Better CTAD for C++20

Timur Doumler

@timur_audio

C++Now
7 May 2019

CppCon 2018: “Grill the Committee”

C++14

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

C++17

C++14

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

C++17

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

C++14

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

std::array<int, 3> a = {1, 2, 3};
```

C++17

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

C++14

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

std::array<int, 3> a = {1, 2, 3};
```

C++17

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

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

C++14

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

std::array<int, 3> a = {1, 2, 3};

std::lock_guard<std::shared_timed_mutex>
lock(mtx);
```

C++17

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

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

C++14

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

std::array<int, 3> a = {1, 2, 3};

std::lock_guard<std::shared_timed_mutex>
lock(mtx);
```

C++17

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

std::array a = {1, 2, 3};

std::lock_guard lock(mtx);
```

C++14

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

std::array<int, 3> a = {1, 2, 3};

std::lock_guard<std::shared_timed_mutex>
lock(mtx);

auto cmp =
  [](int a, int b) { return a > b; };

std::set<int, decltype(cmp)> s({1, 2, 3}, cmp);
```

C++17

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

std::array a = {1, 2, 3};

std::lock_guard lock(mtx);
```

C++14

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

std::array<int, 3> a = {1, 2, 3};

std::lock_guard<std::shared_timed_mutex>
lock(mtx);

auto cmp =
  [](int a, int b) { return a > b; };

std::set<int, decltype(cmp)> s({1, 2, 3}, cmp);
```

C++17

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

std::array a = {1, 2, 3};

std::lock_guard lock(mtx);


std::set s(
  {2, 3, 5, 7, 11},
  [](int a, int b) { return a > b; });
```

C++14

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

std::array<int, 3> a = {1, 2, 3};

std::lock_guard<std::shared_timed_mutex>
lock(mtx);

auto cmp =
  [](int a, int b) { return a > b; };

std::set<int, decltype(cmp)> s({1, 2, 3}, cmp);

auto p = std::make_pair(42, "Hello");

auto t = std::make_tuple(0, widget, false);
```

C++17

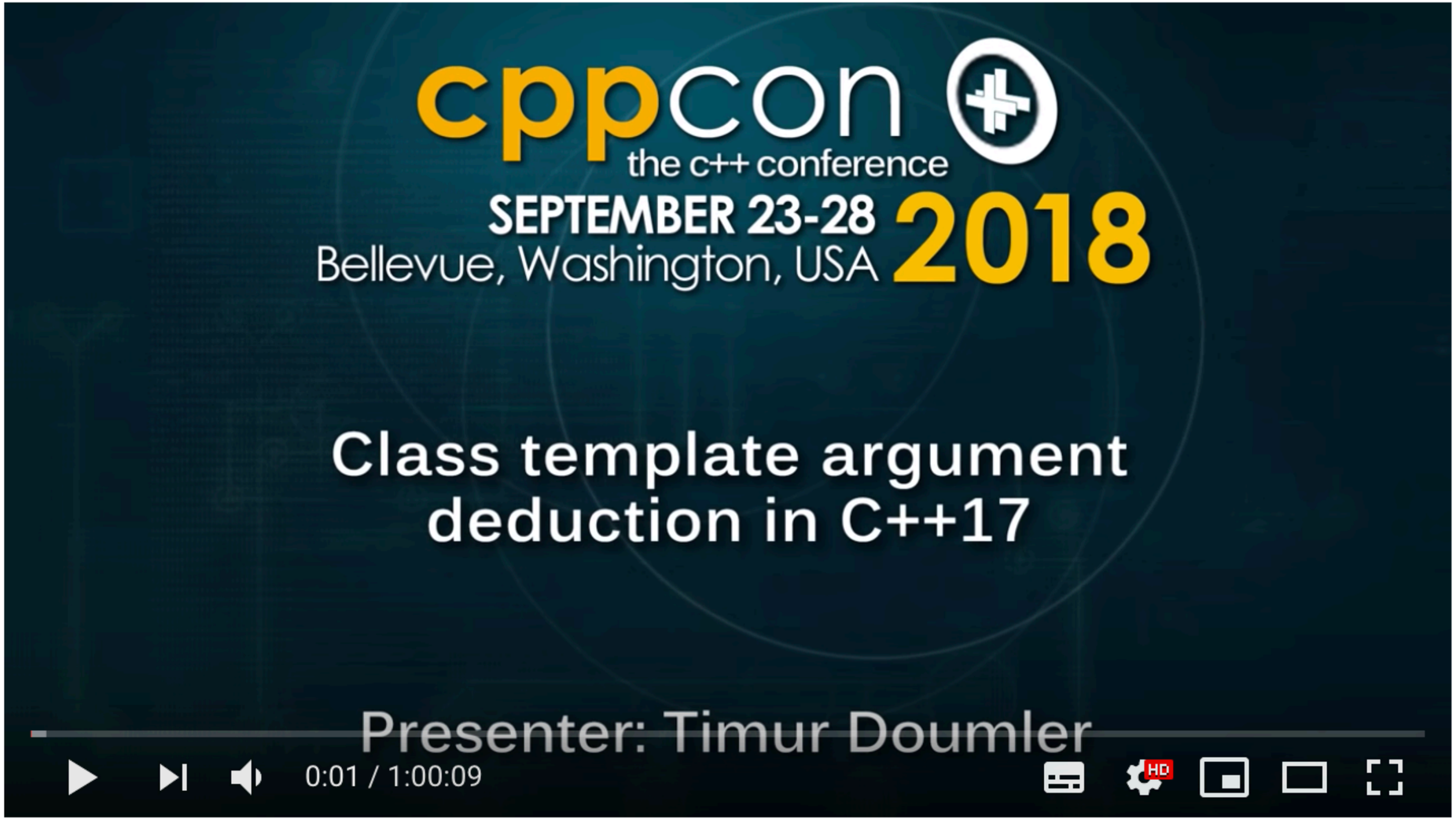
```
std::vector v = {1, 2, 3};

std::array a = {1, 2, 3};

std::lock_guard lock(mtx);


std::set s(
  {2, 3, 5, 7, 11},
  [](int a, int b) { return a > b; });
```

