TDDD38/726G82 - Advanced programming in C++

C++ 20

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

1 Introduction
2 Concepts
3 Ranges
4 Modules
5 Coroutines

Introduction

What is in C++20?

The big four:

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

Introduction

What is in C++20?

- In this seminar we will only cover *concepts* and *ranges* in detail.
- Modules are not particularly useful today. Those compilers that have partial support are still very buggy and hard to use.
- Coroutines are supported in most compilers, *but* they will not be especially useful until C++23 (at the earliest).
- There are links published with these slides where you can read more.

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

Concepts

Example

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

Concepts

Example

- `remainder` only works for *integral types* (`int`, `bool`, `char`, `long` etc.).
- Floating point numbers (`float`, `double` and `long double`) doesn't support `operator%`.
- However, the concept of remainders is still applicable to floating point numbers.
- In `<cmath>` there is the function `std::fmod` that calculates the remainder for floating point numbers.

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

- This will unfortunately not work...
- We have two equally valid overloads of `remainder`
- Any call to `remainder` will therefore be ambigous

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

Example

- This works, but is very terse.
- Since we are forced to use trailing return types for our SFINAE usage, we can't use `auto` as return type anymore.
- So we have to duplicate our return statement in a `decltype` statement if we want the compiler to deduce the right return type.
- If `T` is something other than an integral or floating point type, the errors are horribly unhelpful.

Concepts

Enter `requires`

```
// 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`

- `requires` is a new keyword that specifies requirements on function templates.
- It can be placed immediately after the template header,
- Or immediately after the function header.
- Requirements are expressed as `bool` expressions,
- which means that we can use `&&` and `||` to chain multiple requirements together.

Concepts

Enter `requires`

```
// 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

Enter `requires`

- Now we are checking that `T` is either an integral/floating point type,
- **or** convertible to `int` and `double` respectively.
- This works since `requires` just expects a `bool` expression.

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

Even simpler with Concepts

- Concepts are a collection of requirements bundled together into one entity, called a *concept*.
- Concepts are used to induce requirements on template parameters. They also act as `bool` conditions.
- The standard library provides several pre-defined concepts in `<concepts>`,
- For example: `std::integral` and `std::floating_point`.
- But you can also make your own.

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

A simpler way to to use concepts

- We can replace `typename` in template parameter declarations with a specific concept instead.
- This means that whatever that template parameter is instantiated as, it must fulfill the requirements specified by the concept.

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 own concepts

- You create your own concept with the `concept` keyword.
- Each concept must take template parameters which are then constrained.
- After the concept name there must be a = that are then followed by the constraints, which are all `bool` expressions.

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

Creating our own concepts

- We can join multiple constraints together with `&&` and `||`, which allows for more complicated concepts.
- We should also note that concepts acts as `bool` conditions, so we can also use those to define other, more constrained concepts.
- In this case we are saying that `T` must either fulfill `std::integral`, or it must be convertible to `int` in order to fulfill the `like_int` concept.
- Let's look at another example:

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

Another example

- In this example we want to make a `get` function that takes a container and an index and returns the element at the corresponding index in the container. The elements are returned as the most appropriate reference type.
- If the container has an `at` function, we want to use that, otherwise we want to use iterators. Note that the iterator case only occurs if we *don't* have an `at` function due to the requirement (`!has_at<T>`).
- We need the `has_at` and `has_iterator` concept.

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

Implementing `has_at` and `has_iterator`

- A concept can also use a *requires clause*,
- which is a way for us to specify requirements on the interface of the passed in types.
- This is done by checking that certain expressions are valid for the type.
- These expressions are never actually evaluated, they are just examined by the compiler to check if they are valid for the specified type.

Concepts

Implementing `has_at` and `has_iterator`

- In order to make such checks we need access to objects that are involved in the expression we are checking.
- Because of this, a requires clause can take parameters. We can add how many parameters we want, and they can be whatever type we want.
- These parameters are never actually created and we never pass anything to them. They are just there so we have access to symbolic objects that are used in the validation of the expressions.

