Fixing CTAD for aggregates

Timur Doumler (papers@timur.audio)

Document #:	P2082R1
Date:	2020-02-14
Project:	Programming Language C++
Audience:	Core Working Group

Abstract

C++20 introduces CTAD for aggregate types (motivation see [P1021R5]; wording see [P1816R0]). However, the current wording still has several technical issues, some of which unintentionally break existing C++17 code. This paper proposes to fix those issues.

1 Fix breakage of existing deduction guides

Consider:

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

In C++17, the existing deduction guide will deduce `std::array<int, 2>`. However, in the current C++20 draft, an aggregate deduction candidate would be synthesised and would fail, because `std::array` has only one aggregate element (its raw array data member) but the *braced-init-list* has two initialisers. With the current wording, this would make the above program ill-formed.

We propose to fix any such unwanted interference between aggregate CTAD and existing deduction guide by omitting aggregate CTAD in case there are any explicit deduction guides defined for the class template in question. This ensures that aggregate types with explicit deduction guides like `std::array` keep working unchanged in C++20 in all cases.

2 Fix breakage of copy construction

Consider:

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

int main() {
  X<int> x1;
  X x2 {x1};
}
```

In C++17, this will deduce `X<int>` as the type of `x2`. However, in the current C++20 draft, aggregate CTAD would try to synthesise a deduction candidate, which again would fail because there are zero aggregate elements but one initialiser.

We propose to fix this by removing the aggregate deduction candidate from the overload set in case an element of the *initializer-list* does not have a corresponding aggregate element that it initialises, instead of making the program ill-formed.

3 Fix deduction for aggregates with elements of array type

3.1 Allow deduction of array size

Consider a type with a data member of array type for which we want to perform aggregate CTAD:

```
template <typename T, std::size_t N>
struct A {
  A array[N];
};
```

The expectation is that the following declarations should work:

```
A a1 = {{1, 2, 3}};  // should deduce A<int, 3>
A a2 = {"meow"};     // should deduce A<const char, 5>
```

Unfortunately, neither of these work in the current C++20 draft, because the parameter of the synthesised aggregate deduction candidate is subject to array-to-pointer decay, and therefore the synthesised candidate is useless.

We propose to fix this by specifying that, if the aggregate element being deduced is of array type, and the initialiser can be used to initialise an array (i.e. it is a *braced-init-list* or a string literal), we use a reference to array (instead of just array) for the corresponding parameter type in the aggregate deduction candidate. As a result, the parameter is no longer subject to array-to-pointer decay, and its array bound can be used to deduce the array bound of the data member.

3.2 Allow brace elision if array size is known

Consider:

```
template <typename T>
struct B {
  T array[2];
};

B b = {0, 1};
```

In the current C++20 draft, this code will not compile, because brace elision is not considered for aggregate elements of dependent type. This rule was originally introduced because in general, if the type of the aggregate element is unknown, it is unknown whether it is itself a subaggregate, and if so, how many elements it has. Therefore, the compiler cannot know how to interpret the initialiser list if brace elision is possible.

However, the “brace elision is not considered for any element of dependent type” wording feels unnecessarily restrictive. In the case of an array member, the number of subaggregate elements is always equal to the size of the array. If the size of the array is known, then the number of subaggregate elements is known, too. It is therefore possible to consider brace elision even if we haven’t yet deduced the array type, resulting in code that works the way the user would expect.

We propose to relax the above rule to say that brace elision is not considered for any element of dependent non-array type or of array type with a dependent bound.

4 Fix interaction between aggregate CTAD and pack expansion

4.1 Trailing pack expansion

Consider:

```
template<typename... T>
struct C : T... {};

C c = {
  []{ return 1; },
  []{ return 2; }
};
```

This is an aggregate with a variadic number of aggregate elements. Since the initialiser list contains two elements, the expectation is that those two elements would be used to initialise two aggregate elements, which become the two base classes of `C`. In other words, aggregate CTAD should synthesise a deduction guide that achieves the same effect as

```
template<typename... T>
C(T...) -> C<T...>;
```

However, the current wording ("Let $x_1, ..., x_n$ be the elements of the *initializer-list* [...] For each x_i, let e_i be the corresponding element of `C` that would be initialized by x_i") is under-specified for this case. It seems to suggest that either, the ill-formed deduction guide

```
template<typename... T>
C(T, T) -> C<T...>;
```

is produced instead, or that deduction fails because there is no such element of `C`. Both interpretations lead to a wrong result.

We propose to fix this by specifying that if the last element of the aggregate is a trailing pack expansion, it is assumed to correspond to all remaining elements of the initialiser list.

4.2 Non-trailing pack expansion

Consider:

```
template<typename... T>
struct C : T... {
  std::any a;
};

