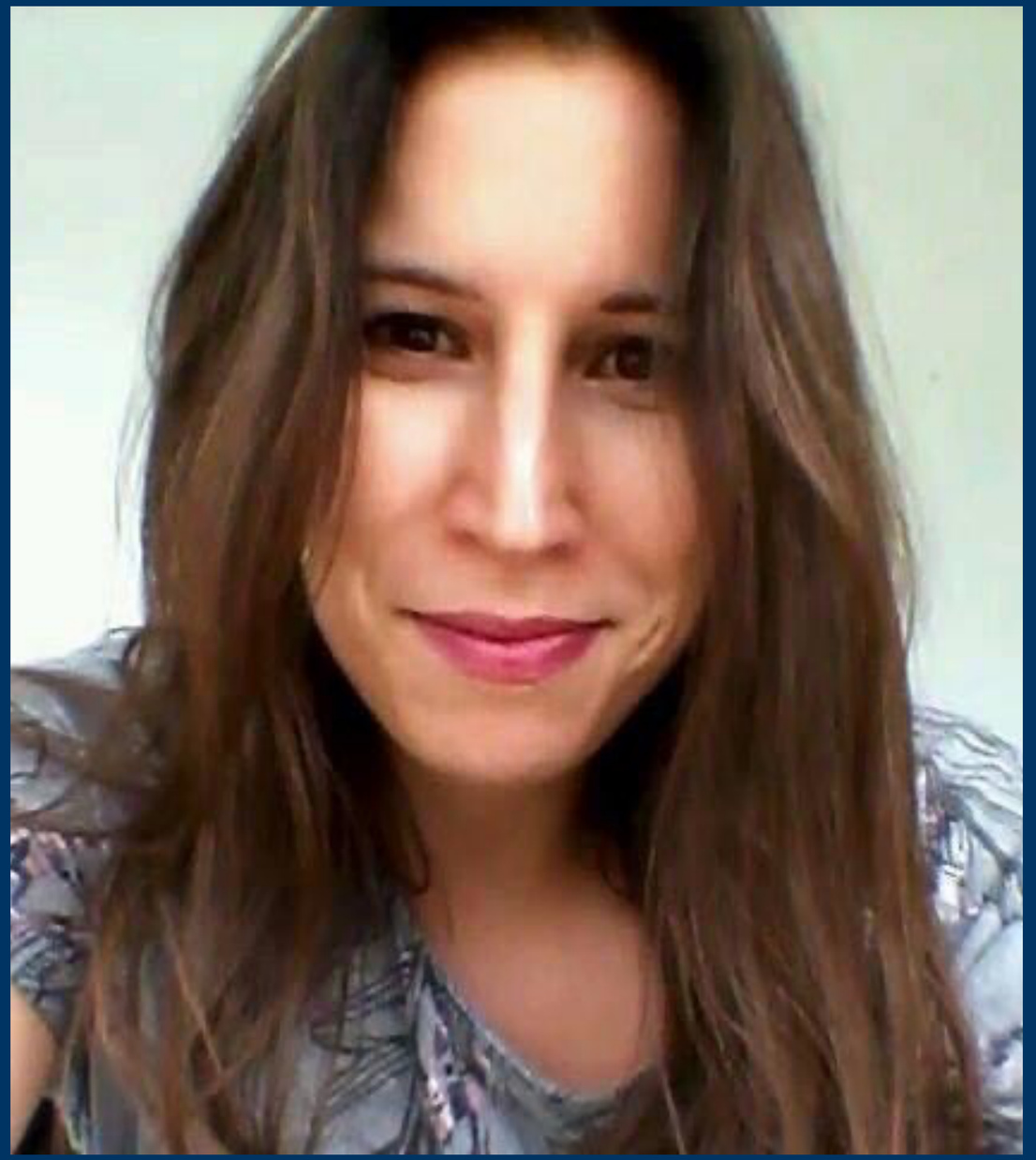
Embracing CTAD

NINA RANNS

2023
October 01 - 06

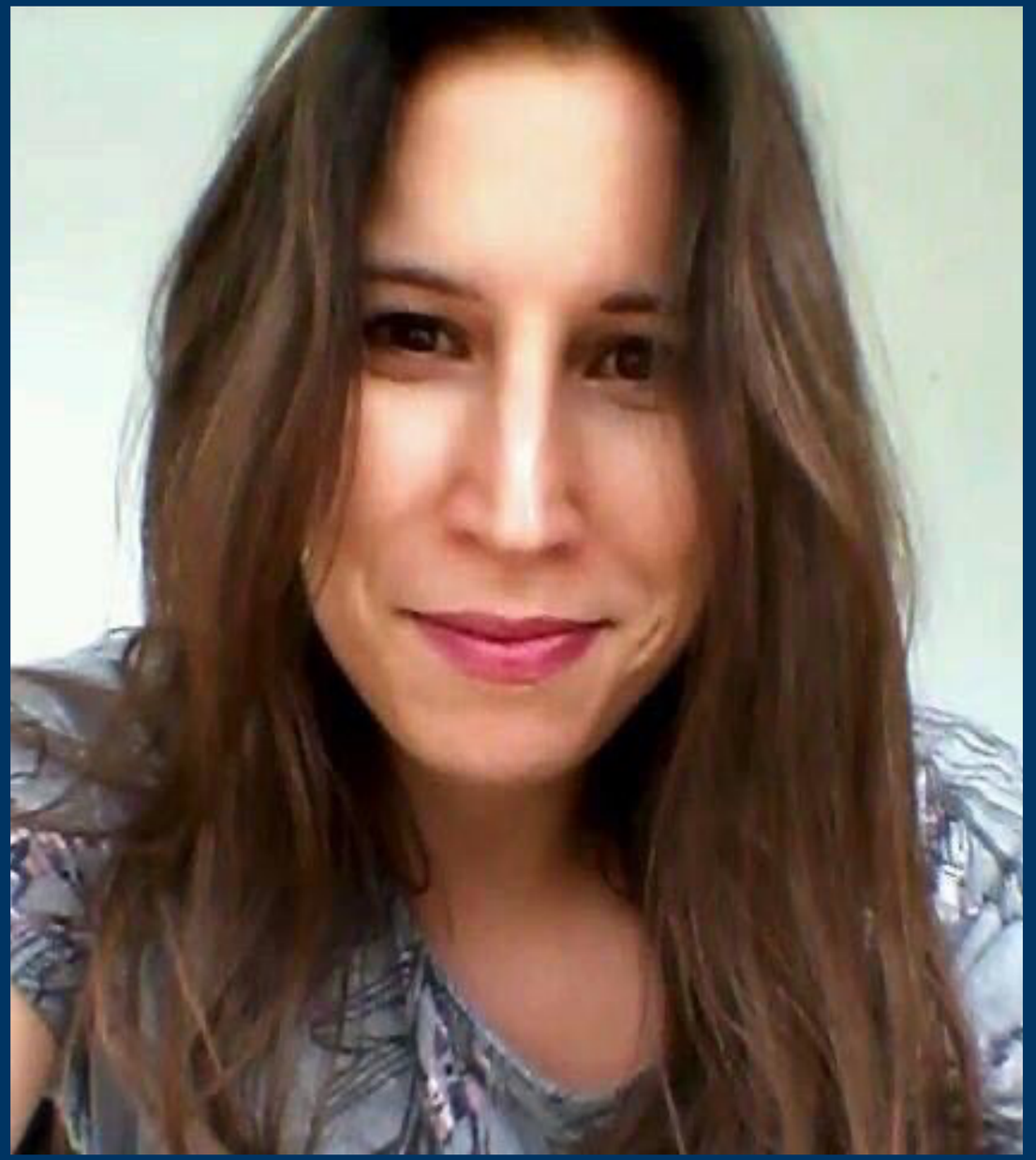
Nina Ranns

Cat owner,
Coffee drinker,
Chocolate lover,
Committee member,
Comma supervisor,
C++ nerd



Nina Ranns

Cat owner,
Coffee drinker,
Chocolate lover,
Committee member,
Comma supervisor,
C++ nerd here to talk about CTAD



- John Lakos
- Vittorio Romeo
- Rostislav Khlebnikov
- Alisdair Meredith
- An extended team of collaborators + Nina

What is in this talk

- What is Class Template Argument Deduction (CTAD) and how it makes our life easier

