

What's new in C++20



3 | Ranges

3.1 | STL

STL

cppcon presentation Tristan Brindle

- STL has been unchanged for a 25 years
- Since it is/was good
- But the language C++ evolved
- STL 2.0
- Most new stuff in `std::ranges`²⁰

Thanks to the new concept^{AT} language feature

- STL has now *constrained* algorithms

Check out the following example:

```
1 std::list<int> list{2,3,1};  
2  
3 std::sort(list.begin(), list.end());
```

Is this correct?



Ranges 2



- It is not going to work
- sort needs random access iteration
- and a list does not provide that

What is the error the compiler gives for this slide of code?

442 lines of error messages (clang).

Let's try again:

```
1 std::list<int> list{2,3,1};  
2  
3 ranges::sort(list.begin(), list.end());
```

10 lines of error messages, pinpointing to the reason!



Ranges 3



Let's do it correctly:

```
1 std::vector<int> vec{2,3,1};
2
3 std::sort(vec.begin(), vec.end());
```

Or the new way of writing:

```
1 std::vector<int> vec{2,3,1};
2
3 std::ranges::sort(vec);
```



Ranges 4



3.2 | What is a range

What is a range

- calling `begin()` returns an iterator
- calling `end()` may return a *sentinel*

The job of the sentinel is to mark the end of a range.

- it should be comparable to an iterator
- it sometimes enables more efficient code generation

Example:

```
1 const std::string big_string = read_file();
2 // guaranteed to contain '\n'
3
4 auto get_newline_pos(const std::string &str)
5 {
6     return std::find(str.begin(), str.end(), '\n');
7 }
```



Ranges 5



Implementation of `find`:

```
1 template<typename I, typename Val>
2 I find(I first, I last, const Val &val)
3 {
4     while(first!=last)
5     {
6         if(*first==val)
7             break;
8         ++first;
9     }
10    return first;
11 }
```

Since we are sure we won't reach the end, this code is inefficient. But we cannot tell the compiler.



Ranges 6



But with ranges we can!

```
1 auto get_newline_pos(const std::string &str)
2 {
3     return ranges::find(str.begin(), ranges::unreachable_sentinel, '\n');
4 }
```



Ranges 7



3.3 | Projections

Projections

A projection is a

- Unary callable which may be passed to most algorithms
- modifies the view of the data that algorithms see



Ranges 8



Paycheck problem:

```
1 struct Employee {
2     std::string name;
3     int id;
4 };
5
6 struct Payslip {
7     std::string pay_info;
8     int employee_id;
9 };
10
11 std::vector<Employee> employees;
12 std::vector<Payslip> payslips;
```

We want to check:

- Every employer has exactly 1 payslip



Ranges 9



Solution:

- sort employees by id
- sort payslips by employee_id
- compare them by id



Ranges 10



Old way:

```
1 std::sort(employees.begin(), employees.end(),
2   [] (const Employee &x, const Employee &y) {
3       return x.id < y.id; });
4
5 std::sort(payslips.begin(), payslips.end(),
6   [] (const Payslip &x, const Payslip &y) {
7       return x.employee_id < y.employee_id; });
8
9 std::equal(employees.begin(), employees.end(),
10  payslips.begin(), payslips.end(),
11  [] (const Employee &e, const Payslip &p) {
12      return e.id==p.employee_id; });
```



Ranges 11



Using range based overloads:

```
1 std::ranges::sort(employees,
2   [] (const Employee &x, const Employee &y) {
3       return x.id < y.id; });
4
5 std::ranges::sort(payslips,
6   [] (const Payslip &x, const Payslip &y) {
7       return x.employee_id < y.employee_id; });
8
9 std::ranges::equal(employees, payslips,
10  [] (const Employee &e, const Payslip &p) {
11      return e.id==p.employee_id; });
```



Ranges 12



```
1 std::ranges::sort(employees,
2   [] (const Employee &x, const Employee &y) {
3       return x.id < y.id; });
```

Use a projection to project the Employee to its id

```
1 std::ranges::sort(employees, std::ranges::less{},
2   [] (const Employee &e) { return e.id });
```

We can now fallback to the standard less comparator for int.



Ranges 13



After projecting the others we get:

```
1 std::ranges::sort(employees, std::ranges::less{},
2   [] (const Employee &e) { return e.id; });
3
4 std::ranges::sort(payslips, std::ranges::less{},
5   [] (const Payslip &p) { return p.employee_id; });
6
7 std::ranges::equal(employees, payslips,
8   std::ranges::equal_to{},
9   [] (const Employee &e) { return e.id; },
10  [] (const Payslip &p) { return p.employee_id; });
```

Still a lot of lambda noise...



Ranges 14



But, the new standard allows pointers to members to be used as callables!

```
1 std::ranges::sort(employees, std::ranges::less{}, &Employee::id);
2
3 std::ranges::sort(payslips, std::ranges::less{}, &Payslip::employee_id);
4
5 std::ranges::equal(employees, payslips,
6   std::ranges::equal_to{}, &Employee::id, &Payslip::employee_id);
```



Ranges 15



Since the default comparator for `sort` is `less` and the default comparator for `equal` is `equal_to` so we can use the default constructor for them:

```
1 std::ranges::sort(employees, {}, &Employee::id);  
2  
3 std::ranges::sort(payslips, {}, &Payslip::employee_id);  
4  
5 std::ranges::equal(employees, payslips, {}, &Employee::id, &Payslip::employee_id);
```

3.4 | Lab exercise

3.4.1 | Ranges

Exercise 3: Ranges

You can find the exercises in the directory `ranges`.

Sort

In the directory `ranges/sort` you will find a program called `sort.cpp`. Your task is to use `std::ranges::sort` to sort the vector of names by their height using a projection. Unfortunately we cannot refer to the corresponding field name (since it is private). Find a way to use the getter instead and still use a projection. Test your code.

The resulting sorted vector is dumped using a loop. Get rid of the loop and use `std::ranges::copy` to copy it to an `ostream`. To get this working you have to overload the `operator<<` function. Test your code.

Bonus point if you manage to create a solution without public fields and/or a friend.

The solution.

