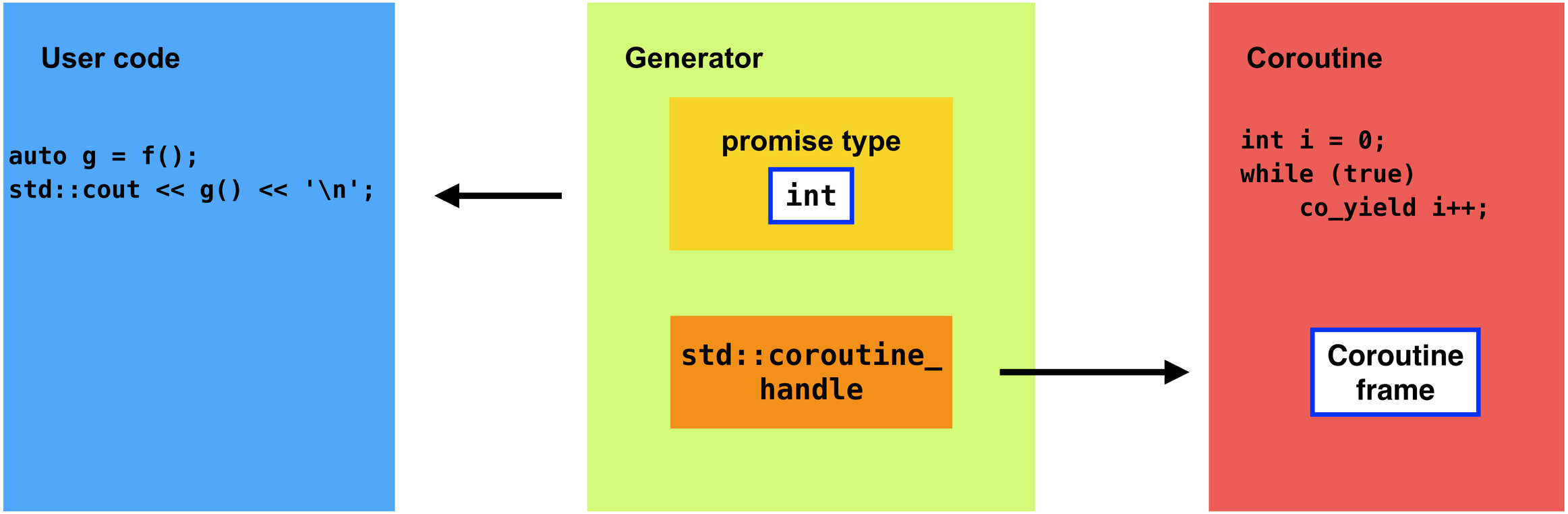

User code
auto g = f();
std::cout << g() << '\n';
promise type
int
std::coroutine_
handle
Coroutine
frame

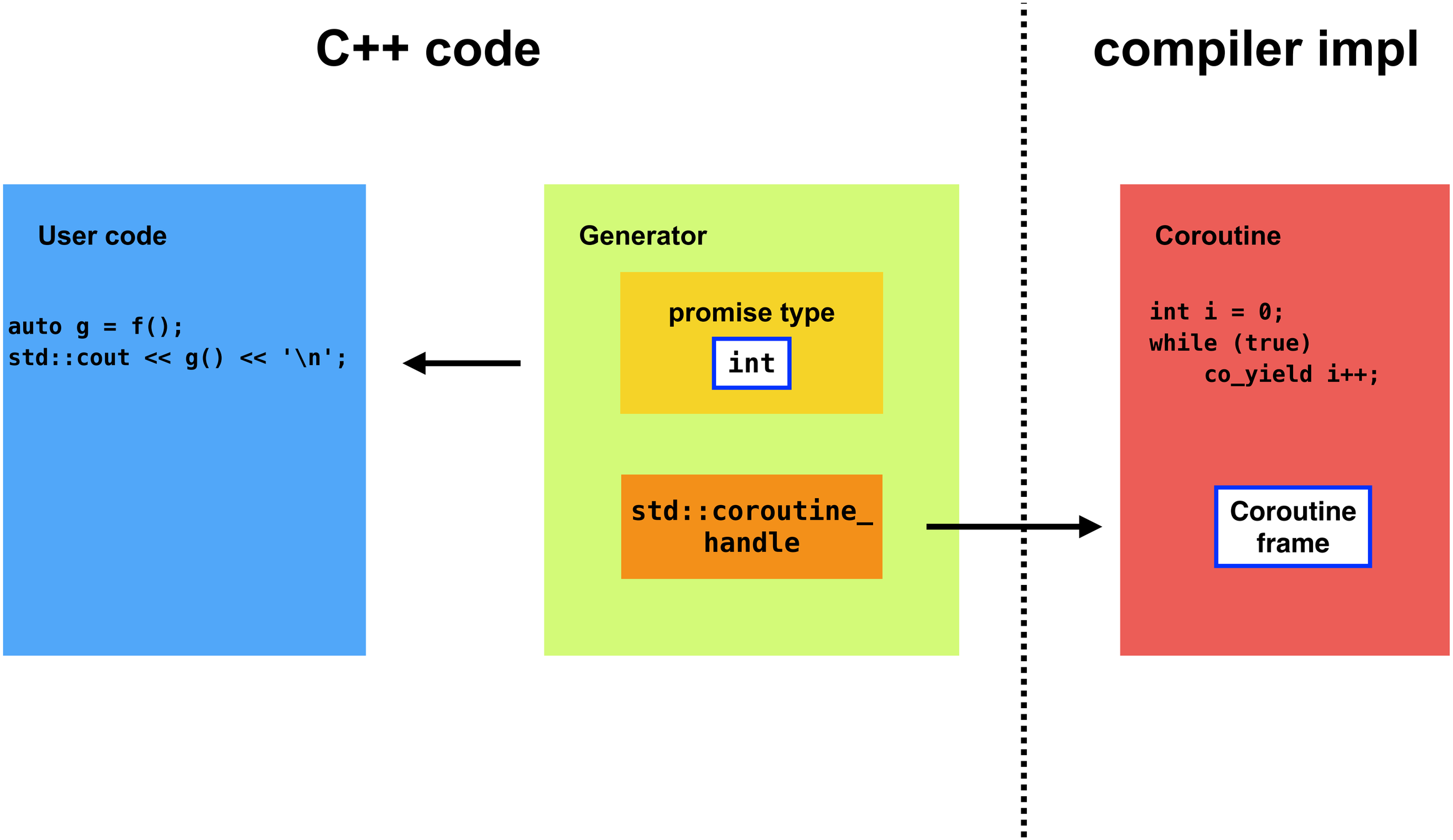
```
int main() {

    auto g = [i = 0]() mutable {
        return i++;
    };

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

// Output:
// 0
// 1
// 2
```