What is in this talk

- What is Class Template Argument Deduction (CTAD) and how it makes our life easier
- The principles of how CTAD works

What is in this talk

- What is Class Template Argument Deduction (CTAD) and how it makes our life easier
- The principles of how CTAD works
- Things to be aware of when it comes to CTAD

What isn’t in this talk

What isn’t in this talk

- In depth explanation of CTAD rules

What isn’t in this talk

- In depth explanation of CTAD rules
- Extensive tutorial on writing your own deduction guides

What isn’t in this talk

- In depth explanation of CTAD rules
- Extensive tutorial on writing your own deduction guides
- CTAD take down

What isn’t in this talk

- In depth explanation of CTAD rules
- Extensive tutorial on writing your own deduction guides
- CTAD take down or CTAD sales pitch

What isn’t in this talk

- In depth explanation of CTAD rules
- Extensive tutorial on writing your own deduction guides
- CTAD take down or CTAD sales pitch
- Popcorn

What is CTAD ?

Deducing function-template template argument

```
template<typename T>
void myFunc(const T&){}
```

```
myFunc<int>(1);      // invokes myFunc<int> specialisation
```

Deducing function-template template argument

```
template<typename T>
void myFunc(const T&){}
```

```
myFunc(1);      // deduces T = int, invokes myFunc<int> specialisation
```

Deducing function-template template argument

```
template<typename T>
void myFunc(const T&){}
```

Template argument deduction

```
myFunc(1);      // deduces T = int, invokes myFunc<int> specialisation
```

Deducing class-template template argument - pre C++17

```
template<typename T>
struct myStruct
{
    myStruct(const T&){};
};
```

```
myStruct<int> m(1);      // creates object of myStruct<int> type
```

Deducing class-template template argument - pre C++17

```
template<typename T>
struct myStruct
{
    myStruct(const T&){};
};
```

```
myStruct m(1);      // error in c++14, missing explicit template parameters
```

Deducing class-template template argument - pre C++17

```
template<typename T>
struct myStruct
{
    myStruct(const T&){};
};


template<typename T>
myStruct<T> make_myStruct(const T&);


auto m = make_myStruct(1);  // ok, make_myStruct deduces T=int
```

Deducing class-template template argument - C++17

```
template<typename T>
struct myStruct
{
    myStruct(const T&){};
};
```

```
myStruct m(1);      // deduces T=int, creates object of myStruct<int> type
```

Deducing class-template template argument - C++17

```
template<typename T>
struct myStruct
{
    myStruct(const T&){};
};
```

Class template argument deduction (CTAD)

```
myStruct m(1);      // deduces T=int, creates object of myStruct<int> type
```

```
std::vector<int> v{1,2,3};
```

```
std::vector<int> v1{1,2,3};
std::vector v2{1,2,3};
```

```
std::vector<int> v1{1,2,3};
std::vector v2{1,2,3};

std::array<int, 4> a1{1,2,3,4};
```

```
std::vector<int> v1{1,2,3};
std::vector v2{1,2,3};

std::array<int, 4> a1{1,2,3,4};
std::array a2{1,2,3,4};
```

```
std::vector<int> v1{1,2,3};
std::vector v2{1,2,3};

std::array<int, 4> a1{1,2,3,4};
std::array a2{1,2,3,4};


const std::string s = "example vector";
std::pair<std::vector<int>, std::string> p1{v, s};
```

```
std::vector<int> v1{1,2,3};
std::vector v2{1,2,3};

std::array<int, 4> a1{1,2,3,4};
std::array a2{1,2,3,4};


const std::string s = "example vector";
std::pair<std::vector<int>, std::string> p1{v, s};
std::pair p2{v, s};
```

```
std::vector<int> v1{1,2,3};
std::vector v2{1,2,3};

std::array<int, 4> a1{1,2,3,4};
std::array a2{1,2,3,4};


const std::string s = "example vector";
std::pair<std::vector<int>, std::string> p1{v, s};
std::pair p2{v, s};



auto it = std::find(s.begin(), s.end(), 'x');
std::tuple<std::string, std::string::const_iterator, char> t1{s, it, 'x'};
```

```
std::vector<int> v1{1,2,3};
std::vector v2{1,2,3};

std::array<int, 4> a1{1,2,3,4};
std::array a2{1,2,3,4};


const std::string s = "example vector";
std::pair<std::vector<int>, std::string> p1{v, s};
std::pair p2{v, s};



auto it = std::find(s.begin(), s.end(), 'x');
std::tuple<std::string, std::string::const_iterator, char> t1{s, it, 'x'};
std::tuple t2{s, it, 'x'};
```