4 | Concepts

4.1 | Templates

Templates

- Templates are a powerful mechanism
- Good to realize it is based on code expansion/duplication
- Back in the days we used the C preprocessor for this(!)
- Simple, but can also lead to problems

Example:

```
1 template<typename T>
2 auto max(const T a, const T b)
3 {
4     if(a>b)
5         return a;
6     else
7         return b;
8 }
```

- What would be an appropriate T?
- int, double, char *, Person?
- Somehow we should be able to *restrict* expansion!
- This is what concepts are all about



Concepts 2



4.2 | Template substitutions

Template substitutions

- Substitution failure is not an error (SFINAE). If a (particular) substitution of template parameters leads to an failure it does not neccessarily means an error.
- Term coined by David Vandevorde, Belgian computer scientist.
- This principle has been used as a programming technique.
- For example, it allows a template to determine certain properties of the template arguments at compile time.
- Following well known example taken from wikipedia: how to detect if a class has a type definition:



Concepts 3



```
1 template <typename T>
2 struct has_typedef_foobar {
3     typedef char yes[1];
4     typedef char no[2];
5
6     template <typename C> static yes& test(typename C::foobar*);
7     template <typename> static no& test(...);
8
9     static const bool value = sizeof(test<T>(nullptr)) == sizeof(yes);
10 };
11
12 struct foo { typedef float foobar; };
13 int main() {
14     static_assert(!has_typedef_foobar<int>::value);
15     static_assert(has_typedef_foobar<foo>::value);
16     return 0;
17 }
```

4.3 | SFINAE

SFINAE

SFINAE

- Allows us to insert a requirement
- When a specific substitution fails, just abandon the function
- Look for another function in the overload set
- Special header `typetraits` to make life easier

4.4 | Example

Example

Assume we want to make an add functions which should only work for floating point numbers (float and double).

```
1 template<typename T>
2 auto add(const T a, const T b)
3 {
4     return a+b;
5 }
```

The above definition does not stop us from calling add with integers. We can solve this using `decltype`¹¹.

- a set of classes to obtain type information compile time
- sometimes referred to as metaprogramming



Concepts 6



4.5 | Type Traits

Type Traits

Given a generic type T

- is it an integer?
- is it a function, pointer or a class?
- can you copy it?
- will it throw exceptions?

Type traits also allow you to transform types:

- add/remove const
- make it a reference or pointer
- turn it into signed/unsigned type

This is usefull in conditional compilation and strict programming.



Concepts 7



4.6 | What is a type trait

What is a type trait

A type trait¹¹ is a template struct which contains a member constant that holds the answer to the question or the transformation it performs.

This member is `value` or `type`:

```
1 template<typename T>
2 struct is_floating_point;
3
4 template<typename T>
5 struct remove_reference;
6
7 std::cout << std::is_floating_point<int>::value << std::endl;
8 std::remove_reference<int &>::type x;
```

This example will print 0 (since `int` is not a floating point type) and declares an `int` variable `x`.



Concepts 8



4.7 | Nicer Type Traits: Helper Types

Nicer Type Traits: Helper Types

A syntax extension¹² allows us to write `_v` and `_t` instead of the ugly `::value` and `::type` in our code:

```
1 std::is_signed<T>::value; /* ---> */ std::is_signed_v<T>;
2 std::remove_const<T>::type; /* ---> */ std::remove_const_t<T>;
```



Concepts 9



4.8 | `enable_if`

`enable_if`

`enable_if` is a meta function that:

- presents a convenient way to use SFINAE functionality
- is mainly used to restrict template functionality
- it 'enables' the template function

`enable_if<bool B, class T=void>` expands to

- `T` if `B` is true (1)
- no such member if `B` is false (0)



Concepts 10



4.9 | Template Type Restricted

Template Type Restricted

```
1 template <typename T,  
2         typename = std::enable_if_t<std::is_floating_point_v<T>>>  
3 auto add(const T a, const T b)  
4 {  
5     return a+b;  
6 }
```

- Restriction in the form of an extra (unnamed) typename parameter to the template function.
- For non-float types the evaluation of the default value of this typename parameter will fail and the function is ignored.
- Solution is a bit of a hack, `add<int, void>(1,2)` will still work (forcing the second typename parameter).



Concepts 11



4.10 | Type Traits Drawbacks

Type Traits Drawbacks

Using type traits is cumbersome:

- Error messages are long and hard to read
- Requirement specification is clumsy
- Not intended to be used like this when designed

Solution: use requirements and concepts¹².



Concepts 12



4.11 | The Use of Requires

The Use of Requires

Using the `requires`¹³ keyword we can more eloquently state our requirements:

```
1 template<typename T>
2 auto add(const T a, const T b) requires std::is_floating_point_v<T>
3 {
4     return a+b;
5 }
```

- No extra template parameter (no hack)
- Better error messages
- Specification works as documentation



Concepts 13



4.12 | Introducing concepts

Introducing concepts

We can even take it one step further by defining a concept¹⁴:

```
1 template<typename T>
2 concept floating_point = std::is_floating_point_v<T>;
3
4 template<floating_point T>
5 auto add(const T a, const T b)
6 {
7     return a+b;
8 }
```

Advantages:

- If requirements are not met the compiler generates a very clear error message
- We can easily define re-occurring concepts (type requirements)



Concepts 14



4.13 | Auto concepts (Terse Syntax)

Auto concepts (Terse Syntax)

Concepts used in combination with `auto` present an even nicer solution:

```
1 template<typename T>
2 concept floating_point = std::is_floating_point_v<T>;
3
4 auto add(const floating_point auto a, const floating_point auto b)
5 {
6     return a+b;
7 }
```

- Now the function header itself contains a complete specification of the function.
- Note that this last solution accepts a combination of `float` and `double` arguments while the template version did not.



Concepts 15



4.14 | Terse Syntax

Terse Syntax

```
1 template<typename T>
2 concept floating_point = std::is_floating_point_v<T>;
3
4 floating_point auto add(const floating_point auto a, const floating_point auto b)
5 {
6     return a+b;
7 }
8
9 floating_point auto c=add(1.2,2.3);
```

Terse syntax is allowed not only on constrained parameters, but also for the result type (line 4) and constrained declarations (line 9).