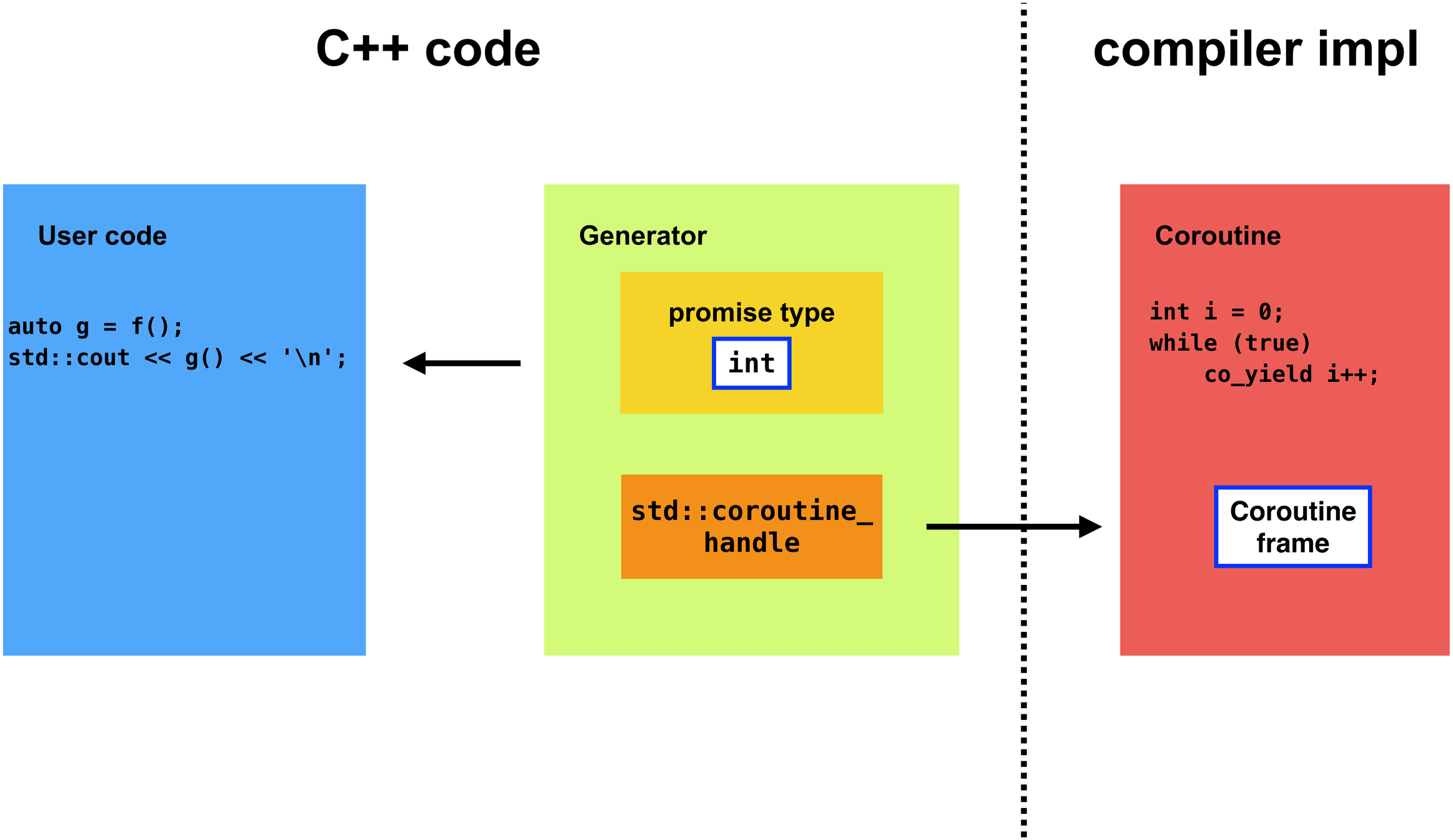
```
int main() {

    auto g = [i = 0]() mutable {
        return i++;
    };

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

// Output:
// 0
// 1
// 2
```

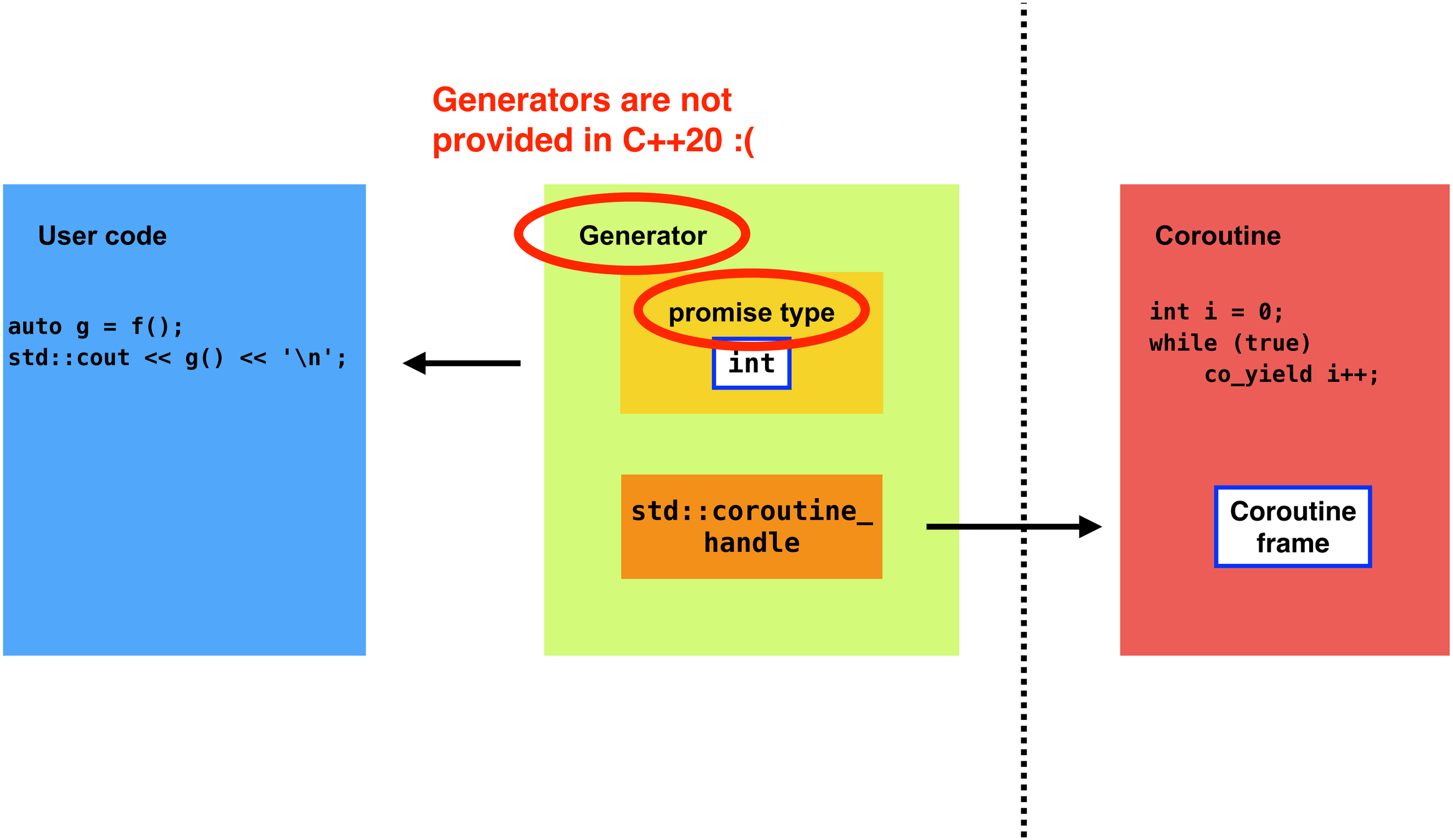

User code
auto g = f();
std::cout << g() << '\n';
promise type
int
std::coroutine_
handle
Coroutine
frame


