TDDD38/726G82 - Advanced programming in C++

C++ 20

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Introduction

What is in C++20?

The big four:

- **Concepts**
- Coroutines
- Modules
- **Ranges**

Introduction

What is in C++20?

Other minor things:

- Three-way comparison operator
- Extended template parameters
- Lambda improvements
- `constinit` and `consteval`
- Various language and library features
- More concurrency primitives

LINKÖPING UNIVERSITY

Concepts

Example

```
template <typename T>
auto remainder(T a, T b)
{
  return a % b;
}
```

Concepts

Example

```
template <typename T>
auto remainder(T a, T b)
{
  return a % b;
}

template <typename T>
auto remainder(T a, T b)
{
  return std::fmod(a, b);
}
```

Concepts

Example

```
example.cc:11:6: error: redefinition of
`template<class T> auto remainder(T, T)`
   11 | auto remainder(T a, T b)
      |      ^~~~~~~~~
example.cc:5:6: note: `template<class T> auto remainder(T, T)`
previously declared here
    5 | auto remainder(T a, T b)
      |      ^~~~~~~~~
```

Concepts

Example

```
template <typename T>
auto remainder(T a, T b)
    -> std::enable_if_t<std::is_integral_v<T>,
                        decltype(a % b)>
{
  return a % b;
}

template <typename T>
auto remainder(T a, T b)
    -> std::enable_if_t<std::is_floating_point_v<T>,
                        decltype(std::fmod(a, b))>
{
  return std::fmod(a, b);
}
```

Concepts

Enter `requires` clauses!

```
// requires after template header
template <typename T> requires std::is_integral_v<T>
auto remainder(T a, T b)
{
  return a % b;
}

// requires after function header
template <typename T>
auto remainder(T a, T b) requires std::is_floating_point_v<T>
{
  return std::fmod(a, b);
}
```

Concepts

Enter `requires` clauses!

```
// requires after template header
template <typename T>
  requires std::is_integral_v<T> || std::is_convertible_v<T, int>
auto remainder(T a, T b)
{
  return a % b;
}

// requires after function header
template <typename T>
auto remainder(T a, T b)
  requires std::is_floating_point_v<T> || std::is_convertible_v<T, double>
{
  return std::fmod(a, b);
}
```

Concepts

Even simpler with Concepts

```
template <typename T>
auto remainder(T a, T b)
  requires std::integral<T>
{
  return a % b;
}

template <typename T>
auto remainder(T a, T b)
  requires std::floating_point<T>
{
  return std::fmod(a, b);
}
```

Concepts

A simpler way to use concepts

```
template <std::integral T>
auto remainder(T a, T b)
{
  return a % b;
}

template <std::floating_point T>
auto remainder(T a, T b)
{
  return std::fmod(a, b);
}
```

Concepts

Implementation of `std::integral` and `std::floating_point`

```
namespace std
{
  template <typename T>
  concept integral = std::is_integral_v<T>;

  template <typename T>
  concept floating_point = std::is_floating_point_v<T>;
}
```

Concepts

Creating our own concepts

```
template <typename T>
concept like_int = std::integral<T> ||
                   std::is_convertible_v<T, int>;

template <like_int T>
concept remainder(T a, T b)
{
  return a % b;
}
```

Concepts

Another example

```
template <has_at T>
auto&& get(T&& t, int i)
{
  return t.at(i);
}

template <has_iterator T> requires (!has_at<T>)
auto&& get(T&& t, int i)
{
  return *std::next(std::begin(t), i);
}
```

Concepts

Implementing `has_at` and `has_iterator`

```
template <typename T>
concept has_at = requires(T t, int i)
{
  t.at(i);
};

template <typename T>
concept has_iterator = requires(T t)
{
  { std::begin(t) } -> std::input_iterator;
};
```