Concepts 16



4.15 | Concept Definition

Concept Definition

```
1 template<typename T, typename U>
2 concept MyConcept = std::same_as<T,U> &&
3                     (std::is_class_v<T> ||
4                     std::is_enum_v<T>);
```

A concept definition consists of:

- a template header
- the concept name
- constraints, a logical formula of applications of other concepts or requirements



Concepts 17



4.16 | Concept Definition

Concept Definition

A constraint may be a requires clause which forces an expression to compile:

```
1 template<typename T>
2 concept hashable = requires(T t)
3 {
4     t.hash();
5 }
```

Or an expression to compile and produce a result that is convertible to int:

```
1 template<typename T>
2 concept hashable = requires(T t)
3 {
4     { t.hash() } -> std::convertible_to<int>;
5 }
```



Concepts 18



4.17 | Lab exercise

4.17.1 | Concepts

Exercise 4: Concepts

All exercises about the creation of concepts can be found under `concepts`.

Animal farm

The sources for this exercise can be found in `concepts/animals`.

It is a bit of a zoo out there, with all sorts of animals running around. For every animal we get we have to write special petting instructions. What if we could have `cat_like` instead of `cat`, so that when we finally get a lion we can use the same instructions?

Translating this to C++: why don't we use *concepts*?

We have created several animal classes in `animals.h`:

```
#pragma once
#include <iostream>
#include <string>

class animal {
public:
    void wag_tail(){}
};

class elephant {
public:
    void wag_tail(){}
};

class door_knob {
public:
    void give_hand(){}
};

class robin {
public:
    void flap_wings(){}
    void wag_tail(){}
};

class bird {
public:
    void flap_wings(){}
    void wag_tail(){}
};

class cat {
public:
    void meow(){}
    void wag_tail(){}
};

class dog {
public:
    void bark(){}
    void wag_tail(){}
};

class Aethalops {
public:
    void bark(){}
    void flap_wings(){}
    void wag_tail(){}
};
```

We also provided *petting instructions* in `pet_funcs.h`:

```
#pragma once
#include "animals.h"
#include "tools.h"

void pet (animal a){
    std::cout << "careful!\n";
}

void pet (cat a){
    std::cout << "stroke gently\n";
}

void pet (dog a){
    std::cout << "ruffle fur\n";
}

void pet (bird a){
    std::cout << "don't ruffle the feathers\n";
}

void pet (auto a){
    WHAT<decltype(a)>("template version of pet()");
    //std::basic_ofstream<char8_t> stream("/dev/tty");
    // slightly violent:
    std::cout << reinterpret_cast<const char*>(u8"\U0001f600\n");
}
```

The program (animals.cpp) is simple:

```
#include "animals.h"
#include "pet_concepts.h"
#include "pet_funcs.h"

int main(){
    elephant dombo;
    cat felix;
    dog tarzan;
    robin tweet;
    Aethalops nr412;
    door_knob front_door;

    pet(dombo);
    pet(felix);
    pet(tarzan);
    pet(tweet);
    pet(nr412);
    pet(front_door);
}
```

You're expected to put the concepts you write in pet_concepts.h (it's empty but gets included in the program).

Animals

When you run the program (`make; animals`) you'll notice that most animals will go to the templated function, instead of to the overload with `animal`.

Your first step will be to write a concept called `animal_like`, that needs to select on something that animals do (`wag_tail` perhaps). When the concept is available, modify the `animal` overload to expect `animal_like` `auto` (or template form).

Compile and run. You should see a lot more calls to your `animal_like` overload. Please note that the door knob is not an animal and should always end up with the template version.

Cats

Now do the same with `cat_like` (concept and function).

If you get a message like this:

```
animals.cpp:14:8: error: call of overloaded 'pet(cat&)' is ambiguous
   14 |     pet(felix);
      |     ~~~^~~~~~
In file included from animals.cpp:3:
pet_funcs.h:5:6: note: candidate: 'void pet(auto:11) [with auto:11 = cat]'
    5 | void pet(animal_like auto a){
      |     ^~~
pet_funcs.h:10:6: note: candidate: 'void pet(auto:12) [with auto:12 = cat]'
   10 | void pet(cat_like auto a){
      |     ^~~
pet_funcs.h:26:6: note: candidate: 'void pet(auto:16) [with auto:16 = cat]'
   26 | void pet(auto a){
      |     ^~~
```

you have probably forgotten that a cat is also an animal. In the error message it is *obvious* that the compiler considers both `cat_like` and `animal_like`. Make sure that `cat_like` is a refinement of `animal_like`.

When it compiles continue with the other named animals (dog and bird). Eventually Aethalops will come to haunt you.

Aethalops

You will see an error message similar to this:

```
animals.cpp:17:8: error: call of overloaded 'pet(Aethalops&)' is ambiguous
   17 |     pet(nr412);
      |     ~~~^~~~~~
In file included from animals.cpp:3:
pet_funcs.h:5:6: note: candidate: 'void pet(auto:11) [with auto:11 = Aethalops]'
    5 | void pet(animal_like auto a){
      |     ^~~
pet_funcs.h:13:6: note: candidate: 'void pet(auto:13) [with auto:13 = Aethalops]'
   13 | void pet(dog_like auto a){
      |     ^~~
pet_funcs.h:17:6: note: candidate: 'void pet(auto:14) [with auto:14 = Aethalops]'
   17 | void pet(bird_like auto a){
      |     ^~~
pet_funcs.h:21:6: note: candidate: 'void pet(auto:15) [with auto:15 = Aethalops]'
   21 | void pet(auto a){
      |     ^~~
```

Fix the Aethalops problem by adding a `flying_dog_like` concept and corresponding function.

The solution.