User code
auto g = f();
std::cout << g() << '\n';
Generator
promise type
int
std::coroutine_
handle
Coroutine
frame


User code
auto g = f();
std::cout << g() << '\n';
Generator
promise type
int
std::coroutine_
handle
Coroutine
int i = 0;
while (true)
co_yield i++;
Coroutine
frame


C++ code
compiler impl
User code
auto g = f();
std::cout << g() << '\n';
Generator
promise type
int
std::coroutine_
handle
Coroutine
int i = 0;
while (true)
co_yield i++;
Coroutine
frame


C++ code
compiler impl
User code
auto g = f();
std::cout << g() << '\n';
Generator
promise type
int
std::coroutine_
handle
Coroutine
int i = 0;
while (true)
co_yield i++;
Coroutine
frame


Generators are not
provided in C++20 :(
User code
auto g = f();
std::cout << g() << '\n';
Generator
promise type
int
std::coroutine_
handle
Coroutine
int i = 0;
while (true)
co_yield i++;
Coroutine
frame

`std::generator`: Synchronous Coroutine Generator for Ranges

Document #:	P2168R0
Date:	2020-05-16
Project:	Programming Language C++
Audience:	LEWG

Reply-to:	Lewis Baker <lbaker@fb.com >
	Corentin Jabot <corentin.jabot@gmail.com>

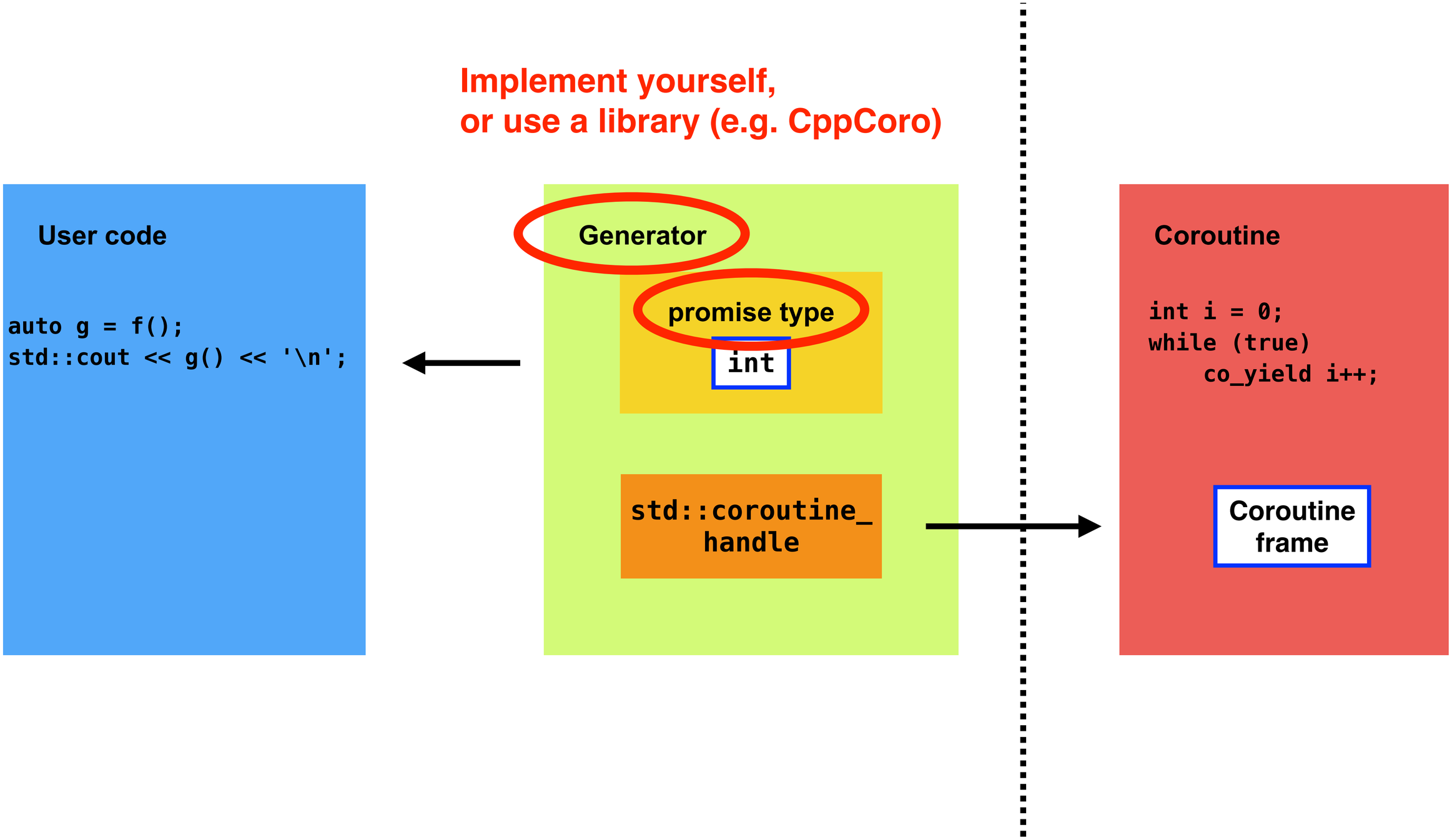
Abstract

We propose a standard library type `std::generator` which implements a coroutine generator compatible with ranges.

Example

```
std::generator<int> fib (int max) {
    co_yield 0;
    auto a = 0, b = 1;
```

Coming in C++23 :)


Implement yourself,
or use a library (e.g. CppCoro)
User code
auto g = f();
std::cout << g() << '\n';
Generator
promise type
int
std::coroutine_
handle
Coroutine
int i = 0;
while (true)
co_yield i++;
Coroutine
frame

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

    auto get_return_object() {
        return generator(*this);
    }

    auto initial_suspend() {
        return std::experimental::suspend_always{};
    }

    auto final_suspend() {
        return std::experimental::suspend_always{};
    }

    auto yield_value(const T& value) {
        current_value = value;
        return std::experimental::suspend_always{};
    }

    void return_void() {}

    void unhandled_exception() {
        std::terminate();
    }
};
```

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

    auto get_return_object() {
        return generator(*this);
    }

    auto initial_suspend() {
        return std::experimental::suspend_always{};
    }

    auto final_suspend() {
        return std::experimental::suspend_always{};
    }

    auto yield_value(const T& value) {
        current_value = value;
        return std::experimental::suspend_always{};
    }

    void return_void() {}

    void unhandled_exception() {
        std::terminate();
    }
};
```

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

    using handle_type = std::coroutine_handle<promise_type>;
    handle_type handle;

    struct promise_type {
        // implementation...
    };

    generator(promise_type& promise) :
        handle(handle_type::from_promise(promise)) {
    }

    generator(const generator&) = delete;

    generator& operator=(const generator&) = delete;

    generator(generator&& other) : handle(other.handle) {
        other.handle = nullptr;
    }

    generator& operator=(generator&& other) {
        if (this != &other) {
            handle = other.handle;
            other.handle = nullptr;
        }
        return *this;
    }