Concepts

Even simpler implementation

```
template <typename T>
concept has_at = requires(T t, int i)
{
  t.at(i);
};

template <typename T>
concept has_iterator = requires(T t)
{
  { std::begin(t) } -> std::input_iterator;
};

template <has_iterator T>
auto&& get(T&& t, int i)
{
  if constexpr (has_at<T>)
    return t.at(i);
  else
    return *std::next(std::begin(t), i);
}
```

Concepts

How did we do this before C++ 20?

```
template <typename T>
auto get_helper(T&& t, int i, int)
  -> decltype(( t.at(i) ))
{
  return t.at(i);
}

template <typename T>
auto get_helper(T&& t, int i, long)
  -> decltype(( *std::next(std::begin(t), i) ))
{
  return *std::next(std::begin(t), i);
}

template <typename T>
auto&& get(T&& t, int i)
{
  return get_helper(std::forward<T>(t), i, 0);
}
```

Concepts

Requires expression in the wild

```
template <typename T>
void print(std::ostream& os, T const& data)
  requires std::is_class_v<T> &&
           requires { os << data; }
{
  os << data;
  if (requires { T{data}; })
  {
    os << " (copyable)";
  }
}
```

Concepts

Concepts + classes = ♥

```
template <typename From, typename To>
concept convertible_to = std::is_convertible_v<From, To>;

template <convertible_to<int> T>
  requires std::copy_constructible<T>
class Cls
{
};
```

LINKÖPING UNIVERSITY

Ranges

Example

```
using namespace std;

vector<int> v { 3, 1, -5, 4 };
sort(begin(v), end(v));
copy(begin(v), end(v),
     ostream_iterator<int>{cout, " "});
```

Ranges

Same example in C++ 20

```
using namespace std;

vector<int> v { 3, 1, -5, 4 };
ranges::sort(v);
ranges::copy(v, ostream_iterator<int>{cout, " "});
```

Ranges

What are ranges?

```
template <typename T>
concept range = requires(T& t)
{
  std::begin(t);
  std::end  (t);
};
```

Ranges

Let's look at an example from before C++ 20

```
struct Point
{
  int x;
  int y;
};

int main()
{
  std::vector<Point> points { /* ... */ };

  auto filter = [](Point p) { return p.x < 0 || p.y < 0; };

  points.erase(
    std::remove_if(std::begin(points), std::end(points), filter),
    std::end(points));

  auto printer = [](Point p) { cout << p.x << ", " << p.y << endl; };

  std::for_each(std::begin(points), std::end(points), printer);
}
```

Ranges

Enter range adaptors!

```
struct Point
{
  int x;
  int y;
};

using namespace std;

int main()
{
  vector<Point> const points { /* ... */ };

  auto filter = [](Point p) { return p.x >= 0 && p.y >= 0; };

  auto result = points | ranges::views::filter(filter);
  for (Point p : result)
  {
    cout << p.x << ", " << p.y << endl;
  }

}
```

Ranges

Let’s break it down

- **views**
- range adaptors

Ranges

Views

```
list<int> my_list { 1, 2, 3, 4, 5, 6 };
```

my_list						
	1	2	3	4	5	6

Ranges

Views

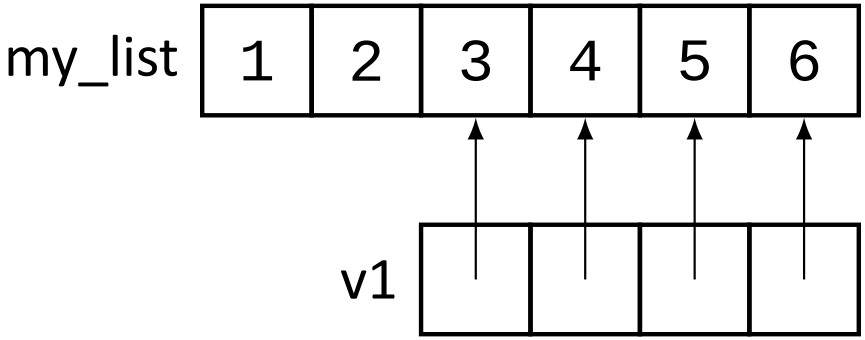
```
auto v1 = ranges::views::drop(my_list, 2);
```