C++14

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

std::array<int, 3> a = {1, 2, 3};

std::lock_guard<std::shared_timed_mutex>
lock(mtx);

auto cmp =
  [](int a, int b) { return a > b; };

std::set<int, decltype(cmp)> s({1, 2, 3}, cmp);

auto p = std::make_pair(42, "Hello");

auto t = std::make_tuple(0, widget, false);
```

C++17

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

std::array a = {1, 2, 3};

std::lock_guard lock(mtx);


std::set s(
  {2, 3, 5, 7, 11},
  [](int a, int b) { return a > b; });

std::pair p(42, "Hello");

std::tuple t(0, widget, false);
```



CppCon 2018: Timur Doumler “Class template argument deduction in C++17”

This talk

This talk

1. [over.match.class.deduct]

This talk

1. [over.match.class.deduct]

2. P1021: “Filling holes in CTAD”

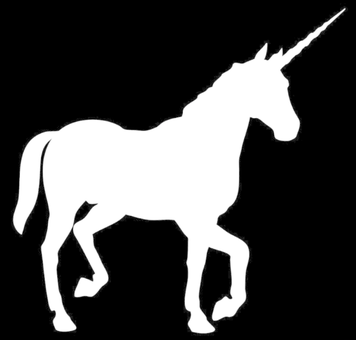
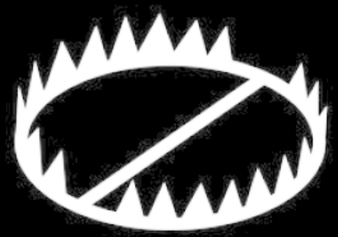
   Mike Spertus, Timur Doumler, Richard Smith

This talk

1. [over.match.class.deduct]

2. P1021: “Filling holes in CTAD”

   Mike Spertus, Timur Doumler, Richard Smith

3.  

1. To CTAD or not to CTAD?

```
std::vector v = {1};
```

```cpp
std::vector v = {1};  // template-name with no <> is a placeholder
```

```cpp
std::vector v = {1};  // template-name with no <> is a placeholder

auto i = 1;           // these are also placeholders
decltype(auto) j = 1;
```

```cpp
std::vector v = {1};  // template-name with no <> is a placeholder

auto i = 1;           // these are also placeholders
decltype(auto) j = 1;

ConceptName auto i = 1;
ConceptName decltype(auto) j = 1;
```

What does CTAD do?

– synthesises some function templates
– does FTAD + overload resolution on them

2. The “deduction candidate” overload set

2. The “deduction candidate” overload set

– from constructors
– from deduction guides
– the “default” candidate
– the “copy deduction” candidate