5 | Views

5.1 | Views

Views

- The standard algorithms are great
- Drawback: they do not compose well
- This is because of their inherent *eager* evaluation style

Let's look at an example

```
1 void print_squares(const vector<int> &vec)
2 {
3     for(int i: vec)
4         cout << i*i;
5 }
```



Views 2



Of course we do not want to use raw loops (Sean Parent, C++ Seasoning)
Possible solution:

```
1 void print_squares(const vector<int> &vec)
2 {
3     ranges::transform(vec, ostream_iterator<int>{cout},
4                       [](int i) { return i*i; } );
5 }
```



Views 3



Suppose the spec change, we want to print the even squares:

```
1 void print_even_squares(const vector<int> &vec)
2 {
3     for(int i: vec)
4         if(i%2 == 0)
5             cout << i*i;
6 }
```

How do we perform this with the STL?



Views 4




We could do it like this:

```
1 void print_even_squares(const vector<int> vec)
2 {
3     auto removed = ranges::remove_if(vec, [] (int i) { return i%2!=0 });
4     ranges::transform(vec.begin(), removed.begin(), ostream_iterator<int>{cout},
5                       [] (int i) { return i*i; } );
6 }
```

Not really nice...

- we changed the vector
- reason: STL is eager: we finish one step before we go to the next

The new range adapters, views  offer lazy evaluation instead.
(something that functional programmers do for years)



Views 5



Use two views now, filter and transform:

```
1 void print_even_squares(const std::vector<int>& vec)
2 {
3     auto square = [] (auto i) { return i*i; };
4     auto is_even = [] (auto i) { return i%2==0; };
5
6     auto view = ranges::views::transform(
7         ranges::views::filter(vec, is_even), square);
8
9     ranges::copy(view, ostream_iterator<int>{cout});
10 }
```

Setting up the view is not doing any work! The real work start when copying. The notation is a bit weird, you have to read it inside out.



Views 6



That is why ranges overloads the | operator:

```
1 void print_even_squares(const std::vector<int>& vec)
2 {
3     auto square = [] (auto i) { return i*i; };
4     auto is_even = [] (auto i) { return i%2==0; };
5
6     auto view = vec
7         | ranges::view::filter(is_even)
8         | ranges::view::transform(square);
9
10     ranges::copy(view, ostream_iterator<int>{cout});
11 }
```

This is exactly the same as previous slide, but now more readable. The | resembles the unix pipe!



Views 7



5.2 | Lab exercise

5.2.1 | Views

Exercise 5: Views

The exercise(s) for this subject can be found in `views`.

Large primes

You find a starting point in `views/primes`.

Create a program that prints ten prime numbers above 10,000. The print twenty of them.

You can use the following function (`prime.h`) to test for primality (or roll your own):

```
#pragma once
#include <concepts>

struct is_prime{
    template<std::integral N>
    bool operator() (N x) {
        for (N d{2}; d*d<=x; ++d) {
            if (x%d == 0) return false;
        }
        return true;
    }
};
```

A framework can be found in `primes.cpp`:

```
#include <iostream>
#include <ranges>

#include "prime.h"

int main() {
}
```

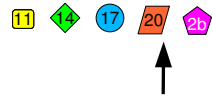
The solution.

7.1 | Three way comparison

Three way comparison

Topics:

- Spaceship operator
- Automatic generation
- DIY
- Types of ordering
- Types of equality



7.2 | Example

Example

Remember the **Person** class in the ranges presentation?

```
01 struct Person {
02     int bsn;
03     std::string name;
04     std::string surname;
05     auto operator <=> (const Person&) const = default;
06 };

```

default will also default == if you don't

default will figure this out based on std::string and int

default will generate something like this (as a method):

```
01 auto operator <=> (const Person& p2) const {
02     if(auto cmp = (bsn <=> p2.bsn); cmp != 0)
03         return cmp;
04     if(auto cmp = (name <=> p2.name); cmp != 0)
05         return cmp;
06     if(auto cmp = (surname <=> p2.surname); cmp != 0)
07         return cmp;
08     return 0;
09 };

```

Better get the right ordering here

No <=> (default or otherwise): no comparisons.

7.3 | Three way compare (operator <=>) <compare>

Three way compare (operator <=>) <compare>

You provide operator <=> (and ==) and the compiler generates all comparison operations...

But, not all types are totally (strongly) ordered.

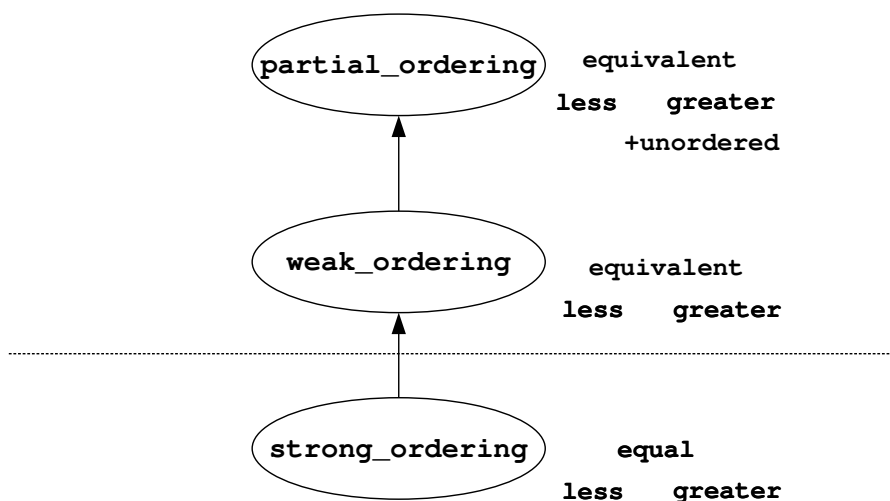
The return types of <=> determine:

- the ordering
 - `std::partial_ordering`: (a<b, b<c, a<c may not be true) (concepts)
 - `std::weak_ordering`: (a<b, b<c, a<c)
 - `std::strong_ordering`: as weak ordering, but with strong equality.
- the return type determines the comparison operations you get:
 - `std::partial_ordering`: <, >
 - `std::weak_ordering`: <,>, <=, >=
 - `std::strong_ordering`: as weak ordering.

Defining (or defaulting) == will also define !=.

7.4 | Return relations

Return relations



The Math is simple: -1 0 $+1$ mostly no-op

7.5 | Lab exercise

7.5.1 | Three way comparison

Exercise 7: Three way comparison

The exercise(s) for this topic can be found in `threeway`.