With CTAD:

```
vector<string> vs{"Thank", " ", "you"," ", "Jeff", "!"};
auto jv = ranges::join_view(vs);
```

With CTAD:

```
vector<string> vs{"Thank", " ", "you"," ", "Jeff", "!"};
auto jv = ranges::join_view(vs);
```

Without CTAD:

```
vector<string> vs{"Thank", " ", "you"," ", "Jeff", "!"};
auto jv = ranges::join_view<ranges::ref_view<vector<string>>>(vs);
```

CTAD type support - C++17

```
template<typename T>
struct NonAggregate
{
   NonAggregate(const T&){};
   T t;
};

NonAggregate a(1); // ok since C++17
```

CTAD type support - C++20

```
template<typename T>
struct NonAggregate
{
   NonAggregate(const T&){};
   T t;
};

NonAggregate a(1); // ok since C++17

template<typename T>
struct Aggregate
{
   T t;
};

Aggregate b(1); // ok since C++20*
```

*P1021R4 (Spertus, Doumler, Smith)

CTAD type support - C++20

```
template<typename T>
struct NonAggregate
{
   NonAggregate(const T&){};
   T t;
};

NonAggregate a(1); // ok since C++17

template<typename T>
struct Aggregate
{
   T t;
};

Aggregate b(1); // ok since C++20*

template<typename T>
using myAlias = Aggregate<T>;

myAlias c('c'); // ok since C++20*
```

*P1021R4 (Spertus, Doumler, Smith)

```
template<typename T>
struct S
{
   S(const T&);
};



S x1 = 1;
S x2{1};
```

```
template<typename T = int>
struct S
{
   S();
   S(const T&);
};


S x1 = 1;
S x2{1};

S x3; // deduces S<int>
```

```
template<typename T = int>
struct S
{
   S();
   S(const T&);
};


S x1 = 1;
S x2{1};

S x3; // deduces S<int>

auto p1 = new S{1};
```

```
template<typename T = int>
struct S
{
   S();
   S(const T&);
};


S x1 = 1;
S x2{1};

S x3; // deduces S<int>

auto p1 = new S{1};

std::mutex mutex;
auto lock = std::lock_guard(mutex);
```

```
template<typename T = int>
struct S
{
   S();
   S(const T&);
};


S x1 = 1;
S x2{1};

S x3; // deduces S<int>

auto p1 = new S{1};

std::mutex mutex;
auto lock = std::lock_guard(mutex);

S          *p2 = new S{1};    // error, type of pointer is not deduced.
```

```
template<typename T = int>
struct S
{
   S();
   S(const T&);
};


S x1 = 1;
S x2{1};

S x3; // deduces S<int>

auto p1 = new S{1};

std::mutex mutex;
auto lock = std::lock_guard(mutex);

S           *p2 = new S{1};    // error, type of pointer is not deduced.
const S&     r1 = x1;          // error, reference type is not deduced.
```

```
template<typename T = int>
struct S
{
   S();
   S(const T&);
};


S x1 = 1;
S x2{1};

S x3; // deduces S<int>

auto p1 = new S{1};

std::mutex mutex;
auto lock = std::lock_guard(mutex);

S          *p2 = new S{1};    // error, type of pointer is not deduced.
const S&    r1 = x1;          // error, reference type is not deduced.
S         a1[] = {x1, x2, x3};// error, array type is not deduced.
```

```
template<typename T = int>
struct S
{
   S();
   S(const T&);
};


S x1 = 1;
S x2{1};

S x3; // deduces S<int>

auto p1 = new S{1};

std::mutex mutex;
auto lock = std::lock_guard(mutex);

auto       *p2 = new S{1};    // ok
const auto& r1 = x1;          // ok
auto       a1[] = {x1, x2, x3};// error
```

```
template<typename T>
struct S
{
   constexpr S(T){};
};



template<S s>
void foo();

foo<0>();  // ok in C++20, deduces foo<S<int>>
```

```
template<typename T>
struct S
{
   constexpr S(T){};
};



template<S s>
void foo();

foo<0>();  // ok in C++20, deduces foo<S<int>>



void bar(S s);  // error, no CTAD in function parameters
```