2. The “deduction candidate” overload set

– from constructors

– from deduction guides

– the “default” candidate

– the “copy deduction” candidate

```cpp
template <typename T, typename U>
struct pair
{
    T first;
    U second;

    pair(const T& _first, const U& _second)
        : first(_first),
          second(_second)
    {}

    pair(T&& _first, U&& _second)
        : first(std::move<T>(_first)),
          second(std::move<U>(_second))
    {}
};
```

```cpp
template <typename T, typename U>
struct pair
{
    T first;
    U second;

    pair(const T& _first, const U& _second)
        : first(_first),
          second(_second)
    {}

    pair(T&& _first, U&& _second)
        : first(std::move<T>(_first)),
          second(std::move<U>(_second))
    {}
};

pair p = {42, true};
```

```
template <typename T, typename U>
struct pair
{

    pair(const T& _first, const U& _second);

    pair(T&& _first, U&& _second);

};

pair p = {42, true};
```

```
template <typename T, typename U>
struct pair
{

    pair(const T& _first, const U& _second);

    pair(T&& _first, U&& _second);

};

pair p = {42, true};
```

```
template <typename T, typename U>
pair(const T& _first, const U& _second);
```

```
template <typename T, typename U>
pair(T&& _first, U&& _second);
```

```
pair p = {42, true};
```

```
  template <typename T, typename U>
  pair<T, U> pair(const T& _first, const U& _second);
```

```
  template <typename T, typename U>
  pair<T, U> pair(T&& _first, U&& _second);
```

```
pair p = {42, true};
```

```
template <typename T, typename U>
pair<T, U> pair(const T& _first, const U& _second);
```

```
template <typename T, typename U>
pair<T, U> pair(T&& _first, U&& _second);
```

template argument deduction + overload resolution

```
pair p = {42, true};
```

```
template <typename T, typename U>
pair<T, U> pair(const T& _first, const U& _second);
```

```
template <typename T, typename U>
pair<T, U> pair(T&& _first, U&& _second);
```

These are NOT forwarding refs!

template argument deduction + overload resolution

```
pair p = {42, true};
```

```
template <typename T, typename U>
pair<T, U> pair(T&& _first, U&& _second);
```

Success!

```
pair p = {42, true};
```

```
template <typename T, typename U>
pair<T, U> pair(T&& _first, U&& _second);
```

pair<int, bool> **deduced!**

```
pair p = {42, true};
```

```
pair<int, bool> p = {42, true};
```

```
pair<int, bool> p = {42, true};
```

This can still

– actually call some other constructor

– fail

```
template <typename T>
struct Widget
{
private:
    Widget(T) {}
};

Widget w(0);
```

```
template <typename T>
struct Widget
{
private:
    Widget(T) {}
};

Widget w(0);  // Error: calling private constructor of Widget<int>
```

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

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

int main() {
    Widget w(3);   // what does this do?
}
```

```cpp
template <typename T>
struct Widget {
};
```

CTAD considers primary template only

```cpp
template<>
struct Widget<int> {
    Widget(int) {}
};

int main() {
    Widget w(3);   // Error: deduction failed!
}
```

Specialisations are ignored!

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

template<>
struct Widget<int> {
    Widget(double&) {}
};

int main() {
    Widget w(3);   // what does this do?
}
```

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

```
template<>
struct Widget<int> {
    Widget(double&) {}
};

int main() {
    Widget w(3);   // what does this do?
}
```

```cpp
template <typename T>
struct Widget {
    Widget(T) {}
};

template<>
struct Widget<int> {
    Widget(double&) {}
};

int main() {
    Widget<int> w(3);
}
```

CTAD successful!

```
template<>
struct Widget<int> {
    Widget(double&) {}
};

int main() {
    Widget<int> w(3);
}
```

```
template<>
struct Widget<int> {
    Widget(double&) {}
};

int main() {
    Widget<int> w(3);   // Error: no matching c'tor
}
```

```cpp
template <typename T>
struct Widget
{
    template <typename U>
    Widget(T, U) {}
};

int main() {
    Widget w(3, true);
}
```

Templated constructor?

```cpp
template <typename T>
struct Widget
{
    template <typename U>
    Widget(T, U) {}
};

int main() {
    Widget w(3, true);
}
```

```
template <typename T, typename U>
Widget(T, U) {}
```

```
int main() {
    Widget w(3, true);
}
```

```
    template <typename T, typename U>
    Widget<T> Widget(T, U) {}


int main() {
    Widget w(3, true);
}
```

```
    template <typename T, typename U>
    Widget<T> Widget(T, U) {}
```

template argument deduction + overload resolution

```
int main() {
    Widget w(3, true);
}
```

```
    template <typename T, typename U>
    Widget<T> Widget(T, U) {}
```

Success! `Widget<int>` **deduced!**

```
int main() {
    Widget w(3, true);
}
```

```cpp
template <typename T>
struct Widget
{
    template <typename U>
    Widget(T, U) {}
};

int main() {
    Widget<int> w(3, true);
}
```

2. The “deduction candidate” overload set

– from constructors
– from deduction guides
– the “default” candidate
– the “copy deduction” candidate