Generating the spaceship operator

A framework directory can be found in `threeway/spaceship`.

It contains a few samples of what we expect you to do.

Create one or more structures with data members than can be compared. Then try to figure out what the return type of the spaceship operator (`<=>`) is. You can use `SHOW<your type>(msg)` from `helper.h`:

```
#include "tools.h"

template <typename T>
concept has_spaceship =
    requires(T a, T b) {
        a <=> b;
    };

template <has_spaceship T>
void SHOW(std::string s="", fudge::source_location loc=fudge::source_
    location::current()) {
    SHOW_ORDERING<decltype(std::declval<T>() <=> std::declval<T>())>(s, loc);
}
```

Do you need to tell the compiler to generate the spaceship operator?

Do the same using a tuples, for example:

```
struct X {
    int x;
    string s;
};
```

```
tuple<int, string>
```

The solution.

8 | Modules

8.1 | Forward declarations

Forward declarations

C++ has a particular property:

- Stuff should be declared before used (inherited from C)
- Leads to reverse order programs

```
1 int square(int x) {  
2     return x*x;  
3 }  
4  
5 int func(int a, int b) {  
6     return a+square(b);  
7 }  
8  
9 int main(int argc, char **argv) {  
10     std::cout << func(1,2)<< std::endl;  
11 }
```



- It is not always possible to find a right order
- What if function a uses function b and b uses a
- Same holds for class declarations
- Solution: use prototyping (forward declaration)
- The prototype is *not* a definition, just an announcement of what is to come



Modules 2



8.2 | Prototypes

Prototypes

Previous example with (function) prototypes:

```
1 int square(int x);
2 int func(int a, int b);
3
4 int main(int argc, char **argv) {
5     std::cout << func(1,2)<< std::endl;
6 }
7
8 int func(int a, int b) {
9     return a+square(b);
10 }
11
12 int square(int x) {
13     return x*x;
14 }
```



Modules 3



For classes we just mention the name:

```
1 class Node;  
2 class Tree  
3 {  
4     private:  
5         Node *node;  
6     ...  
7 };
```

Note that the use of the class `Node` here is limited:

- It is called an incomplete type
- Cannot use stuff that relates to its size or members

8.3 | Include files

Include files

Traditionally forward declarations are stored in an include file

- include file ends with `.h` or `.hpp`

Include files are also used to store:

- class definitions (not the implementation)
- templates (cannot be precompiled)

This way we can 'include' the information into more than one compilation unit.

8.4 | Chaos

Chaos

Since functions may use other functions and classes use other classes:

- include files may include other include files
- the order might be relevant (unsolved)
- include files may be included twice (solved)

But class definitions should not occur more than once

- Solution: use preprocessor to prevent multiple inclusion



Modules 6



8.5 | Preventing multiple inclusion

Preventing multiple inclusion

File: Person.h

```
1 #ifndef _TREE_H_
2 #define _TREE_H_
3
4 #include "Address.h"
5
6 class Person
7 {
8     private:
9         Address *address;
10     ...
11 };
12 #endif
```

The preprocessor variable guards multiple inclusion

- Variable naming scheme is up to you
- Might clash with others (these bugs are hard to find!)



Modules 7



More modern solution:

```
1 #pragma once
2
3 #include "Address.h"
4
5 class Person
6 {
7     private:
8         Address *address;
9     ...
10 };
```

8.6 | Modules

Modules

Recent compilers support modules⁹ to circumvent (some of) the problems with includes files:

- gcc 11 or higher
- clang 8 or higher
- MSVC 19.28 or higher

As we will see the introduction of modules has an impact on the build system.

Multiple files are involved with the module system:

- The *Primary Module Interface* (PMI). This file:
 - Introduces the module name (using the `export module` keywords)
 - Exports the functions and classes of the module
 - Has extension `.cpp` (gcc/clang) or `.ixx` (msvc)
 - This file will be compiled into an internal format
 - Note: templates should always be defined in this file
- The *implementation file(s)*. These file(s)
 - Refer to the module name (using the `module` keyword)
 - Define the classes and functions announced in the PMI
- An *application file*. This file
 - use the library (using the `import` keyword)



Modules 10



Example Primary Module Interface, file `stuff.cpp`:

```
1 export module stuff;  
2  
3 export int foo() { return 7};  
4 export some();
```

Note that for one function the implementation is provided. The other function is defined in the:

Implementation module, file `stuff_impl.cpp`:

```
1 module stuff;  
2  
3 int some() { return 42; }
```



Modules 11



The module can be used in file `main.cpp`:

```
1 import stuff;
2
3 #include <iostream>
4
5 int main(int argc, char **argv)
6 {
7     std::cout << foo() << std::endl;
8     std::cout << some() << std::endl;
9 }
```

The order of compilation is important: the PMI should be compiled *before* the implementation and/or application modules.



8.7 | Building Order

Building Order

- In more complex situations modules can use other modules
- This makes the building order non-trivial
- For closed programming environments (like Visual Studio) this is not a problem
- For external portable building tools (like CMake) this *is* a problem
- The CMake developers contacted Microsoft, gcc and clang to implement compiler options that generate the build hierarchy
- Bleeding edge versions of CMake (3.28) and gcc/clang/MSVC correctly implement and build modules



8.8 | Namespaces

Namespaces

Modules are *not* namespaces, but you can use them inside a PMI and implementation module:

```
1 export module stuff;  
2  
3 namespace Stuff {  
4     export int bar();  
5 }  
6  
7 export namespace Stuff {  
8     int foo();  
9 }  
10  
11 namespace Stuff {  
12     int foo() { return 4712; }  
13     int bar() { return 12; }  
14 }
```



Modules 14



Note that on the previous slide:

- The export and definition of the functions are separated (and not in an implementation file)
- Exporting a namespace just exports everything defined in that namespace block

Application of this module:

```
1 import stuff;  
2  
3 #include <iostream>  
4  
5 int main(int argc, char **argv)  
6 {  
7     std::cout << Stuff::foo() << std::endl;  
8     std::cout << Stuff::bar() << std::endl;  
9 }
```



Modules 15



What if we want to include (legacy) stuff in our PMI or Implementation Module?