Concepts

Implementing `has_at` and `has_iterator`

- This can be seen in the `has_at` concept where we take a `T` object called `t` and an `int` parameter called `i`. These parameters are never created, and we don't have to care about them when using our concept.
- These are then used in the body of the requires clause to check if we can call `t.at(i)`.
- This simply means that the compiler checks: does `T` have a member function which can be called on an instance of `T` with an `int` parameter?

Concepts

Implementing `has_at` and `has_iterator`

- We can also set requirements for the return type of an expression, which can be seen in `has_iterator`.
- We do this by wrapping our expression in curly braces, i.e. `{ std::begin(t) }` followed by an arrow (`->`).
- After this arrow we specify what concept the return type should fulfill.
- In this case we are using the concept `std::input_iterator` to check that `std::begin(t)` at least returns an *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

Even simpler implementation

- Concepts can also be evaluated as `bool` expressions.
- This means that we can also use them in if-statements, loop conditions, `static_assert` statements etc.
- So we can simplify our code even more by utilizing `constexpr`-if statements.
- Here we require `T` to have iterators, but then we check in the body if there is an `at` function. If so, we use that instead of iterators.

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

How did we do this before C++ 20?

- This is about the same amount of code as the C++ 20 version, but this is more error-prone and utilizes a lot of “hacks”.
- We also have a lot more power with concepts. Notice that we are unable to check if `std::begin(t)` returns an input iterator or not, but with concepts we can.
- In C++ 17 we have to make several overloads and we have to induce a priority, but with concepts we can just make one function overload and then specialize it with concepts. Much simpler.

Concepts

Requires clause 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

Requires clause in the wild

- A requires clause is just a `bool` expression, so we can use them outside the context of concepts.
- Parameters are optional for requires clauses.
- Here we are using a requires clause to constrain the template parameter `T`. We are first checking if `T` is a class, and then we make sure that we can print it with `operator`<<.
- Then we are using it in a normal if-statement to check if `T` has a copy constructor.