    ~generator() {
        if (handle)
            handle.destroy();
    }

    T operator()() {
        assert(handle != nullptr);
        handle.resume();
        return handle.promise().current_value;
    }
};
```

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

    using handle_type = std::coroutine_handle<promise_type>;
    handle_type handle;

    struct promise_type {
        // implementation...
    };

    generator(promise_type& promise) :
        handle(handle_type::from_promise(promise)) {
    }

    generator(const generator&) = delete;

    generator& operator=(const generator&) = delete;

    generator(generator&& other) : handle(other.handle) {
        other.handle = nullptr;
    }

    generator& operator=(generator&& other) {
        if (this != &other) {
            handle = other.handle;
            other.handle = nullptr;
        }
        return *this;
    }

    ~generator() {
        if (handle)
            handle.destroy();
    }

    T operator()() {
        assert(handle != nullptr);
        handle.resume();
        return handle.promise().current_value;
    }
};
```

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

    using handle_type = std::coroutine_handle<promise_type>;
    handle_type handle;

    struct promise_type {
        // implementation...
    };

    generator(promise_type& promise) :
        handle(handle_type::from_promise(promise)) {
    }

    generator(const generator&) = delete;

    generator& operator=(const generator&) = delete;

    generator(generator&& other) : handle(other.handle) {
        other.handle = nullptr;
    }

    generator& operator=(generator&& other) {
        if (this != &other) {
            handle = other.handle;
            other.handle = nullptr;
        }
        return *this;
    }

    ~generator() {
        if (handle)
            handle.destroy();
    }

    T operator()() {
        assert(handle != nullptr);
        handle.resume();
        return handle.promise().current_value;
    }
};
```

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

    using handle_type = std::coroutine_handle<promise_type>;
    handle_type handle;

    struct promise_type {
        // implementation...
    };

    generator(promise_type& promise) :
        handle(handle_type::from_promise(promise)) {
    }

    generator(const generator&) = delete;

    generator& operator=(const generator&) = delete;

    generator(generator&& other) : handle(other.handle) {
        other.handle = nullptr;
    }

    generator& operator=(generator&& other) {
        if (this != &other) {
            handle = other.handle;
            other.handle = nullptr;
        }
        return *this;
    }

    ~generator() {
        if (handle)
            handle.destroy();
    }

    T operator()() {
        assert(handle != nullptr);
        handle.resume();
        return handle.promise().current_value;
    }
};
```

DON’T TRY THIS AT HOME

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

int main() {
    auto g = f();
    std::cout << g() << '\n';
    std::cout << g() << '\n';
    std::cout << g() << '\n';
}

// Output:
// 0
// 1
// 2
```

reader

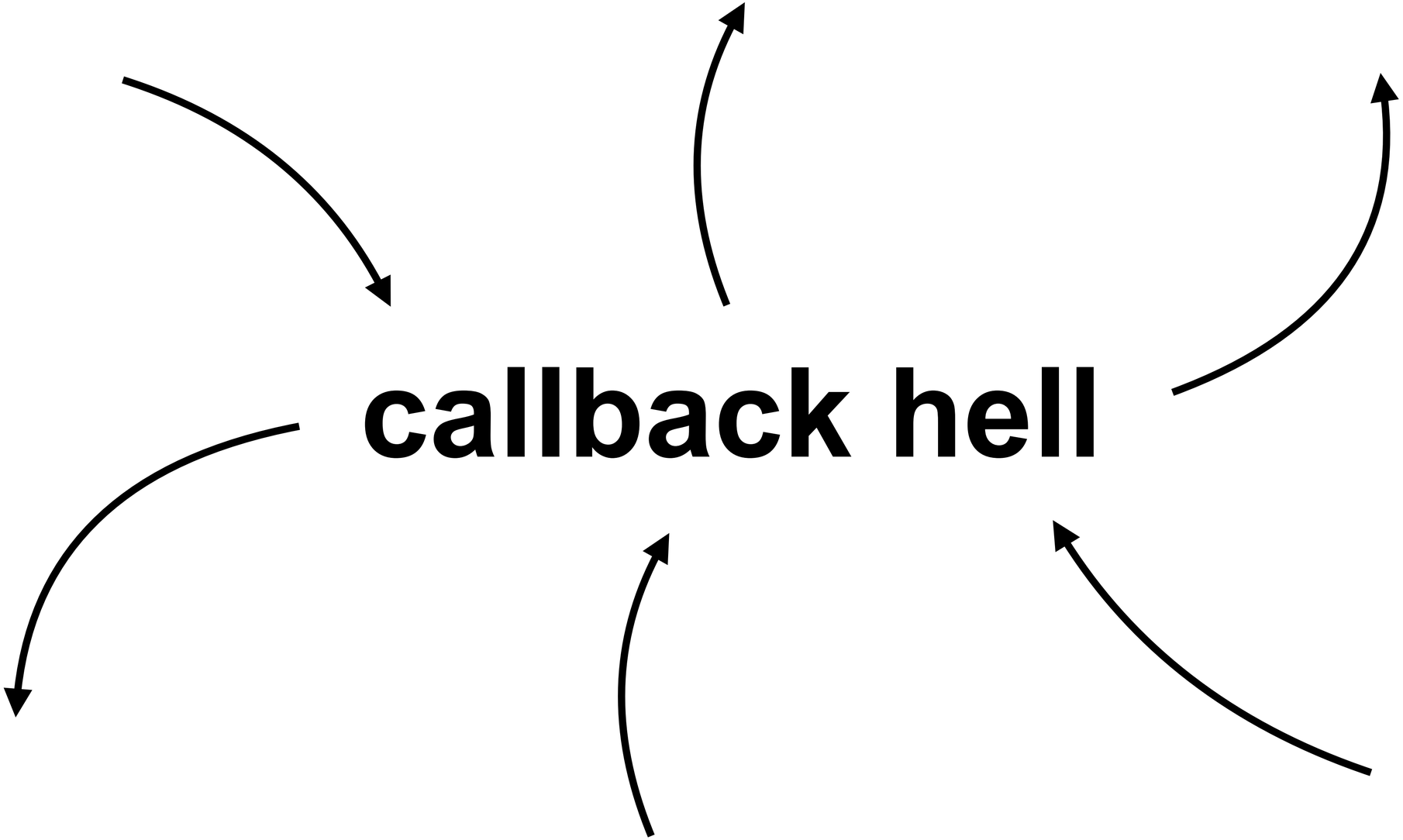
```
generator<char>
get_char() {
  while (!eof) {

    // code...