- This is only allowed in the *Global Module Fragment* (GMF), a dedicated code section which is created at the beginning of the file starting with `module;`

```
1 module;  
2 #include <string>  
3 export module stuff;  
4 ...
```

As an alternative we can *import* the header file (and don't have to create the GMF):

```
1 export module stuff;  
2 import <string>;  
3 ...
```



Modules 16



To import an header file we have to convert it to a *header-unit*. Availability of these header units is compiler dependant. For gcc we can create them manually:

```
1 g++ -fmodules-ts -x c++-system-header string
```

Although the second (header import method) is favourable, I would advise against it with the current gcc compiler (version 12) due to the buggy (unfinished) implementation.



Modules 17



8.9 | Lab exercise

8.9.1 | Modules

Exercise 8: Modules

Create a simple die rolling library

Sources can be found in `modules/roll`.

Check out the first part of the file `main.cpp` (upto the `#if`). Add code to the Primary Module Interface `roll.cpp` so that it:

- defines the module name (`roll`)
- sets up a namespace (`roll`)
- exports a class named `Die` that implements the rolling of a simple die. The functionality intended can be derived from the application code in `main.cpp`.

Do not write all code in the PMI, use an implementation module for the `Die` class. It is also nice to write the class definition in a separate `.h` file which can be included from the PMI. You can use the c function `rand()` to create a random number, just include `<cstdlib>` (in the right place!).

Extend your roll module with the second part of the `main.cpp` file. Now we have to implement the `DieSet` class that implements a set of dice. Again update your PMI, and add definition and implementation of the new class. Hint: use a vector to store the dice of the set.

Playing around with the module from the presentation

Sources can be found in `modules/stuff`.

There is one module in the directory.

Play around with it a bit, perhaps messing around with the compilation order (remember the build system is very important with modules).

Templates in modules

Sources can be found in `modules/frutssel`

Try calling `Frutssel::bar` with a new type of parameter. Does that work?

Try moving the template body to `frutssel_impl.cpp`.

The solution.

9 | Co-routines

9.1 | What is a coroutine?

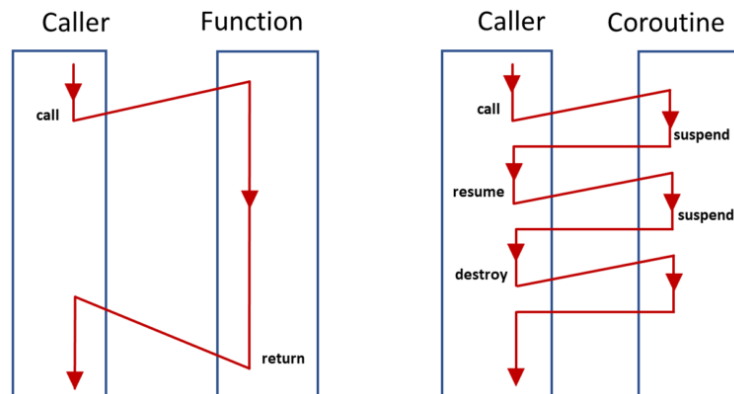
What is a coroutine?

A co-routine is a function

- That can be called
- That runs 'concurrently' with the caller
- Can yield multiple results
- Can be resumed

Co-routines can be seen as:

- poor man's parallelism
- time slicing which depends on trust
- parallelism without the headaches of multi-threading



source: modernescpp.com



Coroutines 2



Do we really need coroutines?

- There are other ways to store the context
- Why just call a method again
- What about the calling history (stack contents)

Typical applications

- generators (python)
- lazy ranges
- traversal of recursive structures
- pseudo parallelism



Coroutines 3



9.2 | Example use of coroutines

Example use of coroutines

Imagine developing a game

- High framerate, limited time to perform updates
- Main gameloop has to perform multiple tasks:
 - calculate new position of npc's
 - scan keyboard for user commands
 - ...
- each task has it's own 'environment', variables etc
- ideal situation to run subtasks in coroutines



Coroutines 4



9.3 | First attempt

First attempt

According to the standard, a coroutine^{AT} is a function that uses

- `co_await` (suspend until resumed) or
- `co_yield` (suspend and return a value) or
- `co_return` (exit while returning a value)

First (not yet correct) attempt:

```
1 int co_f() {
2     while(true)
3         co_yield i++;
4 }
5
6 auto r=co_f();
7 for(auto v: r) {
8     std::cout << v << std::endl;
9 }
```



Coroutines 5



9.4 | The coroutine return type

The coroutine return type

The coroutine return type is special and functions as a contract between the caller and the coroutine, it contains

- the coroutine state, which stores the stack/local variables
- a handle, manipulated from outside the coroutine
- the promise object, manipulated from inside the coroutine which implements some predefined functions and usually contains the actual return value

The return type controls the coroutine and the communication. In the coming slides we will study several implementations.



Coroutines 6



9.5 | Coroutine without suspensions

Coroutine without suspensions

We start out simple, suppose we have the following situation:

```
1 nothing co_f()
2 {
3     co_return;
4 }
5
6 int main()
7 {
8     auto r=co_f();
9 }
```



Coroutines 7



A return type that allows the coroutine to run to completion returning nothing.

```
1 struct nothing {
2     struct promise_type {
3         promise_type() { }
4         nothing get_return_object() {
5             return {};
6         }
7         void return_void() { }
8         void unhandled_exception() { }
9         std::suspend_never initial_suspend() {
10             return {};
11         }
12         std::suspend_never final_suspend() noexcept {
13             return {};
14         }
15     };
16 };
```



Coroutines 8



Assume we have a coroutine that returns a value:

```
1 result<int> co_f()
2 {
3     co_return 77;
4 }
5
6 int main()
7 {
8     auto r=co_f();
9     std::cout << r.h.promise().value << std::endl;
10 }
```



Coroutines 9



A return type that allows the coroutine to run to completion returning something. The actual return type is a template parameter.

```
1 template<typename T>
2 struct result {
3     struct promise_type {
4         result get_return_object() {
5             return { .h = std::coroutine_handle<promise_type>::from_promise(*this), };
6         }
7         //void return_void() { }
8         void return_value(T t) {
9             value=t;
10        }
11        T value;
12    };
13    std::coroutine_handle<promise_type> h;
14 };
```