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
{
};
```

Concepts

Concepts + classes = ♥

- Of course, template parameters for class templates can also be constrained with the help of concepts.
- Notice here that our concept takes two parameters, `From` and `To`.
- When we constrain a template parameter with a concept the first one will always implicitly be the template parameter itself.
- All other template parameters to the concept must however be set.

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

Example

- In this example we sort a vector of integers and print the result to `std::cout`.
- This is quite a good usage of the STL.
- But, it is quite annoying having to write `begin(v)` and `end(v)` whenever we want to do something with the entire vector `v`.

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

Same example in C++ 20

- C++ 20 introduces a new concept called *ranges*.
- This allows us to call algorithms with entire containers instead of having to pass iterators everywhere.
- *ranges* are not supposed to replace iterators, but will instead complement them.

Ranges

What are ranges?

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

Ranges

What are ranges?

- A *range* is a type which we can call `std::begin` and `std::end` on. This means that we can think of ranges as objects that have an iterator pair. This means that for example all *containers* are ranges.
- All (most) algorithms now has an *iterator* version and a *range* version.
- So what's the big deal? Sure we get a bit cleaner syntax, but besides that?

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

Let's look at an example

- We remove all points were either `x` or `y` are negative and then print the remaining points. However, this could be done better for several reasons.
- For starters, it is very annoying having to write `std::begin(v)` and `std::end(v)` everytime we want to operate on the vector.
- But there is also a performance issue here: we have to remove elements from the container, which is quite slow. For this example wouldn't it be enough to just *not* print the negative values? No need for actual removal.

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

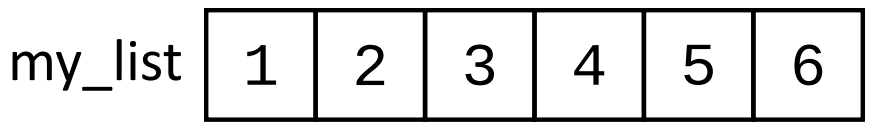
Let’s break it down

- **views**
- range adaptors

Ranges

Views

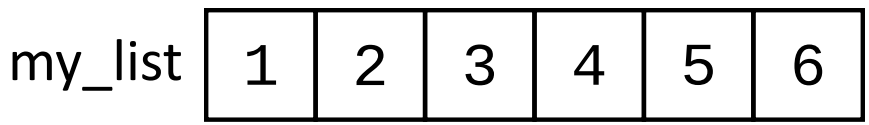
```
list<int> my_list { 1, 2, 3, 4, 5, 6 };
```



Ranges

Views

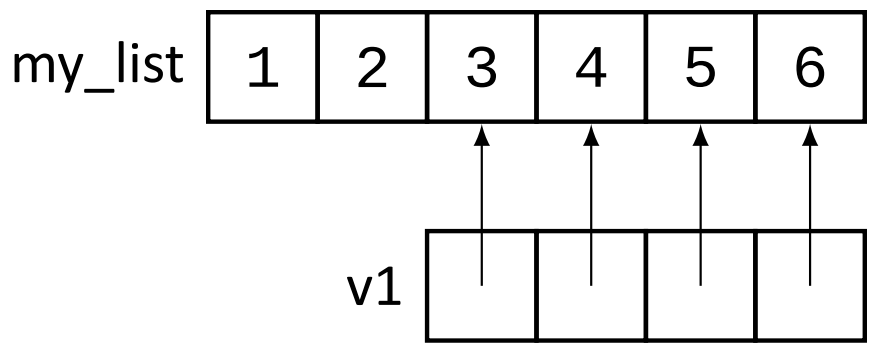
```
auto v1 = ranges::views::drop(my_list, 2);
```



Ranges

Views

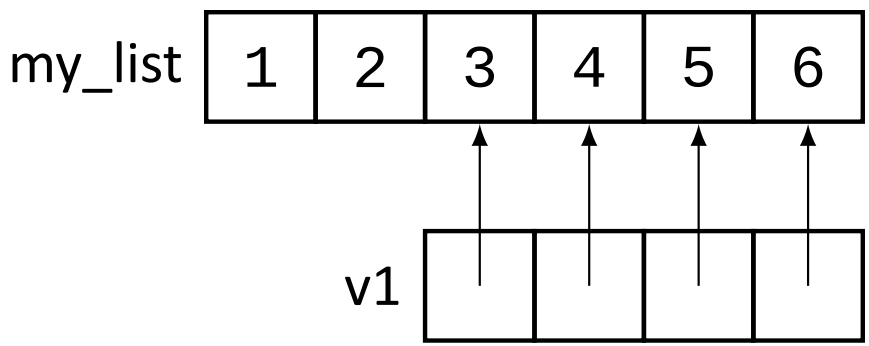
```
auto v1 = ranges::views::drop(my_list, 2);
```



Ranges

Views

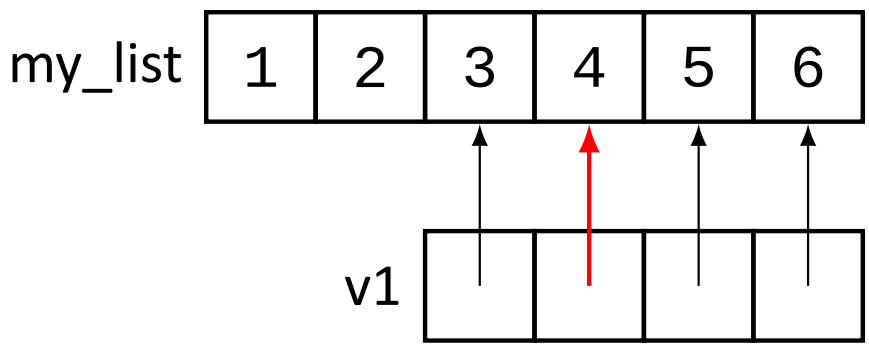
```
auto f = [](int x) { return x % 2 == 0; };
auto v2 = ranges::views::filter(v1, f);
```



Ranges

Views

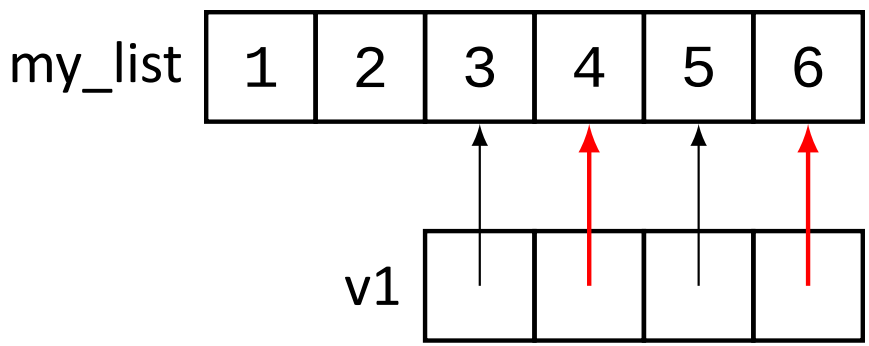
```
auto f = [](int x) { return x % 2 == 0; };
auto v2 = ranges::views::filter(v1, f);
```



Ranges

Views

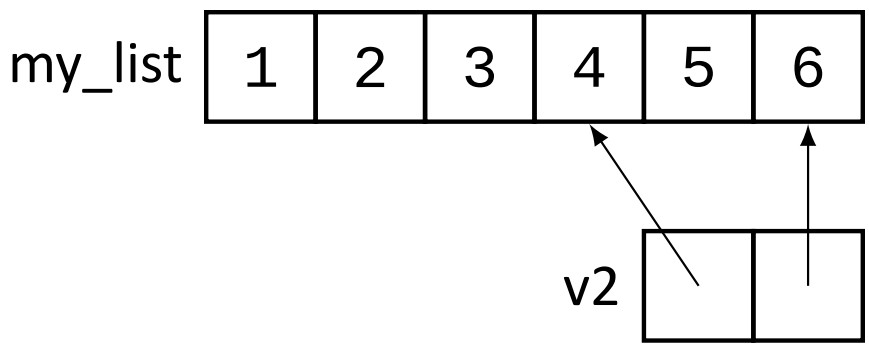
```
auto f = [](int x) { return x % 2 == 0; };
auto v2 = ranges::views::filter(v1, f);
```



Ranges

Views

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