    co_yield c;
  }
}
```

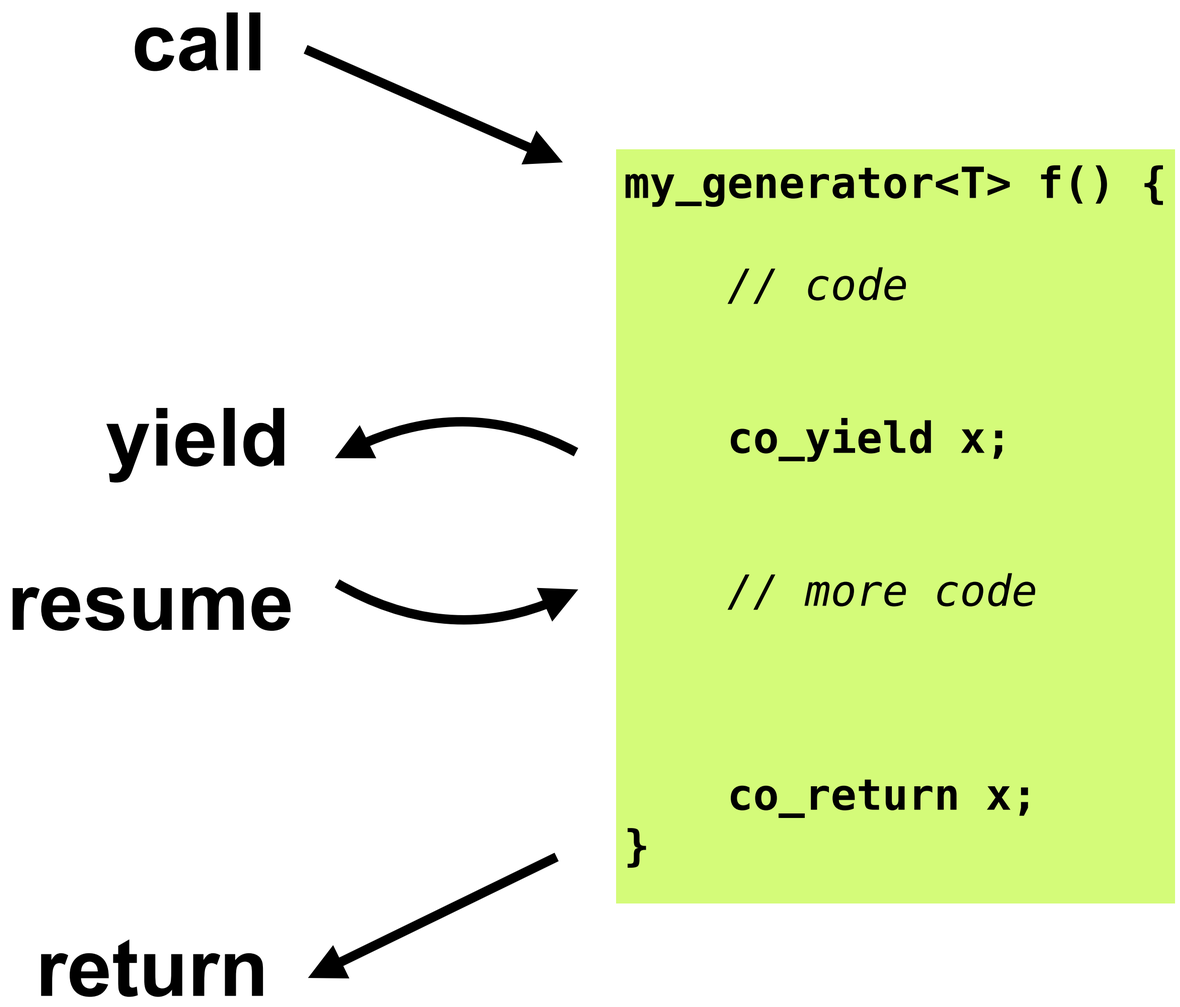
lexer

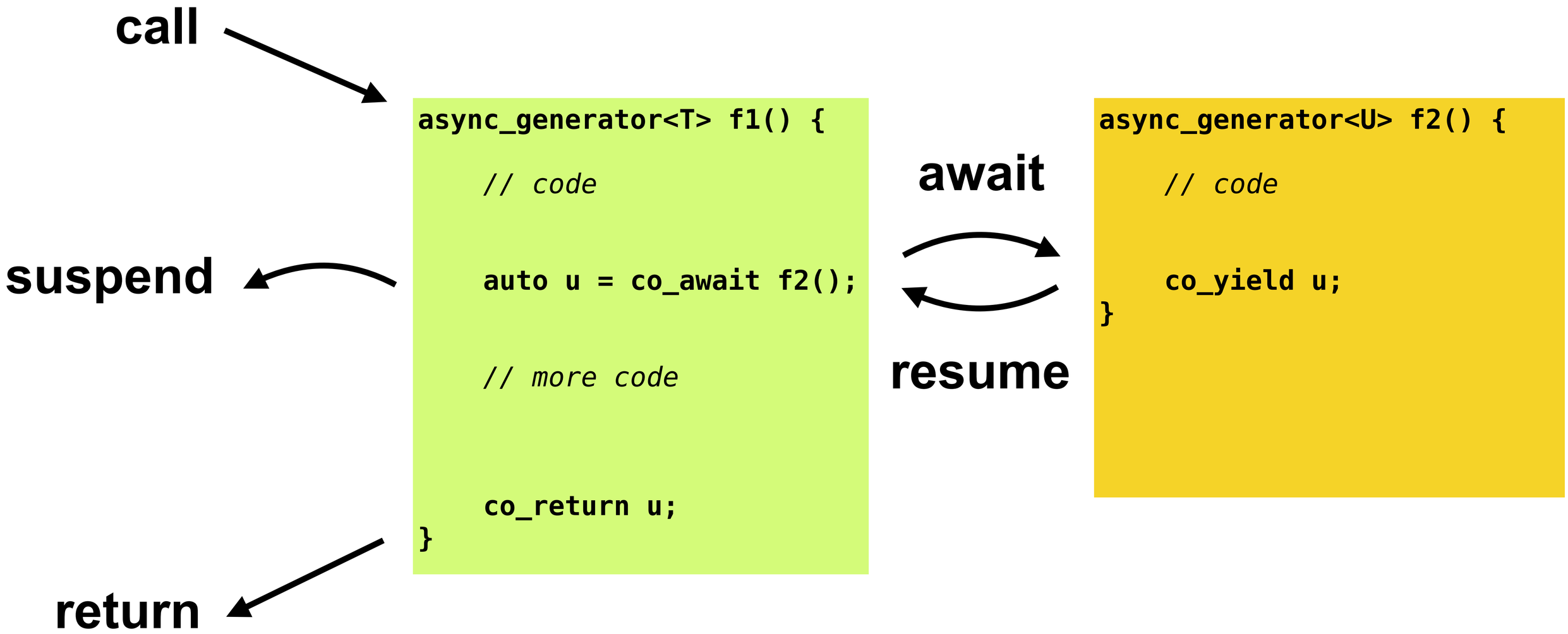
```
generator<token>
get_token() {

  while (auto c = get_char()) {

    // code...