CTAD - how does it work ?

```
template<typename T>
struct X
{
   X(const T&);
   X(const std::optional<T>&);
};


X x = 1;
```

CTAD machinery

- create a set of notional template functions (deduce_X) based on the constructors of the primary class template*
- deduce T for ‘deduce_X(*initialiser*)’ using template argument deduction*
- If T is deduced, substitute T into the statement
- Evaluate statement with deduced type(s) substituted

```
template<typename T>
struct X
{
   X(const T&);
   X(const std::optional<T>&);
};


X x = 1; // deduces T = int, invokes X<int>::X(const int&)


template<typename T>
X<T> deduce_X(const T&);

template<typename T>
X<T> deduce_X(const std::optional<T>&);
```

```
template<typename T>
struct X
{
   X(const T&);
   X(const std::optional<T>&);
};


X x = 1; // deduces T = int, invokes X<int>::X(const int&)
```

```
template<typename T>
X<T> deduce_X(const T&);

template<typename T>
X<T> deduce_X(const std::optional<T>&);
```

Deduction guides

```
template<typename T>
struct X
{
   X(const T&);
   X(const std::optional<T>&);
};


X x = 1; // deduces T = int, invokes X<int>::X(const int&)
```

```
template<typename T>
X(const T&) -> X<T>;

template<typename T>
X(const std::optional<T>&)-> X<T>;
```

Deduction guides

Implicit deduction guides

```
template<typename T = int>
struct X
{
   X();
   X(const T&);
};



template<typename T = int>
X() -> X<T>;

template<typename T = int>
X(const T&) -> X(T);
```

Implicit deduction guides

```
template<myConcept T, typename... Args>
struct X
{
   X(const T&, const Args&...)

};
```

```
template<myConcept T, typename... Args>
X(const T&, const Args&..)-> X<T,Args…>;
```

Implicit deduction guides

```
template<typename T>
struct X
{
   template<typename U>
   X(const T&, const U&);

};



template<typename T, typename U>
X(const T&, const U&)-> X<T>;
```

Implicit deduction guides

```
template<class T>
concept Integral = std::is_integral<T>::value;

template<typename T = int>
struct S
{
   S();
   S(const T&) requires Integral<T>;
};


template<typename T> requires Integral<T>
S(T) -> S<bool>;
```

Implicit deduction guides

```
template<typename T = int>
struct X;


#1 Conditional deduction guide for X()
template<typename T = int>
X<T> deduce_X();
```

Implicit deduction guides

```
template<typename T = int>
struct X;


#1 Conditional deduction guide for X()
template<typename T = int>
X<T> deduce_X();


#2 Copy deduction guide for X(X)
template<typename T = int>
X<T> deduce_X(X<T>)
```

```
template<typename T>
struct X
{
   X(const T&);
   X(const std::optional<T>&);
};



template<typename T>
X(const T&) -> X<T>;   #1

template<typename T>
X(const std::optional<T>&) -> X<T>;  #2

template<typename T>
X(X<T>) -> X<T>; #3


X x = 1; // deduces X = int using #1
X y = x; // deduces X = int using #3
```

Writing your own deduction guide

```
template<class T>
class vector {
    ...
    template<class InputIterator>
    constexpr vector(InputIterator first, InputIterator last);
    ...
};
```

```
template<typename T>
vector(InputIterator first, InputIterator last) -> vector<T>;
```

```
template<class T>
class vector {
    ...
    template<class InputIterator>
    constexpr vector(InputIterator first, InputIterator last);
    ...
};
```

```
template<class T>
class vector {
    ...
    template<class InputIterator>
    constexpr vector(InputIterator first, InputIterator last);
    ...
};
```

```
template<typename T>
vector(
```

```
template<class T>
class vector {
    ...
    template<class InputIterator>
    constexpr vector(InputIterator first, InputIterator last);
    ...
};
```

```
template<typename InputIterator>
vector(InputIterator first, InputIterator last)
```

```
template<class T>
class vector {
    ...
    template<class InputIterator>
    constexpr vector(InputIterator first, InputIterator last);
    ...
};
```

```
template<typename InputIterator>
vector(InputIterator first, InputIterator last) ->
                vector<typename iterator_traits<InputIterator>::value_type>;
```

```
template<typename T>
struct Bob
{
};

template<>
struct Bob<int>
{
   Bob(int);
};
```

```
Bob x{1};  // T can’t be deduced
```

```
template<typename T>
struct Bob
{
};

template<>
struct Bob<int>
{
   Bob(int);
};


Bob(int) -> Bob<int>;



Bob x{1};  // ok, invokes Bob<int>::Bob(int)
```