Ranges

What?

- A *view* is a window into a subset of the elements in some range. It allows us to construct subranges procedurally.
- It's important to note that a view is much more powerful than just a pair of iterators. As we saw with `views::filter`, it can produce gaps from the original range.
- **Note:** a view never copies any elements, nor does it keep track of which elements it is pointing to (as the image might suggest). Instead, a view is computed lazily.

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

- `std::ranges::views::iota` is a view that *lazily* generates a sequence of numbers. In this case it will generate the numbers: `1, 2, 3, ...` and so on *forever*.
- This implies that views doesn't construct all its elements in memory, since `iota` will literally generate numbers forever!
- This means that it *must* generate each number as it is requested. This is what's meant when we say it is *lazy*.

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

- `v2` is also lazily evaluated, but this time we will filter out all values that are *not* even.
- This means that `v2` will generate the numbers `2, 4, 6, ...` and so on.
- What *filter* whenever we request a number from it, is that it will request a number `n` from `v1`. If `even(n)` returns `false` it will request a new number and repeat the process until it retrieves a number for which `even` returns `true`. That number will then be handed to the caller.

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

Lazy computation of views

- Finally, if we apply `take`, we will get a terminating loop.
- This is because `take(v2, 4)` will not deliver any more numbers after the user has requested 4 numbers.
- It is still done lazily though. So it doesn't know what the next number will be, all it knows is how many have been requested so far.
- In this case it will print: `2, 4, 6, 8` and then terminate.

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

Range adaptors!

- We've actually already seen a few *range adaptors*, for example `views::filter`, `views::take` and `views::drop`.
- A *range adaptor* takes a range and applies some kind of operation on it. For example: `views::transform` takes a range and applies a callable object on each element, thus *transforming* each requested value into a new value.

Ranges

Range adaptors!

- There are two ways to apply a range adaptor `C` to a range called `r`:
- The way we've already seen: `C(r, parameters...)`
- Or with `operator|` like this: `r | C(parameters...)`
- The advantage of `operator|` is that we can now chain them range adaptors together without having to create intermediate views. Like in the second example above.

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

Projections

- Some range algorithms, like `ranges::sort`, takes an optional *projection* parameter.
- A projection is pointer to some data member in the passed in class which is used for comparison.
- So instead of creating a lambda which compares based on the entire object, this allows us to use a comparison object that operates on that specific data member directly.

Modules

Pre C++ 20