    co_yield t;
  }
}
```

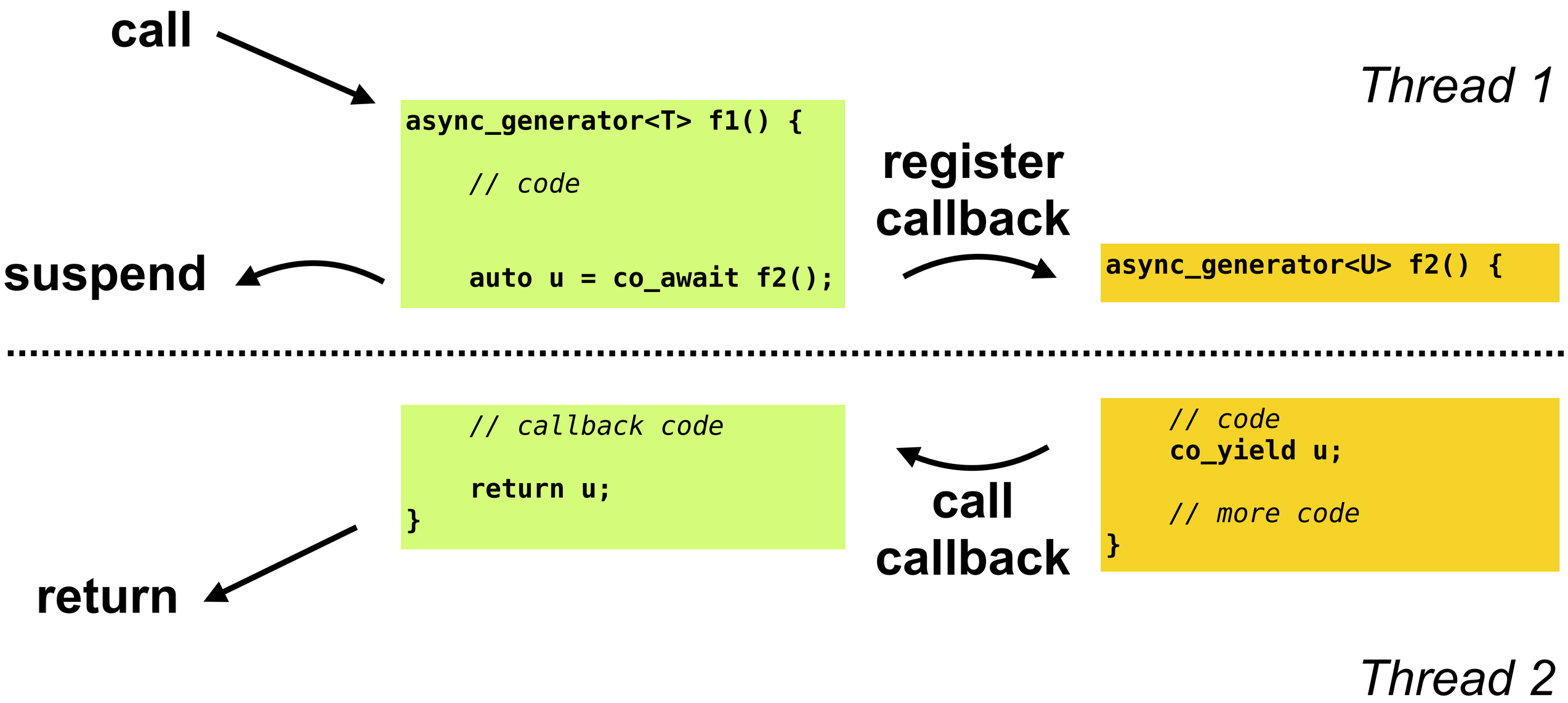
parser

```
generator<ast_node>
parse_expression() {

    auto t = get_token();

    // code...

    co_yield node;
}
```


callback hell


call
yield
resume
return
my_generator<T> f() {
// code
co_yield x;
// more code
co_return x;
}


call
async_generator<T> f1() {
// code
auto u = co_await f2();
// more code
co_return u;
}
suspend
return
await
resume
async_generator<U> f2() {
// code
co_yield u;
}


call
async_generator<T> f1() {
// code
auto u = co_await f2();
suspend
register
callback
async_generator<U> f2() {
Thread 1
// callback code
return u;
}
return
call
callback
// code
co_yield u;
// more code
}
Thread 2

coroutines

concepts

ranges

modules

functions

```
bool is_power_of_2(int i) {
    return i > 0 && (i & (i - 1)) == 0;
}
```

```
bool is_power_of_2(short i) {
    return i > 0 && (i & (i - 1)) == 0;
}

bool is_power_of_2(unsigned short i) {
    return i > 0 && (i & (i - 1)) == 0;
}

bool is_power_of_2(int i) {
    return i > 0 && (i & (i - 1)) == 0;
}

bool is_power_of_2(unsigned int i) {
    return i > 0 && (i & (i - 1)) == 0;
}

bool is_power_of_2(long i) {
    return i > 0 && (i & (i - 1)) == 0;
}

bool is_power_of_2(unsigned long i) {
    return i > 0 && (i & (i - 1)) == 0;
}
```

```
bool is_power_of_2(long long i) {
    return i > 0 && (i & (i - 1)) == 0;
}

bool is_power_of_2(unsigned long long i) {
    return i > 0 && (i & (i - 1)) == 0;
}

bool is_power_of_2(char i) {
    return i > 0 && (i & (i - 1)) == 0;
}

bool is_power_of_2(signed char i) {
    return i > 0 && (i & (i - 1)) == 0;
}

bool is_power_of_2(unsigned char i) {
    return i > 0 && (i & (i - 1)) == 0;
}

bool is_power_of_2(wchar_t i) {
    return i > 0 && (i & (i - 1)) == 0;
}
```

```
template <typename T>
bool is_power_of_2(T i) {
    return i > 0 && (i & (i - 1)) == 0;
}
```

```
template <typename T>
bool is_power_of_2(T i) {
    return i > 0 && (i & (i - 1)) == 0;
}

int main() {
    std::cout << is_power_of_2(0.25);
    // error: invalid operands to binary expression
}
```

```
template <typename T>
bool is_power_of_2(T i) {
    static_assert(std::is_integral_v<T>);
    return i > 0 && (i & (i - 1)) == 0;
}

int main() {
    std::cout << is_power_of_2(0.25);
    // error: static_assert failed due to requirement 'std::is_integral_v<double>'
}
```

```
template <typename T>
bool is_power_of_2(T i) {
    static_assert(std::is_integral_v<T>);
    return i > 0 && (i & (i - 1)) == 0;
}

template <typename T>
bool is_power_of_2(T x) {
    static_assert(std::is_floating_point_v<T>);
    int exponent;
    const T mantissa = std::frexp(x, &exponent);
    return mantissa == T(0.5);
}

int main() {
    std::cout << is_power_of_2(0.25);
    // error: redefinition of function template ‘is_power_of_2’
}
```

```
template <typename T>
auto is_power_of_2(T i) -> std::enable_if_t<std::is_integral_v<T>, bool> {
    return i > 0 && (i & (i - 1)) == 0;
}

template <typename T>
auto is_power_of_2(T x) -> std::enable_if_t<std::is_floating_point_v<T>, bool> {
    int exponent;
    const T mantissa = std::frexp(x, &exponent);
    return mantissa == T(0.5);
}


int main() {
    std::cout << is_power_of_2(0.25);
}
```

```
template <typename T>
bool is_power_of_2(T i, std::enable_if_t<std::is_integral_v<T>, int> = 0) {
return i > 0 && (i & (i - 1)) == 0;
}

template <typename T>
bool is_power_of_2(T x, std::enable_if_t<std::is_floating_point_v<T>, int> = 0) {
    int exponent;
    const T mantissa = std::frexp(x, &exponent);
    return mantissa == T(0.5);
}


int main() {
    std::cout << is_power_of_2(0.25);
}
```

```
template <typename T, typename = std::enable_if_t<std::is_integral_v<T>>>
bool is_power_of_2(T i) {
    return i > 0 && (i & (i - 1)) == 0;
}

template <typename T, typename = std::enable_if_t<std::is_floating_point_v<T>>>
bool is_power_of_2(T x) {
    int exponent;
    const T mantissa = std::frexp(x, &exponent);
    return mantissa == T(0.5);
}


int main() {
    std::cout << is_power_of_2(0.25);
    // error: redefinition of function template ‘is_power_of_2’
}
```