CTAD machinery

- create a set of notional template functions (deduce_X) based on the constructors of the primary class template (deduce_X)*
- deduce T for ‘deduce_X(*initialiser*)’ using template argument deduction*
- If T is deduced, substitute T into the statement
- Evaluate statement with deduced type(s) substituted

CTAD machinery

- create a set of deduction guides for the class (implicit + user defined ones)

- deduce T from the deduction guides using template argument deduction*
- If T is deduced, substitute T into the statement
- Evaluate statement with deduced type(s) substituted

CTAD VS TAD

CTAD vs TAD - no forwarding references

```
template<typename T>
void foo(T&&);

int i = 3;
foo(i); // deduces T = int&, results in foo(int&)
foo(3); // deduces T = int, results in foo(int&&)
```

CTAD vs TAD - no forwarding references

```
template<typename T>
struct X
{
   X(const T&);  // #1 invoked for values we can’t move from
   X(T&&);       // #2 invoked for rvalues
};


int i = 3;
X<int> xl(i);  // invokes #1
X<int> xr(1);  // invokes #2


template<typename T>
X(const T&) -> X<T>;

template<typename T>
X(T&&) -> X<T>;
```

CTAD vs TAD - all or nothing

```
template<typename T, typename U>
struct X
{
   X(const T&, const U&);
};

X<int> x(1, 2); // no CTAD
```

CTAD vs TAD - all or nothing

```
template<typename T, typename U = char>
struct X
{
   X(const T&, const U&);
};

X<int> x(1, 2); // X<int, char> or X<int, int>?
```

CTAD vs TAD - all or nothing

```
template<typename... ARGS>
struct X
{
   X(ARGS&&...);
   X(int i, ARGS&&…);
};

X<int> x(1,'c');  // X<int> or X<int, char>
```

CTAD - what to be aware of

CTAD prefers copying over conversion

```
std::vector a{1,2,3};
std::vector b = a;

static_assert(std::is_same_v<decltype(a), decltype(b)>);
```

CTAD prefers copying over conversion

```
#include <vector>
template<class T>
auto make_vector(T&& t)
{
    return vector{std::forward<T>(t)};
}
int main()
{
    std::vector v1{1,2,3};


    auto v3 = make_vector(1);  // std::vector<int>
    auto v4 = make_vector(v1);

    static_assert(std::is_same_v<decltype(v4), std::vector<vector<int>>);
}
```

CTAD prefers copying over conversion

```
template<typename T>
struct Container;

Container<int> b;
Container      b2{b};  // Container<int> or Container<Container<int>> ?
```

CTAD prefers copying over conversion

```
#include <vector>
template<class T>
auto make_vector(T&& t)
{
    return vector{std::forward<T>(t)};
}
int main()
{
    std::vector v1{1,2,3};


    auto v3 = make_vector(1);  // std::vector<int>
    auto v4 = make_vector(v1);

    static_assert(std::is_same_v<decltype(v4), std::vector<vector<int>>);
}
```