```
// 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
```

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

#include <vector>
#include <iostream>

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

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

#include <vector>

// does something with math.h
```

Modules

Pre C++ 20

- They way we've been compiling large projects has been essentially the same since the inception of C. It is an *archaic* build process, meaning compile times are slower than they need to be.
- Since `#include` is just a copy-paste idiom, the content of `math.h` will be included in both `main.cc` and `other.cc`, meaning the content of `math.h` is compiled twice. This can be somewhat mitigated by just having declarations in the header file, but when we are dealing with templates this is impossible.

Modules

Pre C++ 20

- When we include library headers like `<iostream>` we are therefore *forced* to recompile the entire content of the file every time we include it in a translation unit.
- Strictly speaking we can *at most* include a header once in each translation unit. Due to this we need header guards, to make sure we only include each header once in every translation unit.
- This means that it is *our* responsibility to make sure that everything is included properly.

Modules

Pre C++ 20

- Because of the simplicity of the include model, there is a greater responsibility on us as programmers. But also, there are negative efftects on the compile times.
- For example, on my system `#include <iostream>` includes 18750 lines of code that the compiler must process *each time it is included*. This adds up quickly.
- Wouldn't it be better if `<iostream>` was pre-compiled? This is one of the problems modules aim to solve.

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
```

Modules

Explanation

- Standard headers can be *imported* as *Header Units*, for example: `import <vector>;`. This is functionally equivalent to including the headers, but with the difference that once a header has been imported *anywhere* in the code, it will not be included again.
- Instead of dealing with header files, we can create a *Module Implementation file* where we *export* our module by giving it a name.

Modules

Explanation

- The act of *exporting* something means we are making it available outside of the module. Thus anything not exported will be invisible in any other context. This means that we can have entities that are hidden from the API of the module. For example the `sum` function.
- When we import our exported modules we will automatically have access to everything that was exported.
- Since our module is named, we don't have to refer to any files. Instead we can just refer to the module name.

Modules

Explanation

- How we compile projects with modules is dependent on the compiler.
- Currently `Microsoft Visual Studio`, followed by `clang` has the best support for modules. `g++` only has experimental support for modules. If you want to try it out yourself, I have linked some articles that goes into more depth on this.

Modules

Explanation

Some observations:

- In module files it is totally fine to do `using namespace std;` since it will not leak into the users code.