```
template <typename T, std::enable_if_t<std::is_integral_v<T>, int> = 0>
bool is_power_of_2(T i) {
    return i > 0 && (i & (i - 1)) == 0;
}

template <typename T, std::enable_if_t<std::is_floating_point_v<T>, int> = 0>
bool is_power_of_2(T x) {
    int exponent;
    const T mantissa = std::frexp(x, &exponent);
    return mantissa == T(0.5);
}


int main() {
    std::cout << is_power_of_2(0.25);
}
```

```cpp
template <typename T, std::enable_if_t<std::is_integral_v<T>, void*> = nullptr>
bool is_power_of_2(T i) {
    return i > 0 && (i & (i - 1)) == 0;
}

template <typename T, std::enable_if_t<std::is_floating_point_v<T>, void*> = nullptr>
bool is_power_of_2(T x) {
    int exponent;
    const T mantissa = std::frexp(x, &exponent);
    return mantissa == T(0.5);
}


int main() {
    std::cout << is_power_of_2(0.25);
}
```

```
template <typename T> requires std::integral<T>
bool is_power_of_2(T i) {
    return i > 0 && (i & (i - 1)) == 0;
}

template <typename T> requires std::floating_point<T>
bool is_power_of_2(T x) {
    int exponent;
    const T mantissa = std::frexp(x, &exponent);
    return mantissa == T(0.5);
}


int main() {
    std::cout << is_power_of_2(0.25);
}
```

```
template <std::integral T>
bool is_power_of_2(T i) {
    return i > 0 && (i & (i - 1)) == 0;
}

template <std::floating_point T>
bool is_power_of_2(T x) {
    int exponent;
    const T mantissa = std::frexp(x, &exponent);
    return mantissa == T(0.5);
}


int main() {
    std::cout << is_power_of_2(0.25);
}
```

```
bool is_power_of_2(std::integral auto i) {
    return i > 0 && (i & (i - 1)) == 0;
}


template <std::floating_point T>
bool is_power_of_2(T x) {
    int exponent;
    const T mantissa = std::frexp(x, &exponent);
    return mantissa == T(0.5);
}


int main() {
    std::cout << is_power_of_2(0.25);
}
```

```
#include <concepts>

template <typename T>
concept arithmetic = std::integral<T> || std::floating_point<T>;
```

```
#include <concepts>

template <typename T>
concept arithmetic = std::integral<T> || std::floating_point<T>;

template <typename T>
concept my_number = arithmetic<T> && sizeof(T) <= 8;
```

```
#include <concepts>

template <typename T>
concept arithmetic = std::integral<T> || std::floating_point<T>;

template <typename T>
concept my_number = arithmetic<T> && sizeof(T) <= 8;

auto f(my_number auto x) {
    // code...
}
```

```
#include <concepts>

template <typename T>
concept arithmetic = std::integral<T> || std::floating_point<T>;

template <typename T>
concept my_number = arithmetic<T> && sizeof(T) <= 8;

auto f(my_number auto x) {
    // code...
}

int main() {
    auto x = f(2.0);  // OK
    auto y = f(2.0L); // error: no matching function for call to ‘f’
}                     // note: candidate template ignored: constraints not satisfied
                      // note: because ’long double’ does not satisfy ‘my_number’
                      // note: because ‘sizeof(long double) <= 8’ (16 <= 8)
                                 evaluated to false
```

```
template <typename T>
concept hashable = requires(T t) {
    { std::hash<T>{}(t) } -> std::convertible_to<std::size_t>;
};
```

```
template <typename T>
concept hashable = requires(T t) {
    { std::hash<T>{}(t) } -> std::convertible_to<std::size_t>;
};

template <hashable T>
class hash_map;
```

coroutines

concepts

ranges

modules

```
struct User {
    std::string name;
    int age;
};

std::vector<User> users = { /* ... */ };

int main() {
    auto sort_by_age = [] (auto& lhs, auto& rhs) {
        return lhs.age < rhs.age;
    };

    std::sort(users.begin(), users.end(), sort_by_age);
}
```

```
struct User {
    std::string name;
    int age;
};

std::vector<User> users = { /* ... */ };

int main() {
    auto sort_by_age = [] (auto& lhs, auto& rhs) {
        return lhs.age < rhs.age;
    };

    std::sort(users.begin(), users.end(), sort_by_age);
}
```

```
struct User {
    std::string name;
    int age;
};

std::vector<User> users = { /* ... */ };

int main() {
    auto sort_by_age = [] (auto& lhs, auto& rhs) {
        return lhs.age < rhs.age;
    };

    std::ranges::sort(users, sort_by_age);
}
```

```
struct User {
    std::string name;
    int age;
};

std::vector<User> users = { /* ... */ };

int main() {
    std::ranges::sort(users, std::less{}, &User::age);
}
```

```
template <random_access_range R,
          typename Comp = ranges::less,
          typename Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr safe_iterator_t<R> ranges::sort(R&& r, Comp comp = {}, Proj proj = {});
```

```
template <random_access_range R,
          typename Comp = ranges::less,
          typename Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr safe_iterator_t<R> ranges::sort(R&& r, Comp comp = {}, Proj proj = {});

template <typename I,
          typename R = ranges::less,
          class P = identity>
concept sortable = permutable<I>
                   && indirect_strict_weak_order<R, projected<I, P>>
```

```
template <random_access_range R,
          typename Comp = ranges::less,
          typename Proj = identity>
requires sortable<iterator_t<R>, Comp, Proj>
constexpr safe_iterator_t<R> ranges::sort(R&& r, Comp comp = {}, Proj proj = {});

template <typename I,
          typename R = ranges::less,
          class P = identity>
concept sortable = permutable<I>
                   && indirect_strict_weak_order<R, projected<I, P>>

template <typename I>
concept permutable = forward_iterator<I>
                     && indirectly_movable_storable<I, I>
                     && indirectly_swappable<I, I>;
```

```
struct User {
    std::string name;
    int age;
};

std::vector<User> users = { /* ... */ };
```

```
struct User {
    std::string name;
    int age;
};

const std::vector<User> users = { /* ... */ };
```

```
struct User {
    std::string name;
    int age;
};

const std::vector<User> users = { /* ... */ };

bool underage(const User& user) { return user.age < 18; };

int main() {
    std::vector<User> filtered_users;

    std::copy_if(users.begin(), users.end(),
                 std::back_inserter(filtered_users), std::not_fn(underage));

    std::transform(filtered_users.begin(), filtered_users.end(),
                   std::ostream_iterator<int>(std::cout, "\n"),
                   [] (const auto& user) { return user.age; });
}
```