CTAD prefers copying over conversion

```
#include <vector>
template<class T>
auto make_vector(T&& t)
{
    return vector<T>{std::forward<T>(t)};
}
int main()
{
    std::vector v1{1,2,3};


    auto v3 = make_vector(1);  // std::vector<int>
    auto v4 = make_vector(v1);

    static_assert(std::is_same_v<decltype(v4), std::vector<vector<int>>);
}
```

```
std::pair MyPair{1,'1'};    // std::pair<int, char>
std::tuple MyTuple{MyPair}; // What is the deduced type here ?
```

std::tuple<Types...>::tuple

Defined in header <tuple>

`constexpr tuple();`	(1)	(since C++11) (conditionally explicit)
`tuple( const Types&... args );`	(2)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class... UTypes >` `tuple( UTypes&&... args );`	(3)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class... UTypes >` `constexpr tuple( tuple<UTypes...>& other );`	(4)	(since C++23) (conditionally explicit)
`template< class... UTypes >` `tuple( const tuple<UTypes...>& other );`	(5)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class... UTypes >` `tuple( tuple<UTypes...>&& other );`	(6)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class... UTypes >` `constexpr tuple( const tuple<UTypes...>&& other );`	(7)	(since C++23) (conditionally explicit)
`template< class U1, class U2 >` `constexpr tuple( std::pair<U1, U2>& p );`	(8)	(since C++23) (conditionally explicit)
`template< class U1, class U2 >` `tuple( const std::pair<U1, U2>& p );`	(9)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class U1, class U2 >` `tuple( std::pair<U1, U2>&& p );`	(10)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class U1, class U2 >` `constexpr tuple( const std::pair<U1, U2>&& p );`	(11)	(since C++23) (conditionally explicit)
`template< tuple-like UTuple >` `constexpr tuple( UTuple&& u );`	(12)	(since C++23) (conditionally explicit)

std::tuple<Types...>::tuple

Defined in header <tuple>

`constexpr tuple();`	(1)	(since C++11) (conditionally explicit)
`tuple( const Types&... args );`	(2)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class... UTypes >` `tuple( UTypes&&... args );`	(3)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class... UTypes >` `constexpr tuple( tuple<UTypes...>& other );`	(4)	(since C++23) (conditionally explicit)
`template< class... UTypes >` `tuple( const tuple<UTypes...>& other );`	(5)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class... UTypes >` `tuple( tuple<UTypes...>&& other );`	(6)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class... UTypes >` `constexpr tuple( const tuple<UTypes...>&& other );`	(7)	(since C++23) (conditionally explicit)
`template< class U1, class U2 >` `constexpr tuple( std::pair<U1, U2>& p );`	(8)	(since C++23) (conditionally explicit)
`template< class U1, class U2 >` `tuple( const std::pair<U1, U2>& p );`	(9)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class U1, class U2 >` `tuple( std::pair<U1, U2>&& p );`	(10)	(since C++11) (constexpr since C++14) (conditionally explicit)
`template< class U1, class U2 >` `constexpr tuple( const std::pair<U1, U2>&& p );`	(11)	(since C++23) (conditionally explicit)
`template< tuple-like UTuple >` `constexpr tuple( UTuple&& u );`	(12)	(since C++23) (conditionally explicit)

- either
  - `sizeof...(Types)` is not `1`, or
  - (when `Types...` expands to `T` and `UTypes...` expands to `U`) `std::is_convertible_v<decltype(other), T>`, `std::is_constructible_v<T, decltype(other)>`, and `std::is_same_v<T, U>` are all `false`.
- These constructors are `explicit` if and only if `std::is_convertible_v<decltype(std::get<i>(FWD(other))), Ti>` is `false` for at least one `i`.

- These constructors are defined as deleted if the initialization of any element that is a reference would bind it to a temporary object. (since C++23)

8-11) Pair constructor. Constructs a 2-element tuple with each element constructed from the corresponding element of `p`.

Formally, let `FWD(p)` be `std::forward<decltype(p)>(p)`, initializes the first element with `std::get<0>(FWD(p))` and the second element with `std::get<1>(FWD(p))`.

- This overload participates in overload resolution only if
  - `sizeof...(Types) == 2`,
  - `std::is_constructible_v<T0, decltype(std::get<0>(FWD(p)))>` is `true`, and
  - `std::is_constructible_v<T1, decltype(std::get<1>(FWD(p)))>` is `true`
- The constructor is `explicit` if and only if `std::is_convertible_v<decltype(std::get<0>(FWD(p))), T0>` or `std::is_convertible_v<decltype(std::get<1>(FWD(p))), T1>` is `false`.