- **Warning:** You cannot mix includes and import in a module file. You should stick to imports if possible. If you absolutely need to include files there is something called the *Global Module Fragment* dedicated for such things.

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

"Generator" function

- A generator function is something that constructs (or generates) a sequence of values.
- In this example, `get_sequence` is a function that generates consecutive integers.
- This is a useful abstraction because we can then think of the whole sequence as a unit, while still only processing one value at a time.
- We saw a good way to do this with *ranges* earlier, but can it be done with functions?

Coroutines

"Generator" function

- This is one way to do it with functions.
- But it is quite inefficient, because we have to generate all values *eagerly*. This means we have to generate all values and store them in some kind of container.
- The reason why we have to do this is because we can only retrieve a value from a function once it is done.
- But what if a function could pause its execution and gives us a partial result, and then continue where it left of later on?

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

Enter Coroutines!

- A *coroutine* is a generalized function that has some extended features.
- Normally all a function can do to transfer the execution back to its caller is by returning.
- But coroutines introduces other ways to do so. In this example we are *yielding* a value from the coroutine.
- This means that the function is paused with its current state stored away somewhere, and control is given back to whoever called this function. In this case, `main`.

Coroutines

Enter Coroutines!

- From the callers point-of-view this coroutine behaves like a function: we requested a value, and we got a value.
- But the difference is that when we call the coroutine *again*, it will continue where we left of, still keeping track of its state. This means that we can *lazily* generate values from the coroutine.
- So no need for generating all the results before returning to the caller.

Coroutines

Enter Coroutines!

- But what we'll notice here is that the return type of this coroutine is set as `generator<int>`, but we are yielding `int`? And we are not even returning anything, so why does it have a return type?
- Well, the problem with coroutines is that they are just an abstraction built on top of normal functions. This means that whenever we yield from the coroutine the caller must receive *something* from the caller that tells the caller what is going on.

Coroutines

Enter Coroutines!

- C++ doesn't want to force you to do this in one specific way, so because of this each coroutine must have a *handler*. In this case our handler is `generator<int>`.
- This is an object that will *receive* yielded values from the coroutine. This is what allows us to loop over the results of the coroutines, even though the coroutine itself never actually returns anything.
- Let's look at a diagram:

Coroutines

Coroutine handlers

Let's rewrite this code to make it easier to understand.

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