my_list	1	2	3	4	5	6

Ranges

Views

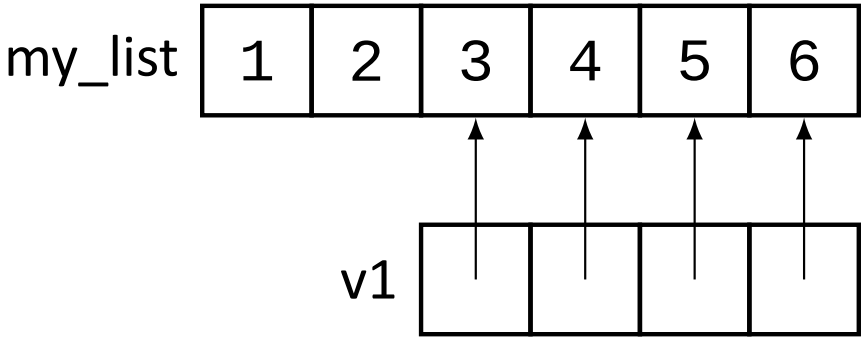
```
auto v1 = ranges::views::drop(my_list, 2);
```



Ranges

Views

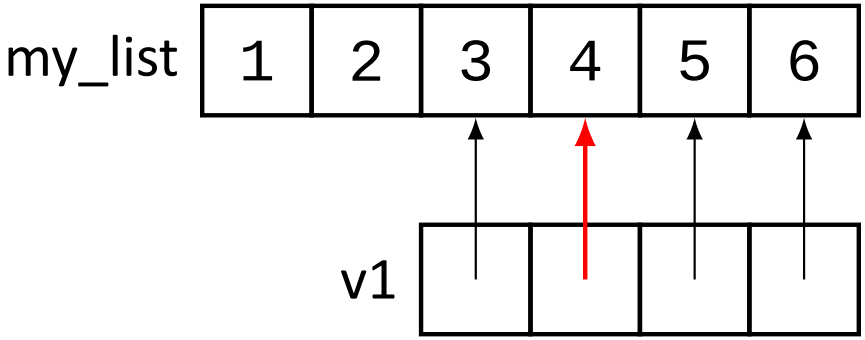
```
auto f = [](int x) { return x % 2 == 0; };
auto v2 = ranges::views::filter(v1, f);
```



Ranges

Views

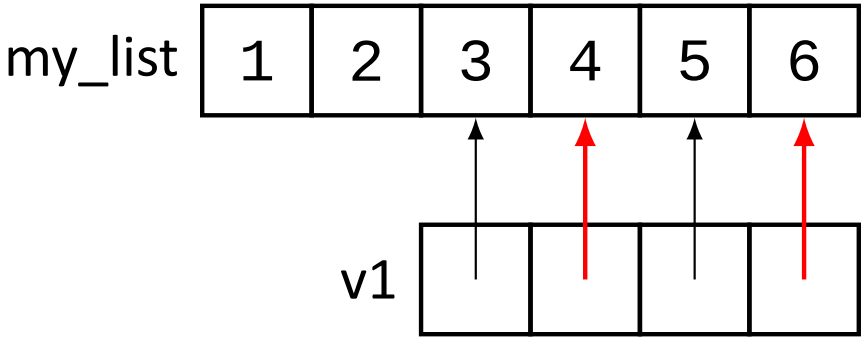
```
auto f = [](int x) { return x % 2 == 0; };
auto v2 = ranges::views::filter(v1, f);
```