Coroutines 10



Now we have a coroutine that returns a series of values:

```
1 multiple<int> co_f()
2 {
3     int i=0;
4     while(true)
5         co_yield i++;
6 }
7
8 int main()
9 {
10    while(true)
11    {
12        auto r=co_f();
13        std::cout << r.h.promise().value << std::endl;
14        if(i>12)
15            break;
16    }
17 }
```



Coroutines 11



A return type that allows the coroutine to return multiple results.

```
1 template<typename T>
2 struct multiple {
3     struct promise_type {
4         result get_return_object() {
5             return { .h = std::coroutine_handle<promise_type>::from_promise(*this), };
6         }
7         std::suspend_always yield_value(T t) {
8             value=t;
9             return {};
10        }
11        T value;
12    };
13    std::coroutine_handle<promise_type> h;
14 };
```



Coroutines 12



9.6 | support

support

Do we have to create our own result type all the time?

- You can, tailored to your needs
- There are no STL provided classes (yet)
- Alternatives (support not guaranteed):
 - CppCoro <https://github.com/lewissbaker/cppcoro> (msvc/clang)
 - libcoro <https://github.com/jbaldwin/libcoro.git> (msvc/clang/gcc)
- May be introduced in future standards



Coroutines 13



```
1 #include <iostream>
2 #include "libcoro/include/coro/generator.hpp"
3
4 coro::generator<const int> fibonacci()
5 {
6     int a = 0, b = 1;
7     while (true)
8     {
9         co_yield b;
10        auto tmp = a; a = b; b += tmp;
11    }
12 }
13
14 int main(int argc, char **argv)
15 {
16     for (auto i : fibonacci())
17     {
18         if (i > 1'000'000) break;
19         std::cout << i << std::endl;
20     }
21 }
```



Coroutines 14

9.7 | Lab exercise

9.7.1 | Coroutines

Exercise 9: Coroutines

Sieve of Eratosthenes

Build your code in `coroutines/sieve`.

For this exercise we are going to use the generator template provided by Baldwin's libcoro. First get the library using the following command:

```
git clone https://github.com/jbaldwin/libcoro.git
```

Upon completion you should have a copy of the library in the libcoro subdirectory.

Finish the coroutine `sieve` in `sieve.cpp` so that the main function prints out all prime numbers below 100. A brief introduction to the algorithm can be found on wikipedia.

Inorder Recursive Traversal of a Binary Tree

Sources can be found in `coroutines/tree`.

In this exercise we are going to write a coroutine member function that recursively yields all values (in order) of a binary tree. Study the application code in `main.cpp` and implement the `inorder` method. If you want to make use of the libcoro library you will have to run the git command in this directory as well.

Bonus: notice that the program will not work for empty trees. Cleanup your code so that it uses `std::optional` instead of `nullptr`.

Tracing the path

Sources can be found in `coroutines/headers`.

Compile the programs and try to understand the different steps involved (all the code has `TRACE`-macros).

The solution.

🎓 Exercise 3: (Solution) Ranges

Sort

sort.cpp

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <vector>

class Person
{
private:
    const char *first_name;
    const char *last_name;
    int length;
public:
    Person(const char *first_name, const char *last_name, int length): first_
↪name(first_name), last_name(last_name), length(length) { ; }
    int get_length() { return length; }
    void operator<<(std::ostream &os) const
    {
        os << first_name << " " << last_name << " " << length << std::endl;
    }
};

std::ostream& operator<<(std::ostream &os, const Person *p)
{
    *p << os;
    return os;
}

int main(int argc, char **argv)
{
    std::vector<Person *> persons;
    persons.push_back(new Person("Franc", "Grootjen", 185));
    persons.push_back(new Person(" ", " ", 178));
    persons.push_back(new Person(" ", " ", 190));
    persons.push_back(new Person(" ", " ", 172));
    persons.push_back(new Person(" ", " ", 175));
    persons.push_back(new Person(" ", " ", 167));
    persons.push_back(new Person(" ", " ", 184));
    persons.push_back(new Person(" ", " ", 193));
    persons.push_back(new Person(" ", " ", 172));
    persons.push_back(new Person(" ", " ", 178));
    persons.push_back(new Person(" ", " ", 192));
    persons.push_back(new Person(" ", " ", 176));

    std::ranges::sort(persons, {}, &Person::get_length);

    std::ranges::copy(persons, std::ostream_iterator<Person *>(std::cout));
}
```

Return to the exercise text.

Exercise 4: (Solution) Concepts

Animal farm

Animals

The concept `animal_like` should look something like this:

```
#pragma once

template <typename T>
concept animal_like =
    requires(T t){
        t.wag_tail();
    };
```

Cats

Naively you might write `cat_like` like this:

```
template <typename T>
concept cat_like =
    requires(T t){
        t.meow();
    };
```

this will cause the ambiguous overload. `cat_like` should be `animal_like` as well (and similar for dog and bird):

```
template <typename T>
concept cat_like = animal_like<T> &&
    requires(T t){
        t.meow();
    };

template <typename T>
concept bird_like = animal_like<T> &&
    requires(T t){
        t.flap_wings();
    };

template <typename T>
concept dog_like = animal_like<T> &&
    requires(T t){
        t.bark();
    };
```

Aethalops

The concept will look like this:

```
#pragma once

template <typename T>
concept flying_dog_like = dog_like<T> && bird_like<T>;
```

and the corresponding function like this:

```
#pragma once
#include "animals.h"

void pet(flying_dog_like auto a){
    std::cout << "ruffle the fur, but don't ruffle the feathers\n";
}
```

[Return to the exercise text.](#)

Exercise 5: (Solution) Views

Large primes

I seemed to remember that numbers of the form $2N^2+1$ are likely prime, so I used that. The first N yielding a number above 10,000 is 50, so I went for generating all numbers from 1, dropping everything up to 50, applying the formula and using filter to test for primality. Finally just take the last 10 (or 20).