Note: This is not real C++

Coroutines

Coroutine handlers

Notice that we only “call” the coroutine once. However, note that the coroutine isn’t executed yet.

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

Note: This is not real C++

Coroutines

Corout

The object which gets returned from the coroutine is called a *handler*. The coroutine is executed by calling this object.

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

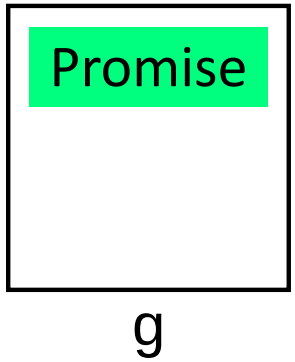
Note: This is not real C++

Coroutines

Coroutine handlers

The handler g has a “Promise” which is where the coroutine will put the yielded value.

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



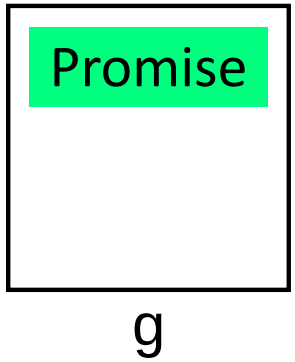
Note: This is not real C++

Coroutines

Coroutine handlers

We can then check: is this “call” to the coroutine still active?

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



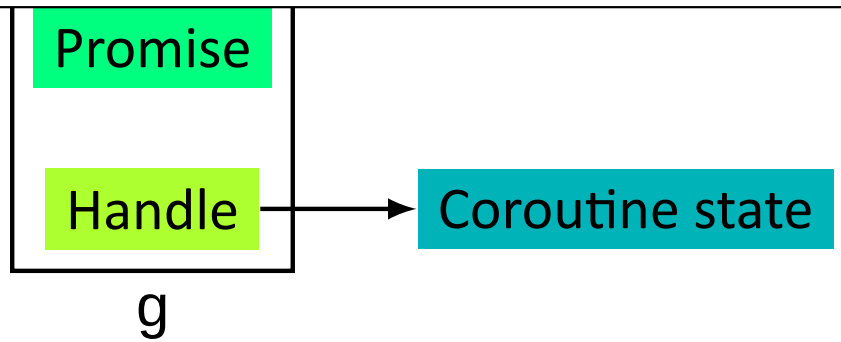
Note: This is not real C++

Coroutines

Corout

For this to work, g needs to keep track of the current state of the coroutine. Otherwise it can't tell if the coroutine is done or not.

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



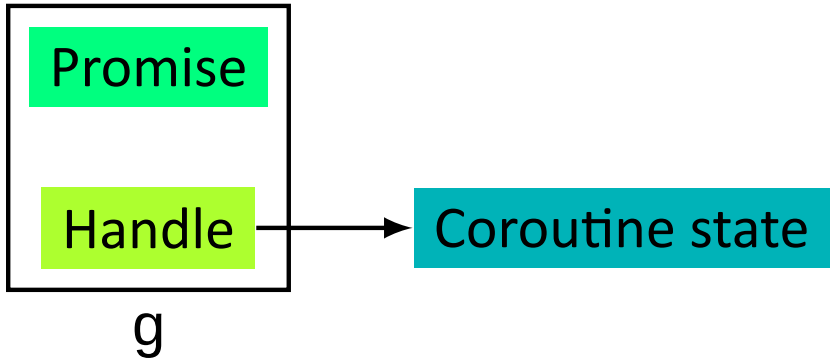
Note: This is not real C++

Coroutines

Coroutine handlers

Now we call it, thus finally executing the coroutine. This will yield us an integer value that is placed in the promise.

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



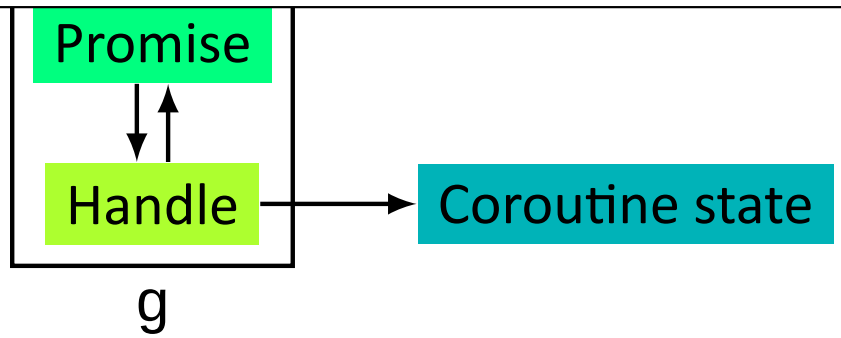
Note: This is not real C++

Coroutines

Corout

This means that the coroutine state and the promise must communicate. So when the coroutine yields, the handler will then know to place that value in the promise.

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



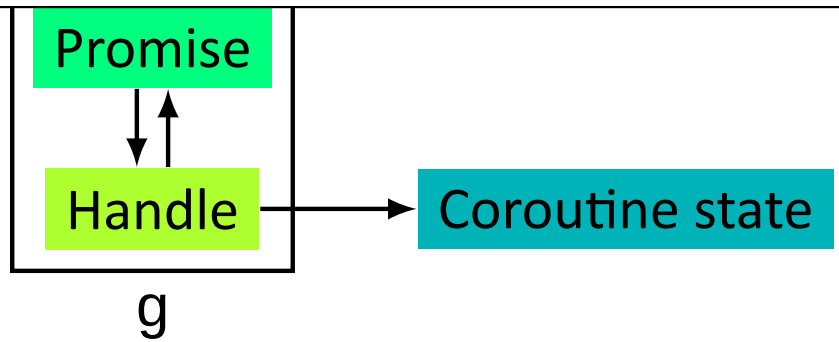
Note: This is not real C++

Coroutines

Corout

Once the coroutine *actually* returns, the promise will be empty, thus informing the caller that the coroutine is done.

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



Note: This is not real C++

Coroutines

Real coroutines

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

Coroutines

Real coroutines

- The behaviour of the returned handler can vary from case to case, so C++ gives us the opportunity to customize its behaviour for our specific coroutines.
- Coroutines must be handled seperately. Coroutines are not normal functions were the keyword `return` means exit the function. When we call a coroutine it doesn't even execute the code until the handler tells it to, so we have to make them distinct.
- Because of these reasons, coroutines uses their own keywords: `co_return`, `co_yield` and `co_await`.

Coroutines

Real coroutines

We will not discuss coroutines in more detail because:

- In order to use coroutines we must implement our own handlers because the standard library does not yet supply any standard handlers (maybe in C++23?).
- Writing your own handler is very hard, so it's recommended not to do that.
- But there are 3rd party libraries popping up that does implement different handlers, so we might be able to use them sooner than C++23!

www.liu.se

LINKÖPING UNIVERSITY