Ranges

Views

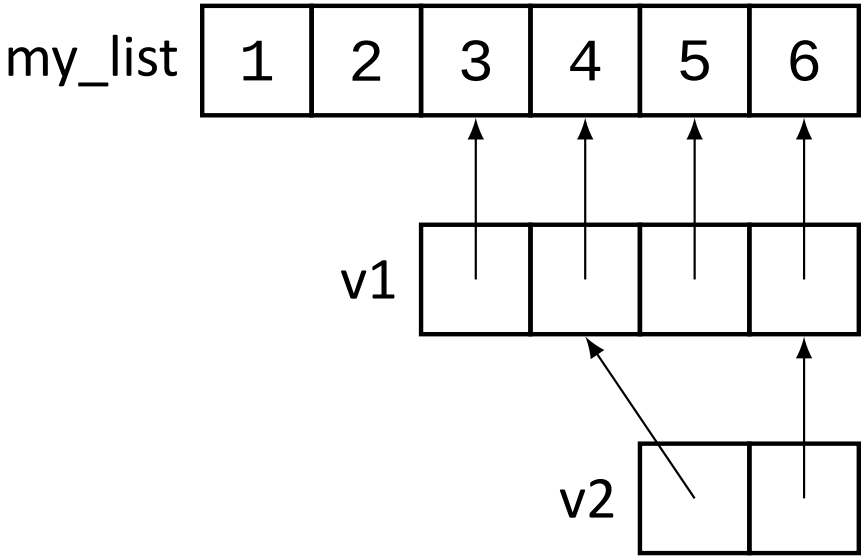
```
auto f = [](int x) { return x % 2 == 0; };
auto v2 = ranges::views::filter(v1, f);
```



Ranges

Views

```
auto f = [](int x) { return x % 2 == 0; };
auto v2 = ranges::views::filter(v1, f);
```



Ranges

Lazy computation of views

```
auto v1 = std::ranges::views::iota(1);
for (int i : v1)
{
  cout << i << endl;
}
```

Ranges

Lazy computation of views

```
auto v1 = std::ranges::views::iota(1);
for (int i : v1)
{
  cout << i << endl;
}
```

Infinite loop!

Ranges

Lazy computation of views

```
auto even = [](int x) { return x % 2 == 0; };

auto v1 = std::ranges::views::iota(1);
auto v2 = std::ranges::views::filter(v1, even);
for (int i : v2)
{
  cout << i << endl;
}
```

Ranges

Lazy computation of views

```
auto even = [](int x) { return x % 2 == 0; };

auto v1 = std::ranges::views::iota(1);
auto v2 = std::ranges::views::filter(v1, even);
for (int i : v2)
{
  cout << i << endl;
}
```

Infinite loop!

Ranges

Lazy computation of views

```
auto even = [](int x) { return x % 2 == 0; };

auto v1 = std::ranges::views::iota(1);
auto v2 = std::ranges::views::filter(v1, even);
auto v3 = std::ranges::views::take(v2, 4);
for (int i : v3)
{
  cout << i << endl;
}
```

Ranges

Lazy computation of views

```
auto even = [](int x) { return x % 2 == 0; };

auto v1 = std::ranges::views::iota(1);
auto v2 = std::ranges::views::filter(v1, even);
auto v3 = std::ranges::views::take(v2, 4);
for (int i : v3)
{
  cout << i << endl;
}
```

Halts!

Ranges

Let’s break it down

- views
- **range adaptors**

Ranges

Range adaptors!

```
using namespace std::ranges;

auto even = [](int x) { return x % 2 == 0; };
auto inc  = [](int x) { return x + 1; };

auto v1 = views::iota(1);
auto v2 = views::filter(v1, even);
auto v3 = views::transform(v2, inc);
auto v4 = views::take(v3, 10);
for (int i : v4)
{
  cout << i << endl;
}
```

Ranges

Range adaptors!

```
using namespace std::ranges;

auto even = [](int x) { return x % 2 == 0; };
auto inc  = [](int x) { return x + 1; };

auto v = views::iota(1)
         | views::filter(even)
         | views::transform(inc)
         | views::take(10);
for (int i : v)
{
  cout << i << endl;
}
```