```
#include <iostream>
#include <ranges>
#include "prime.h"

int main() {
    auto high_primes = std::views::iota(1)
        | std::views::drop_while([](const auto& x) {return x<50;})
        | std::views::transform([](const auto& x) {return 2*x*x+1;})
        | std::views::filter(is_prime{});

    for(auto prime: high_primes | std::views::take(10)) {
        std::cout << prime << ' ';
    }
    std::cout << '\n';

    for(auto prime: high_primes | std::views::take(5)) {
        std::cout << prime << ' ';
    }
    std::cout << '\n';
}
```

If you add this line (and the include for `tools.h`):

```
WHAT<decltype(high_primes)>("high_primes");
```

You will get something like this (mercilessly reformatted):

```
std::ranges::filter_view<
    std::ranges::transform_view<
        std::ranges::drop_while_view<
            std::ranges::iota_view<int, std::unreachable_sentinel_t>,
            main()::<lambda(const auto:16&)>
        >,
        main()::<lambda(const auto:17&)>
    >,
    is_prime>
```

As you can see the pipeline will construct the same object you would get from using the `...._view(range,...)` variant and functional composition. In fact `view | std::views::adaptor(...)` (indirectly) calls `std::ranges::adaptor_view(view, ...)`.

Return to the exercise text.

Exercise 7: (Solution) Three way comparison

Generating the spaceship operator

The results should be rather obvious. If you don't tell the compiler to generate `<=>` you won't get it for `struct s`, but for `tuple s` it is generated by default (which is why we could use `tuple s` to such great effect in projections (`std::ranges::sort`)).

[Return to the exercise text.](#)

Exercise 8: (Solution) Modules

Create a simple die rolling library

role.cpp

```
module;
#include <vector>
export module roll;
export namespace roll
{
    #include "Die.h"
    #include "DieSet.h"
}
```

Die.cpp

```
module;
#include <cstdlib>
module roll;

namespace roll
{
    Die::Die(int max) : max(max)
    {
        ;
    }
    int Die::roll()
    {
        return 1+rand()%max;
    }
}
```

DieSet.cpp

```
module;
#include <vector>
module roll;

namespace roll
{
    int DieSet::roll()
    {
        int total=0;
        for(Die die:dice)
            total+=die.roll();
        return total;
    }
    void DieSet::add(Die die)
    {
        dice.push_back(die);
    }
}
```


Playing around with the module from the presentation

Ask around for help, sharing insights etc.

Templates in modules

Calling `Frutssel::bar` with `3.14159` will work. Just as with header files this does not require of the implementation unit.

Moving the implementation to `frutssel_impl.cpp` will not work.

Return to the exercise text.

Exercise 9: (Solution) Coroutines

Sieve of Eratosthenes

Inorder Recursive Traversal of a Binary Tree

Tracing the path

tree.cpp

```
#include "libcoro/include/coro/generator.hpp"
#include <iostream>

template<typename T>
class BinTree
{
private:
    BinTree *left;
    T value;
    BinTree *right;
public:
    BinTree(BinTree *left, T value, BinTree *right): left(left),
        value(value), right(right)
    {
        ;
    }
    coro::generator<T> traverse()
    {
        if(left!=nullptr)
        {
            for(auto v:left->traverse())
                co_yield v;
        };
        co_yield value;
        if(right!=nullptr)
        {
            for(auto v:right->traverse())
                co_yield v;
        }
    }
};

int main() {
    BinTree<int> *tree=new BinTree<int>(new BinTree<int>(nullptr,4,nullptr),10,new
    BinTree<int>(nullptr,12,nullptr));
    for(auto v : tree->traverse())
        std::cout << v << std::endl;
}
```

sieve.cpp

```
#include <iostream>
#include "libcoro/include/coro/generator.hpp"

coro::generator<int> sieve()
{
    bool candidate[100];
    for(int i=0; i<100; i++)
        candidate[i]=true;
    int i=2;
    while(i<100)
    {
        // yield the prime
        co_yield i;
        // wipe out all multiples
        for(int j=i+i; j<100; j+=i)
            candidate[j]=false;
        i++;
        // skip already wiped out numbers
        while(!candidate[i] && i<100)
            i++;
    }
}

int main(int argc, char **argv)
{
    for (auto i : sieve())
        std::cout << i << std::endl;
}
```

It's only a concept and that is not what this course is about :-) Used the traces myself to understand it all...

[Return to the exercise text.](#)

```
#pragma once
#include <chrono>
#include <ranges>

void last_day(){
    using namespace std::chrono;
    auto m = last/January/2020;
    for(auto x: std::ranges::iota_view(1, 12)){
        std::cout << to_string(m) << '\n';
        auto d = year_month_day{m};
        std::cout << unsigned(d.day()) << '\n';
        m = { m.year(), month_day_last{m.month() + months(1)} };
    }
}
```

This one exploits views a little bit:

```
#pragma once
#include <chrono>
#include <ranges>

void last_day_alt2(){
    using namespace std::chrono;
    auto my_months = std::views::iota(1, 12)
        | std::views::transform([](int m){return month{uint(m)};});
    for(auto current_month: my_months){
        auto date = last/current_month/2020;
        std::cout << to_string(date) << '\n';
        auto d = year_month_day{date};
        std::cout << unsigned(d.day()) << '\n';
    }
}
```

Week days

```
#pragma once
#include <chrono>
#include <ranges>

void day23(){
    using namespace std::chrono;

    auto m = 23d/January/2020;
    for(auto x: std::ranges::iota_view{1,12}|std::views::reverse){
        std::cout << to_string(m) << '\n';
        auto d = year_month_weekday{m};
        std::cout << to_day_name(d.weekday().c_encoding()) << '\n';
        m = { m.year(), m.month() + months(1), m.day() };
    }
}
```

Last Saturday

```
#pragma once
#include <chrono>
#include <ranges>

void last_sat() {
    using namespace std::chrono;
    auto m = Saturday[last]/January/2020;
    for(auto x: std::ranges::iota_view(1,12)){
        std::cout << to_string(m) << '\n';
        auto d = year_month_weekday{m};
        std::cout << rank(d.index()) << '\n';
        m = { m.year(), m.month() + months(1), m.weekday_last() };
    }
}
```

[Return to the exercise text.](#)