- These constructors are defined as deleted if the initialization of any element that is a reference would bind it to a temporary object. (since C++23)

12) *tuple-like* constructor. Constructs a tuple with each element constructed from the corresponding element of `u`.

Formally, for all `i`, initializes ith element of the tuple with `std::get<i>(std::forward<UTuple>(u))`.

- This overload participates in overload resolution only if
  - `std::same_as<std::remove_cvref_t<UTuple>, std::tuple>` is `false`,
  - `std::remove_cvref_t<UTuple>` is not a specialization of `std::ranges::subrange`,
  - `sizeof...(Types)` equals `std::tuple_size_v<std::remove_cvref_t<UTuple>>`,

deduction guides for `std::tuple`

Defined in header <tuple>

```
template<class... UTypes>
tuple(UTypes...) -> tuple<UTypes...>;
```
(1) (since C++17)

```
template<class T1, class T2>
tuple(std::pair<T1, T2>) -> tuple<T1, T2>;
```
(2) (since C++17)

```
template<class Alloc, class... UTypes>
tuple(std::allocator_arg_t, Alloc, UTypes...) -> tuple<UTypes...>;
```
(3) (since C++17)

```
template<class Alloc, class T1, class T2>
tuple(std::allocator_arg_t, Alloc, std::pair<T1, T2>) -> tuple<T1, T2>;
```
(4) (since C++17)

```
template<class Alloc, class... UTypes>
tuple(std::allocator_arg_t, Alloc, tuple<UTypes...>) -> tuple<UTypes...>;
```
(5) (since C++17)

These deduction guides are provided for `std::tuple` to account for the edge cases missed by the implicit deduction guides, in particular, non-copyable arguments and array to pointer conversion.

deduction guides for `std::tuple`

Defined in header <tuple>

```
template<class... UTypes>
tuple(UTypes...) -> tuple<UTypes...>;                                   (1)  (since C++17)

template<class T1, class T2>
tuple(std::pair<T1, T2>) -> tuple<T1, T2>;                              (2)  (since C++17)

template<class Alloc, class... UTypes>
tuple(std::allocator_arg_t, Alloc, UTypes...) -> tuple<UTypes...>;      (3)  (since C++17)

template<class Alloc, class T1, class T2>
tuple(std::allocator_arg_t, Alloc, std::pair<T1, T2>) -> tuple<T1, T2>; (4)  (since C++17)

template<class Alloc, class... UTypes>
tuple(std::allocator_arg_t, Alloc, tuple<UTypes...>) -> tuple<UTypes...>; (5)  (since C++17)
```

These deduction guides are provided for `std::tuple` to account for the edge cases missed by the implicit deduction guides, in particular, non-copyable arguments and array to pointer conversion.

```
std::pair MyPair{1,'1'};    // std::pair<int, char>
std::tuple MyTuple{MyPair}; // What is the deduced type here ?
```

```
std::pair MyPair{1,'1'};    // std::pair<int, char>
std::tuple MyTuple{MyPair}; // std::tuple<int, char>
```

Unintentional CTAD

```
template<typename T>
struct Bob
{
   Bob(const T&){ };
};



Bob  b{1};
```

Unintentional CTAD

```
template<typename T>
struct Bob
{
   Bob(const T&){ };
};



Bob  b{1};
```

Unintentional CTAD

```
template<typename T>
struct Bob
{
   using MyT = std::remove_cvref<T>::type;
   Bob(const MyT&){ };
};


Bob  b{1}; // oops, T can no longer be deduced
```

Unintentional CTAD

```
template<typename T>
struct Bob
{
   using _T = std::type_identity_t<T>;
   Bob(const _T&){ };
};


Bob  b{1};    // error, no deduction possible
Bob  b2 = b1; // ok
```

To CTAD or not to CTAD ?

User perspective - not specifying template arguments

- Be careful in the same situation as with function template argument deduction
- Be careful when writing generic code
- Be careful when wanting to put a container into a container
- Be careful when initialising with an element of the same template class
- Read the manual to see how the type behaves

Vendor perspective - writing template classes

- Test implicit guides to see if they give the right deduction
- Provide additional guides if needed
- Consider whether you want to disable CTAD by making your constructors non deduced context

Q & A

With thanks to Ville Voutilainen :)

Thank you for coming to my talk ! :)