Ranges

Let’s not forget that containers are ranges too!

```
using namespace std;
using namespace std::ranges;

map<string, int> m { /* ... */ };

auto fun = [](pair<string, int> p)
           {
             return p.first + ": " + to_string(p.second);
           };

ranges::copy(m | views::transform(fun) | views::reverse,
     ostream_iterator<string>{cout, "\n"});

vector<pair<string, int>> v { begin(m), end(m) };

// with lambda
ranges::sort(v, [](auto a, auto b) { return a.second < b.second; });

ranges::copy(v | views::transform(fun),
             ostream_iterator<string>{cout, "\n"});
```

Ranges

Let’s not forget that containers are ranges too!

```
using namespace std;
using namespace std::ranges;

map<string, int> m { /* ... */ };

auto fun = [](pair<string, int> p)
           {
             return p.first + ": " + to_string(p.second);
           };

ranges::copy(m | views::transform(fun) | views::reverse,
     ostream_iterator<string>{cout, "\n"});

vector<pair<string, int>> v { begin(m), end(m) };

// with std::less and projection
ranges::sort(v, less<int>{}, &pair<string, int>::second);

ranges::copy(v | views::transform(fun),
             ostream_iterator<string>{cout, "\n"});
```

Ranges

Now we have the tools to understand the previous example!

```
struct Point
{
  int x;
  int y;
};

using namespace std;

int main()
{
  vector<Point> const points { /* ... */ };

  auto filter = [](Point p) { return p.x >= 0 && p.y >= 0; };

  auto result = points | ranges::views::filter(filter);
  for (Point p : result)
  {
    cout << p.x << ", " << p.y << endl;
  }

}
```

LINKÖPING UNIVERSITY

Modules

Pre C++ 20

```cpp
// math.h
#ifndef MATH_H
#define MATH_H

#include <vector>
#include <numeric>

template <typename T>
T sum(std::vector<T> v)
{
  return std::accumulate(
    v.begin(), v.end(), 0);
}
template <typename T>
T average(std::vector<T> v)
{
  return sum(v) / v.size();
}

#endif//MATH_H
```

```cpp
// main.cc
#include "math.h"

#include <vector>
#include <iostream>

int main()
{
  std::vector<float> v { /* ... */ };
  std::cout << average(v) << std::endl;
}
```

```cpp
// other.cc
#include "math.h"

#include <vector>

// does something with math.h
```

Modules

Enter Modules!

```
// math.cc
export module math;

import <vector>;
import <numeric>;
using namespace std;

template <typename T>
T sum(vector<T> v)
{
  return accumulate(
    v.begin(), v.end(), 0);
}

export
template <typename T>
T average(vector<T> v)
{
  return sum(v) / v.size();
}
```

```
// main.cc
import math;

import <vector>;
import <iostream>;

int main()
{
  std::vector<float> v { /* ... */ };
  std::cout << average(v) << std::endl;
}
```

```
// other.cc
import math;

import <vector>;

// does something with math
```

Modules

Enter Modules!

```
// math.cc
export module math;

import <vector>;
import <numeric>;
using namespace std;

template <typename T>
T sum(vector<T> v)
{
  return accumulate(
    v.begin(), v.end(), 0);
}

export
template <typename T>
T average(vector<T> v)
{
  return sum(v) / v.size();
}
```

```
// main.cc
import math;

import <vector>;
import <iostream>;

int main()
{
  std::vector<float> v { /* ... */ };
  std::cout << average(v) << std::endl;
}
```

```
// other.cc
import math;

import <vector>;

// does something with math
```