2. The “deduction candidate” overload set

– from constructors

– from deduction guides

– the “default” candidate

– the “copy deduction” candidate

```
template <typename Iter>
class vector { /* ... */ };
```

```cpp
template <typename Iter>
class vector { /* ... */ };

template <typename Iter>
vector(Iter begin, Iter end)
  -> vector<typename std::iterator_traits<Iter>::value_type>;
```

```cpp
template <typename Iter>
class vector { /* ... */ };

template <typename Iter>
vector(Iter begin, Iter end)
  -> vector<typename std::iterator_traits<Iter>::value_type>;

some_range<int> r;
vector v(r.begin(), r.end());
```

```cpp
template <typename Iter>
class vector { /* ... */ };
```

```cpp
template <typename Iter>
vector(Iter begin, Iter end)
  -> vector<typename std::iterator_traits<Iter>::value_type>;
```

```cpp
some_range<int> r;
vector v(r.begin(), r.end());
```

```cpp
template <typename Iter>
class vector { /* ... */ };

template <typename Iter>
vector(Iter begin, Iter end)
  -> vector<typename std::iterator_traits<Iter>::value_type>;

some_range<int> r;
vector<int> v(r.begin(), r.end());
```

```cpp
template <typename Iter>
class vector { /* ... */ };
```

```cpp
some_range<int> r;
vector v(r.begin(), r.end());  // Error: no deduction guide

template <typename Iter>
vector(Iter begin, Iter end)
  -> vector<typename std::iterator_traits<Iter>::value_type>;
```

```cpp
template <typename Iter>
class vector { /* ... */ };

template <typename Iter>
vector(Iter begin, Iter end)
  -> vector<typename std::iterator_traits<Iter>::value_type>;

some_range<int> r;
vector v(r.begin(), r.end());  // deduces vector<int>
```

```cpp
template <typename Iter>
class vector { /* ... */ };

template <typename Iter>
vector(Iter begin, Iter end)
  -> vector<typename std::iterator_traits<Iter>::value_type>;

some_range<int> r;
vector v(r.begin(), r.end());  // deduces vector<int>
vector v{r.begin(), r.end()};
```

```cpp
template <typename Iter>
class vector { /* ... */ };

template <typename Iter>
vector(Iter begin, Iter end)
  -> vector<typename std::iterator_traits<Iter>::value_type>;

some_range<int> r;
vector v(r.begin(), r.end());  // deduces vector<int>
vector v{r.begin(), r.end()};  // deduces vector<some_range<int>::iterator>
```

```
template <typename Iter>
vector(Iter begin, Iter end)
  -> vector<typename std::iterator_traits<Iter>::value_type>;
```

```
template <typename Iter>
vector(Iter begin, Iter end)
  -> vector<
        std::enable_if_t<
            std::is_base_of_v<
                std::input_iterator_tag,
                typename std::iterator_traits<Iter>::iterator_category>,
            typename std::iterator_traits<Iter>::value_type>>;
```

2. The “deduction candidate” overload set

– from constructors
– from deduction guides
– the “default” candidate
– the “copy deduction” candidate

2. The “deduction candidate” overload set

– from constructors
– from deduction guides
– the “default” candidate
– the “copy deduction” candidate

For a primary class template `C`,
add to the CTAD overload set:

– If `C` is not defined or does not declare any constructors:
  an additional function template derived from a hypothetical constructor `C()`.

– An additional function template derived from a hypothetical constructor `C(C)`.

For a primary class template `C`,
add to the CTAD overload set:

– If `C` is not defined or does not declare any constructors:
  an additional function template derived from a hypothetical constructor `C()`.

– An additional function template derived from a hypothetical constructor `C(C)`.

```
template <typename T = void>
struct less;

less l;  // should work!
```

```
template <typename T = void>
struct less;

template <typename T = void>
less() -> less<T>;

less l;  // works :)
```

Synthesised “default candidate”

For a primary class template `C`,
add to the CTAD overload set:

– If `C` is not defined or does not declare any constructors:
  an additional function template derived from a hypothetical
  constructor `C()`.

– An additional function template derived from a hypothetical
  constructor `C(C)`.

```
template <typename T>
class Widget {};

Widget<int> w1;
Widget w2 = w1; // this should work!
```

```cpp
template <typename T>
class Widget {};

template <typename T>
Widget(Widget<T>) -> Widget<T>;

Widget<int> w1;
Widget w2 = w1; // works :)
```

Synthesised “copy deduction candidate”

```cpp
template <typename... Types>
struct tuple
{
    tuple(const Types&...);
};

tuple t0(0);       // tuple<int>
tuple t1(t0);      // tuple<int>
```

```
template <typename... Types>
struct tuple
{
    tuple(const Types&...);
};

tuple t0(0);       // tuple<int>
tuple t1(t0);      // tuple<int>
tuple t2(t0, t1); // tuple<tuple<int>, tuple<int>>
```

12.3.1.8 Class template argument deduction [over.match.class.deduct]

When resolving a placeholder for a deduced class type ([dcl.type.class.deduct]) where the *template-name* names a primary class template `C`, a set of functions and function templates is formed comprising:

- If `C` is defined, for each constructor of `C`, a function template with the following properties:
  - The template parameters are the template parameters of `C` followed by the template parameters (including default template arguments) of the constructor, if any.
  - The types of the function parameters are those of the constructor.
  - The return type is the class template specialization designated by `C` and template arguments corresponding to the template parameters of `C`.
- If `C` is not defined or does not declare any constructors, an additional function template derived as above from a hypothetical constructor `C()`.
- An additional function template derived as above from a hypothetical constructor `C(C)`, called the *copy deduction candidate*.
- For each *deduction-guide*, a function or function template with the following properties:
  - The template parameters, if any, and function parameters are those of the *deduction-guide*.
  - The return type is the *simple-template-id* of the *deduction-guide*.

Initialization and overload resolution are performed as described in [dcl.init] and [over.match.ctor], [over.match.copy], or [over.match.list] (as appropriate for the type of initialization performed) for an object of a hypothetical class type, where the selected functions and function templates are considered to be the constructors of that class type for the purpose of forming an overload set, and the initializer is provided by the context in which class template argument deduction was performed. As an exception, the first phase in [over.match.list] (considering initializer-list constructors) is omitted if the initializer list consists of a single expression of type *cv* `U`, where `U` is a specialization of `C` or a class derived from a specialization of `C`. If the function or function template was generated from a constructor or *deduction-guide* that had an *explicit-specifier*, each such notional constructor is considered to have that same *explicit-specifier*. All such notional constructors are considered to be public members of the hypothetical class type.

P1021R3
Mike Spertus, Symantec
mike_spertus@symantec.com
Timur Doumler
papers@timur.audio
Richard Smith
richardsmith@google.com
2018-11-26
Audience: Core Working Group

Filling holes in Class Template Argument Deduction

This paper proposes filling several gaps in Class Template Argument Deduction.

Document revision history

R0, 2018-05-07: Initial version
R1, 2018-10-07: Refocused paper on filling holes in CTAD
R2, 2018-11-26: Following EWG guidance, removed proposal for CTAD from partial template argument lists
R3, 2019-01-21: Add wording and change target to Core Working Group

P1021 adds for C++20:

– CTAD for aggregates
– CTAD for alias templates
– CTAD from inherited constructors

P1021 adds for C++20:

– CTAD for aggregates
– CTAD for alias templates
– CTAD from inherited constructors

C++17

C++20

```
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

aggr_pair p = {1, true}; // Error
```

C++17

C++20

```
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

template <typename T, typename U>
aggr_pair(T, U) -> aggr_pair<T, U>;

aggr_pair p = {1, true}; // OK
```

C++17

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

template <typename T, typename U>
aggr_pair(T, U) -> aggr_pair<T, U>;

aggr_pair p = {1, true}; // OK
```

C++20

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

aggr_pair p = {1, true}; // OK
```

C++17

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

template <typename T, typename U>
aggr_pair(T, U) -> aggr_pair<T, U>;

aggr_pair p = {1, true}; // OK

aggr_pair p(1, true); // ?
```

C++20

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

aggr_pair p = {1, true}; // OK
```

C++17

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

template <typename T, typename U>
aggr_pair(T, U) -> aggr_pair<T, U>;

aggr_pair p = {1, true}; // OK

aggr_pair p(1, true); // Error:
                      // no matching ctor
                      // for aggr_pair<int, bool>
```

C++20

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

aggr_pair p = {1, true}; // OK
```

C++17

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

template <typename T, typename U>
aggr_pair(T, U) -> aggr_pair<T, U>;

aggr_pair p = {1, true}; // OK

aggr_pair p(1, true); // Error:
                      // no matching ctor
                      // for aggr_pair<int, bool>
```

C++20

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

aggr_pair p = {1, true}; // OK

aggr_pair p(1, true); // OK (P0960)
```

```
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
}
```

Is this an “aggregate”?

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
}
```

Is this an “aggregate”?

In C++20 (since P1008):

An *aggregate* is an array or a class ([class]) with

- no user-declared or inherited constructors ([class.ctor]),
- no private or protected non-static data members ([class.access]),
- no virtual functions ([class.virtual]), and
- no virtual, private, or protected base classes ([class.mi]).

```cpp
template <typename Base>
struct maybe_aggregate : public Base
{
};
```

Is this an “aggregate”?

In C++20 (since P1008):

An *aggregate* is an array or a class ([class]) with

— no user-declared or inherited constructors ([class.ctor]),

— no private or protected non-static data members ([class.access]),

— no virtual functions ([class.virtual]), and

— no virtual, private, or protected base classes ([class.mi]).