```
struct User {
    std::string name;
    int age;
};

const std::vector<User> users = { /* ... */ };

bool underage(const User& user) { return user.age < 18; };

int main() {
    auto result = users
        | std::views::filter(std::not_fn(underage))
        | std::views::transform([] (const auto& user) { return user.age; });

    std::ranges::copy(result, std::ostream_iterator<int>(std::cout, “\n");
}
```

```
struct User {
    std::string name;
    int age;
};

const std::vector<User> users = { /* ... */ };

bool underage(const User& user) { return user.age < 18; };

int main() {
    auto result = users
        | std::views::filter(std::not_fn(underage))
        | std::views::transform([] (const auto& user) { return user.age; });

    std::ranges::copy(result, std::ostream_iterator<int>(std::cout, “\n");
}
```

```
struct User {
    std::string name;
    int age;
};

const std::vector<User> users = { /* ... */ };

bool underage(const User& user) { return user.age < 18; };

int main() {
    for (auto& age : users
        | std::views::filter(std::not_fn(underage))
        | std::views::transform([] (const auto& user) { return user.age; }))
    {
        std::cout << age << '\n';
    }
}
```

```
int main() {
    for (int i : std::views::iota(0, 10))
        std::cout << i << '\n';
}
```

```
int main() {
    for (int i : std::views::iota(0, 10))
        std::cout << i << '\n';
}

// Output:
// 0
// 1
// 2
// 3
// 4
// 5
// 6
// 7
// 8
// 9
```

coroutines
concepts
ranges
modules

headers

generator.h

```
template <typename T>
struct generator {
    // code ...
};
```

generator.h

```
template <typename T>
struct generator {
    // code ...
};
```

math.h

```
template <typename T>
bool is_power_of_two(T x) {
    // code ...
}
```

generator.h

```
template <typename T>
struct generator {
    // code ...
};
```

math.h

```
template <typename T>
bool is_power_of_two(T x) {
    // code ...
}
```

coroutine_stuff.cpp

```
#include "mylib/math.h"
#include "mylib/generator.h"
#include "coroutine_stuff.h"

void coroutine_stuff::do_something() {
    // code ...
};
```

generator.h

```
template <typename T>
struct generator {
    // code ...
};
```

math.h

```
template <typename T>
bool is_power_of_two(T x) {
    // code ...
}
```

coroutine_stuff.cpp

```
#include "mylib/math.h"
#include "mylib/generator.h"
#include "coroutine_stuff.h"

void coroutine_stuff::do_something() {
    // code ...
};
```

other_stuff.cpp

```
#include "mylib/math.h"
#include "mylib/generator.h"
#include "other_stuff.h"

void other_stuff::do_something_else() {
    // code ...
}
```

generator.h

```
template <typename T>
struct generator {
    // code ...
};
```

math.h

```
template <typename T>
bool is_power_of_two(T x) {
    // code ...
}
```

coroutine_stuff.cpp

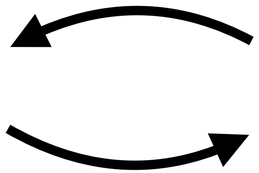
```
#include "mylib/math.h"
#include "mylib/generator.h"
#include "coroutine_stuff.h"

void coroutine_stuff::do_something() {
    // code ...
};
```

other_stuff.cpp

```
#include "mylib/generator.h"
#include "mylib/math.h"
#include "other_stuff.h"

void other_stuff::do_something_else() {
    // code ...
}
```

generator.h

```
template <typename T>
struct generator {
    // code ...
};
```



math.h

```
template <typename T>
bool is_power_of_two(T x) {
    // code ...
}
```

coroutine_stuff.cpp

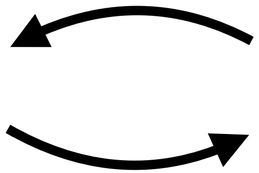
```
#include "mylib/math.h"
#include "mylib/generator.h"
#include "coroutine_stuff.h"

void coroutine_stuff::do_something() {
    // code ...
};
```

other_stuff.cpp

```
#include "mylib/generator.h"
#include "mylib/math.h"
#include "other_stuff.h"

void other_stuff::do_something_else() {
    // code ...
}
```

generator.h

```
template <typename T>
struct generator {
    // code ...
};
```



math.h

```
template <typename T>
bool is_power_of_two(T x) {
    // code ...
}
```

other_stuff.cpp

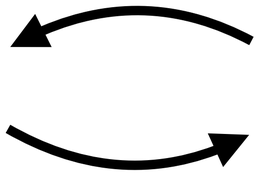
```
#define _NDEBUG
#include "mylib/generator.h"
#include "mylib/math.h"
#include "other_stuff.h"

void other_stuff::do_something_else() {
    // code ...
}
```

coroutine_stuff.cpp

```
#include "mylib/math.h"
#include "mylib/generator.h"
#include "coroutine_stuff.h"

void coroutine_stuff::do_something() {
    // code ...
};
```

generator.h

```
#define YES true

template <typename T>
struct generator {
    // code ...
};
```



math.h

```
template <typename T>
bool is_power_of_two(T x) {
    // code ...
}
```

other_stuff.cpp

```
#define _NDEBUG
#include "mylib/generator.h"
#include "mylib/math.h"
#include "other_stuff.h"

void other_stuff::do_something_else() {
    // code ...
}
```

coroutine_stuff.cpp

```
#include "mylib/math.h"
#include "mylib/generator.h"
#include "coroutine_stuff.h"

void coroutine_stuff::do_something() {
    // code ...
};
```

generator.h

```
#define YES true

template <typename T>
struct generator {
    // code ...
};
```

generator.h

```
#define YES true

template <typename T>
struct generator {
    // code ...
};

struct helper {
    // code ...
};
```

generator.h

```
#define YES true

template <typename T>
struct generator {
    // code ...
};

struct __helper {
    // code ...
};
```

generator.h

```
#define YES true

template <typename T>
struct generator {
    // code ...
};

namespace detail {
    struct helper {
        // code ...
    }
};
```

modules

generator.cppm

```
export module generator;

template <typename T>
export struct generator {
    // code ...
};

struct helper {
    // code ...
};
```

generator.cppm

```
export module generator;

template <typename T>
export struct generator {
    // code ...
};

struct helper {
    // code ...
};
```

generator.bmi

generator.cppm

```
export module generator;

template <typename T>
export struct generator {
    // code ...
};

struct helper {
    // code ...
};
```

generator.bmi

coroutine_stuff.cpp

```
export module coroutine_stuff;

import generator;
import math;

// stuff...
```

long-awaited fixes

```
int main() {
    std::vector v = {7, 3, 2, 4, 9};

    // TODO: remove all odd numbers




    for (int i : v)
        std::cout << i << '\n';
}
```

```
int main() {
    std::vector v = {7, 3, 2, 4, 9};

    v.erase(std::remove_if(v.begin(),
                           v.end(),
                           [](int i) { return i % 2 == 1; }),
            v.end());

    for (int i : v)
        std::cout << i << '\n';
}

// Output:
// 2
// 4
```

```cpp
int main() {
    std::vector v = {7, 3, 2, 4, 9};

    std::erase(v, [](int i) { return i % 2 == 1; }));

    for (int i : v)
        std::cout << i << '\n';
}

// Output:
// 2
// 4
```

How C++20 changes the way we write code

Timur Doumler

@timur_audio

CppCon
18 September 2020