Modules

Enter Modules!

```
// math.cc
export module math;

import <vector>;
import <numeric>;
using namespace std;

template <typename T>
T sum(vector<T> v)
{
  return accumulate(
    v.begin(), v.end(), 0);
}

export
template <typename T>
T average(vector<T> v)
{
  return sum(v) / v.size();
}
```

```
// main.cc
import math;

import <vector>;
import <iostream>;

int main()
{
  std::vector<float> v { /* ... */ };
  std::cout << average(v) << std::endl;
}
```

```
// other.cc
import math;

import <vector>;

// does something with math
```

Modules

Enter Modules!

```
// math.cc
export module math;

import <vector>;
import <numeric>;
using namespace std;

template <typename T>
T sum(vector<T> v)
{
  return accumulate(
    v.begin(), v.end(), 0);
}

export
template <typename T>
T average(vector<T> v)
{
  return sum(v) / v.size();
}
```

```
// main.cc
import math;

import <vector>;
import <iostream>;

int main()
{
  std::vector<float> v { /* ... */ };
  std::cout << average(v) << std::endl;
}
```

```
// other.cc
import math;

import <vector>;

// does something with math
```

LINKÖPING UNIVERSITY

Coroutines

"Generator" function

```
vector<int> generate_sequence(int start, int count)
{
  vector<int> result (count, 0);
  for (int i { start }; i < start + count; ++i)
  {
    result.push_back(i);
  }
  return result;
}

int main()
{
  for (int i : generate_sequence(1, 10))
  {
    cout << i << endl;
  }
}
```

Coroutines

Enter Coroutines!

```
generator<int> generate_sequence(int start, int count)
{
  for (int i { start }; i < start + count; ++i)
  {
    yield i;
  }
}

int main()
{
  for (int i : generate_sequence(1, 10))
  {
    cout << i << endl;
  }
}
```

Note: This is not real C++

Coroutines

Enter Coroutines!

```
generator<int> generate_sequence(int start, int count)
{
  for (int i { start }; i < start + count; ++i)
  {
    yield i;
  }
}

int main()
{
  for (int i : generate_sequence(1, 10))
  {
    cout << i << endl;
  }
}
```

Note: This is not real C++

Coroutines

Enter Coroutines!

```
generator<int> generate_sequence(int start, int count)
{
  for (int i { start }; i < start + count; ++i)
  {
    yield i;
  }
}

int main()
{
  for (int i : generate_sequence(1, 10))
  {
    cout << i << endl;
  }
}
```

Note: This is not real C++

Coroutines

Coroutine handlers

```
int main()
{
  auto g = generate(1, 10);
  for (int i : g)
  {
    cout << i << endl;
  }
}
```

Note: This is not real C++

Coroutines

Coroutine handlers

```
int main()
{
  auto g = generate(1, 10);
  while (!g.done())
  {
    cout << g() << endl;
  }
}
```

Note: This is not real C++

Coroutines

Coroutine handlers

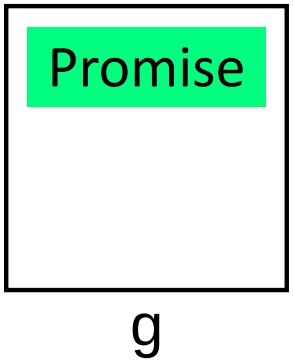
```
int main()
{
  auto g = generate(1, 10);
  while (!g.done())
  {
    cout << g() << endl;
  }
}
```

Note: This is not real C++

Coroutines

Coroutine handlers

```
int main()
{
  auto g = generate(1, 10);
  while (!g.done())
  {
    cout << g() << endl;
  }
}
```

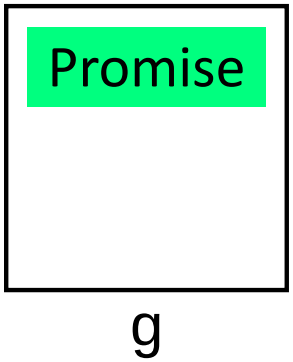


Note: This is not real C++

Coroutines

Coroutine handlers

```
int main()
{
  auto g = generate(1, 10);
  while (!g.done())
  {
    cout << g() << endl;
  }
}
```