```cpp
template <typename Base>
struct maybe_aggregate : public Base
{
};
```

Is this an “aggregate”?

In C++20 (since P1008):

An *aggregate* is an array or a class ([class]) with

- no user-declared or inherited constructors ([class.ctor]),
- no private or protected non-static data members ([class.access]),
- no virtual functions ([class.virtual]), and
- no virtual, private, or protected base classes ([class.mi]).

—> it’s a “potentially aggregate” template
if the above is true,
not considering dependent base classes.

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

aggr_pair p(1, true);
```

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};

aggr_pair p(1, true);
```

The *elements* of an aggregate are:

- for an array, the array elements in increasing subscript order, or
- for a class, the direct base classes in declaration order, followed by the direct non-static data members ([class.mem]) that are not members of an anonymous union, in declaration order.

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};
```

```cpp
template <typename T, typename U>
aggr_pair(T, U) -> aggr_pair<T, U>;
```

```cpp
aggr_pair p(1, true);
```

Synthesised “aggregate deduction candidate”

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};
```

```cpp
template <typename T, typename U>
aggr_pair(T, U) -> aggr_pair<T, U>;
```

```cpp
aggr_pair p(1, true);
```

Synthesised “aggregate deduction candidate”

– no brace elision

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
};
```

```cpp
template <typename T, typename U>
aggr_pair(T, U) -> aggr_pair<T, U>;
```

```cpp
aggr_pair p(1, true);
```

Synthesised “aggregate deduction candidate”

– no brace elision

– correctly initialise elements with no initialisers

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
    int i;
};
```

```cpp
aggr_pair p(1, true);
```

Synthesised “aggregate deduction candidate”

– no brace elision

– correctly initialise elements with no initialisers

```cpp
template <typename T, typename U>
struct aggr_pair
{
    T t;
    U u;
    int i;
};

template <typename T, typename U>
aggr_pair(T, U, int = {})
  -> aggr_pair<T, U>;

aggr_pair p(1, true);
```

Synthesised “aggregate deduction candidate”

– no brace elision

– correctly initialise elements with no initialisers

P1021 adds for C++20:

– CTAD for aggregates

– CTAD for alias templates

– CTAD from inherited constructors

```
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

```cpp
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

C++17

C++20

```cpp
std::pmr::vector v{1, 2, 3};  // Error
```

```
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

C++17

C++20

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

```cpp
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

C++17

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

C++20

```cpp
std::pmr::vector v{1, 2, 3};
```

```cpp
template<Dimension D, Unit U, Number Rep>
class quantity;

template<Dimension D, Unit U, Number Rep>
quantity(Rep r) -> quantity<D, U, Rep>;

template<Unit U = meter, Number Rep = double>
using length = quantity<dimension_length, U, Rep>;

units::length d1(3);    // OK in C++20: quantity<dimension_length, meter, int>
units::length d2(3.14); // OK in C++20: quantity<dimension_length, meter, double>
```

—> Mateusz Pusz: Implementing a Physical Units Library for C++

```cpp
template <typename T, typename U>
struct pair;

template <typename X>
using int_pair = pair<int, X>;
```

```cpp
int_pair ip(0, "Meow");
```

```cpp
template <typename T, typename U>
struct pair;

template <typename X>
using int_pair = pair<int, X>;

template <typename T, typename U>
pair(T, U) -> pair<T, U>;
```

Deduction guide

```cpp
int_pair ip(0, "Meow");
```

```cpp
template <typename T, typename U>
struct pair;

template <typename X>
using int_pair = pair<int, X>;

template <typename T, typename U>
pair(T, U) -> pair<T, U>;
```

Deduction guide:

1. **Deduce return type from type alias**

```cpp
int_pair ip(0, "Meow");
```

```
template <typename T, typename U>
struct pair;

template <typename X>
using int_pair = pair<int, X>;

template <typename T, typename U>
pair(T, U) -> pair<int, X>;
```

Deduction guide:

1. **Deduce return type from type alias**

```
int_pair ip(0, "Meow");
```

```
template <typename T, typename U>
struct pair;

template <typename X>
using int_pair = pair<int, X>;

template <typename X>
pair(int, X) -> pair<int, X>;
```

Deduction guide:

1. **Deduce return type from type alias**
2. **Substitute back into deduction guide**

```
int_pair ip(0, "Meow");
```

```cpp
template <typename T, typename U>
struct pair;

template <typename X>
using int_pair = pair<int, X>;

template <typename X>
int_pair(int, X) -> int_pair<X>;
```

Deduction guide:

1. **Deduce return type from type alias**
2. **Substitute back into deduction guide**
3. **Rewrite in terms of type alias**

```cpp
int_pair ip(0, "Meow");
```

```cpp
template <typename T, typename U>
struct pair;

template <typename X>
using int_pair = pair<int, X>;

template <typename X>
int_pair(int, X) -> int_pair<X>;
```

Deduction guide:

1. **Deduce return type from type alias**
2. **Substitute back into deduction guide**
3. **Rewrite in terms of type alias**

```cpp
int_pair ip(0, "Meow");   // Deduces int_pair<const char*> :)
```

```
template<Dimension D, Unit U, Number Rep>
class quantity;

template<Dimension D, Unit U, Number Rep>
quantity(Rep r) -> quantity<D, U, Rep>;

template<Unit U = meter, Number Rep = double>
using length = quantity<dimension_length, U, Rep>;

units::length d1(3);          // quantity<dimension_length, meter, int>
units::length<mile> d2(3);    // quantity<dimension_length, mile, double>
```

```
template<Dimension D, Unit U, Number Rep>
class quantity;

template<Dimension D, Unit U, Number Rep>
quantity(Rep r) -> quantity<D, U, Rep>;

template<Unit U = meter, Number Rep = double>
using length = quantity<dimension_length, U, Rep>;

units::length d1(3);          // quantity<dimension_length, meter, int>
units::length<mile> d2(3);    // quantity<dimension_length, mile, double>
units::length<> d3(3);        // quantity<dimension_length, meter, double>
```

```
template<Dimension D, Unit U, Number Rep>
class quantity;

template<Dimension D, Unit U, Number Rep>
quantity(Rep r) -> quantity<D, U, Rep>;

template<Unit U = meter, Number Rep = double>
using length = quantity<dimension_length, U, Rep>;

units::length d1(3);          // quantity<dimension_length, meter, int>
units::length<mile> d2(3);    // quantity<dimension_length, mile, double>
units::length<> d3(3);        // quantity<dimension_length, meter, double>
```

Remember: you only get CTAD when you don’t have <>

P1021 adds for C++20:

– CTAD for aggregates
– CTAD for alias templates
– CTAD from inherited constructors

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

Widget w(1);
```

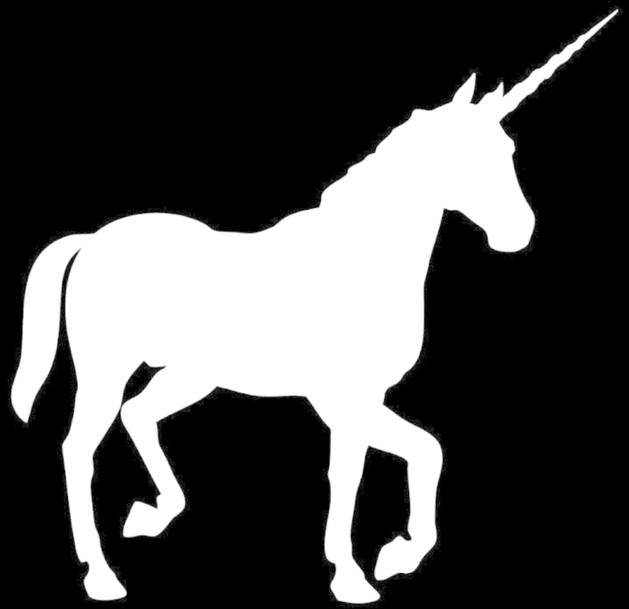
```cpp
template <typename T>
struct WidgetBase
{
    WidgetBase(T) {}
};

template <typename T>
struct Widget : public WidgetBase<T>
{
    using WidgetBase<T>::WidgetBase;
};

Widget w(1);   // Error in C++17! :(
```

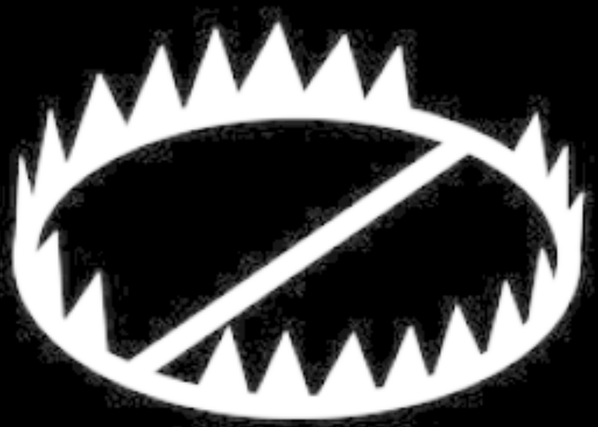
```cpp
template <typename T>
struct WidgetBase
{
    WidgetBase(T) {}
};

template <typename T>
struct Widget : public WidgetBase<T>
{
    using WidgetBase<T>::WidgetBase;
};

Widget w(1);   // Works in C++20! :)
```

```cpp
template <typename T>
struct WidgetBase
{
    WidgetBase(T) {}
};

template <typename T>
struct Widget : public WidgetBase<T>
{
    using WidgetBase<T>::WidgetBase;  // Triggers inheritance of ALL deduction guides
};                                    // (implicit and explicit)

Widget w(1);   // Works in C++20! :)
```