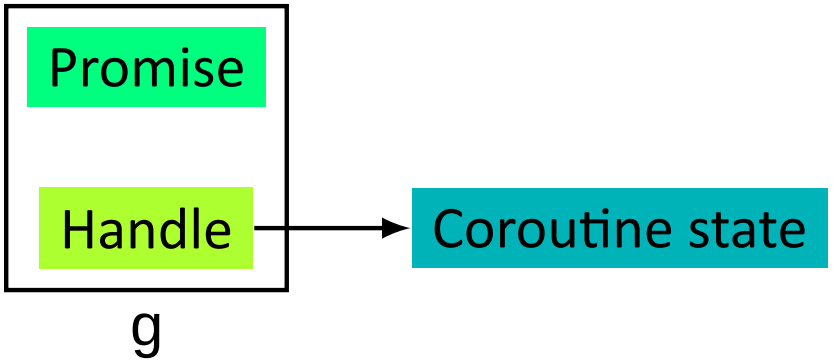


Note: This is not real C++

Coroutines

Coroutine handlers

```
int main()
{
  auto g = generate(1, 10);
  while (!g.done())
  {
    cout << g() << endl;
  }
}
```

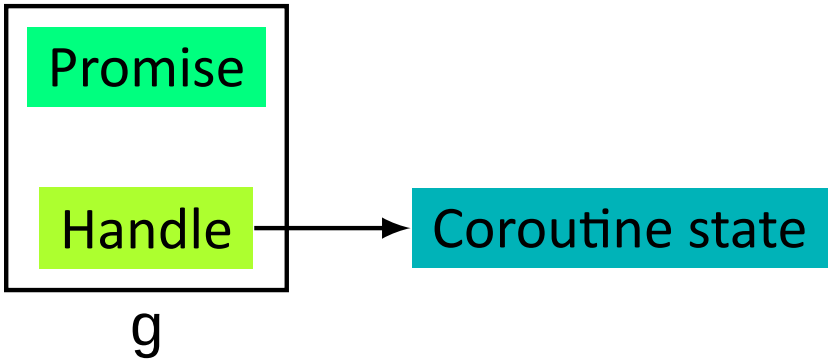


Note: This is not real C++

Coroutines

Coroutine handlers

```
int main()
{
  auto g = generate(1, 10);
  while (!g.done())
  {
    cout << g() << endl;
  }
}
```

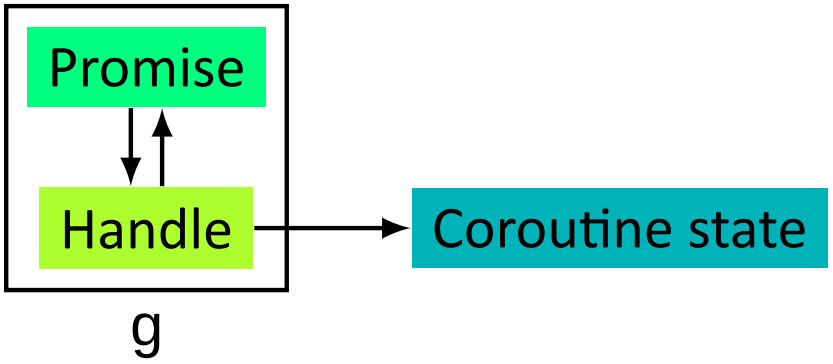


Note: This is not real C++

Coroutines

Coroutine handlers

```
int main()
{
  auto g = generate(1, 10);
  while (!g.done())
  {
    cout << g() << endl;
  }
}
```



Note: This is not real C++

Coroutines

(Almost) Real coroutines

```
generator<int> generate_sequence(int start, int count)
{
  for (int i { start }; i < start + count; ++i)
  {
    co_yield i;
  }
  co_return;
}
```

www.liu.se

LINKÖPING
UNIVERSITY