P1021 adds for C++20:

– CTAD for aggregates
– CTAD for alias templates
– CTAD from inherited constructors





```cpp
template <typename T>
struct Widget
{
    Widget(T) {}
};

Widget w(1);                // OK since C++17
```

```cpp
template <typename T>
struct Widget
{
    Widget(T) {}
};

Widget w(1);              // OK since C++17

Widget getWidget();       // Error
void setWidget(Widget);   // Error
```

```cpp
template <typename T>
struct Widget
{
    Widget(T) {}
};

Widget w(1);                // OK since C++17

Widget getWidget();         // Error
void setWidget(Widget);     // Error

auto getWidget();           // OK since C++11
void setWidget(auto);       // OK in C++20
```

```cpp
template <typename T>
struct Widget
{
    Widget(T) {}
};

Widget w(1);                // OK since C++17

Widget getWidget();         // Error
void setWidget(Widget);     // Error

auto getWidget();           // OK since C++11
void setWidget(auto);       // OK in C++20

WConcept auto getWidget();      // OK in C++20
void setWidget(WConcept auto); // OK in C++20
```

```cpp
template <typename T>
struct Widget
{
    Widget(T) {}
};

Widget w(1);                // OK since C++17

Widget getWidget();         // Error
void setWidget(Widget);     // Error

auto getWidget();           // OK since C++11
void setWidget(auto);       // OK in C++20

WConcept auto getWidget();      // OK in C++20
void setWidget(WConcept auto); // OK in C++20
```

CTAD from incomplete template argument lists?

```cpp
auto t = std::make_tuple<int>(x, true, "Yes");  // works for FTAD

std::tuple<int> t(x, true, "Yes");  // Doesn't work for CTAD
```

CTAD from incomplete template argument lists?

```
auto t = std::make_tuple<int>(x, true, "Yes");  // works for FTAD

std::tuple<int> t(x, true, "Yes");  // Doesn't work for CTAD

std::array<int> a = {i, j, k};  // Doesn't work
std::set<int> s({i, j, k}, comp);  // Doesn't work
```

CTAD from incomplete template argument lists?

```
Widget<int> w(3.0, true);  // Is Widget<int> a placeholder?
```

CTAD from incomplete template argument lists?

```
template <typename T, typename U>
struct Widget
{
    Widget(T, U);
};

Widget<int> w(3.0, true);  // Is Widget<int> a placeholder?
```

CTAD from incomplete template argument lists?

```
template <typename... Types>
struct Widget
{
    Widget(Types...);
};

Widget<int> w(3.0, true);  // Is Widget<int> a placeholder?
```

CTAD from incomplete template argument lists?

```
template <typename... Types>
struct Widget
{
    Widget(Types...);
};

Widget<int> w(3.0, true);  // Is Widget<int> a placeholder?

// If you say "yes", you break existing code:

std::tuple<int> t(5, allocator<int>());  // std::tuple<int> in C++17
```

CTAD from incomplete template argument lists?

```
template <typename T, typename U = int>
struct Widget
{
    Widget(T, U);
};

Widget<int> w(3.0, true);  // Is Widget<int> a placeholder?
```

CTAD from incomplete template argument lists?

```
template <typename T, typename U = int>
struct Widget
{
    Widget(T, U);
};

Widget<int> w(3.0, true);  // Is Widget<int> a placeholder?

// If you say "yes", you break existing code:

std::vector<int>(MyAlloc()); // std::vector<int, std::allocator<int> in C++17
```

CTAD from incomplete template argument lists?

```
template <typename T, typename U>
struct Widget
{
    Widget(T, U);
};

Widget<int> w(3.0, true);  // imagine this would deduce Widget<int, bool>
```

CTAD from incomplete template argument lists?

```
template <typename T, typename U = int>
struct Widget
{
    Widget(T, U);
};

Widget<int> w(3.0, true);  // imagine this would deduce Widget<int, bool>
```

<— In Widget v2.0, you add this

CTAD from incomplete template argument lists?

```
template <typename T, typename U = int>
struct Widget
{
    Widget(T, U);
};

Widget<int> w(3.0, true);  // imagine this would deduce Widget<int, bool>
```

<— In Widget v2.0, you add this

Breaks your users' code!

Widget<int, int>

CTAD from incomplete template argument lists?

No.

CTAD from incomplete template argument lists?

No.

No <> = CTAD.
<> = no CTAD.

CTAD from incomplete template argument lists?

No.

No <> = CTAD.
<> = no CTAD.

:)


Thank you!
@timur_audio
blog: https://timur.audio
includecpp.org