C c = {
  []{ return 1; },
  []{ return 2; } // does this initialise a base class, or the member a?
};
```

Similar to the previous issue, the current wording is under-specified for this case. If the aggregate elements contain a non-trailing parameter pack, it is ambiguous which initialisers should correspond to which aggregate elements, and deduction should fail.

We propose wording that specifies the synthesised aggregate deduction guide to behave like

```
template<typename... T>
C(T..., std::any) -> C<T...>;
```

which is ill-formed, because the non-trailing `T...` is deduced to an empty pack, and then the deduction guide fails to match due to an arity mismatch.

4.3 Conflicting deduction from pack expansions

Consider a case where the aggregate elements contain a parameter pack deduced from multiple places in the initialiser list:

```
template <typename... T>
struct C : std::tuple<T...>, T... {};

C c = {std::tuple<A, B, C>{}, {}, {}};
```

What should happen in this case? If the above was regular function template argument deduction, it would deduce a pack arity of 3 from `std::tuple<A, B, C>`, but a pack arity of 2 from the subsequent template arguments. The program would then be ill-formed because of conflicting deduction. The current wording for aggregate CTAD behaves in the same way.

However we argue that this is the wrong model for aggregate CTAD, because in aggregate initialisation, unlike in a function call, any number of trailing initialisers can be omitted. This should be fine, as long as those omitted initialisers are not needed for deducing the pack arity because it was already deduced by preceding initialisers.

Similarly, this initialisation should work as well:

```
C c = {std::tuple<A, B, C>{}, A{}, B{}};
```

In the model we propose, the first initialiser deduces the pack arity to 3 and the template parameters to `A, B, C`. The trailing initialisers deduce `A` and `B` for the first two template parameters, but don't attempt to deduce the pack arity again, because it was already deduced.

However, the following code should be ill-formed:

```
C c = {std::tuple<A, B, C>{}, A{}, D{}};
```

even if `D` is implicitly convertible to `B`, because now the initialisers deduce conflicting types (`B` vs. `D`) for the second template parameter, rather than just a different pack arity.

5 Proposed wording

The proposed changes are relative to the C++ working paper [N4849].

Modify [over.match.class.deduct], paragraph 1, as follows:

> In addition, if `C` is defined and its definition satisfies the conditions for an aggregate class ([dcl.init.aggr]) with the assumption that any dependent base class has no virtual functions and no virtual base classes, and the initializer is a non-empty *braced-init-list* or parenthesized *expression-list*, and there are no *deduction-guides* for `C`, the set contains an additional function template, called the *aggregate deduction candidate*, defined as follows. Let $x_1, ..., x_n$ be the elements of the *initializer-list* or *designated-initializer-list* of the *braced-init-list*, or of the *expression-list*. For each x_i, let e_i be the corresponding aggregate element of `C` or of one of its (possibly recursive) subaggregates that would be initialized by x_i ([dcl.init.aggr]) if:
>
> — brace elision is not considered for any aggregate element that has a dependent non-array type or an array type with a value-dependent bound, and
>
> — each non-trailing aggregate element that is a pack expansion is assumed to correspond to no elements of the initializer list, and
>
> — a trailing aggregate element that is a pack expansion is assumed to correspond to all remaining elements of the initializer list (if any).

If there is no such aggregate element e_i for any x_i, ~~the program is ill-formed~~the aggregate deduction candidate is not added to the set. The aggregate deduction candidate is derived as above from a hypothetical constructor `C`(`T`$_1$, ..., `T`$_n$), where:

— if e_i is of array type and x_i is a *braced-init-list* or string literal, `T`$_i$ is an rvalue reference to the declared type of e_i, and

— otherwise, `T`$_i$ is the declared type of ~~the element~~ e_i,

except that additional parameter packs of the form `P`$_j$... are inserted into the parameter list in their original aggregate element position corresponding to each non-trailing aggregate element of type `P`$_j$ that was skipped because it was a parameter pack, and the trailing sequence of parameters corresponding to a trailing aggregate element that is a pack expansion (if any) is replaced by a single parameter of the form `T`$_n$... .

Modify [over.match.class.deduct], paragraph 4, as follows:

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 guides of the template named by the placeholder 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 following exceptions apply:

— T~~t~~he 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, or is derived from, a specialization of the class template directly or indirectly named by the placeholder.

— During template argument deduction for the aggregate deduction candidate, the number of elements in a trailing parameter pack is only deduced from the number of remaining function arguments if it is not otherwise deduced.

In [over.match.class.deduct], paragraph 5, append to the example as follows:

```
template <typename... T>
struct Types {};

template <typename... T>
struct F : Types<T...>, T... {};

struct X {};
struct Y {};
struct Z {};
struct W { operator Y(); };

F f1 = {Types<X, Y, Z>{}, {}, {}};     // OK, F<X, Y, Z> deduced
F f2 = {Types<X, Y, Z>{}, X{}, Y{}};  // OK, F<X, Y, Z> deduced
F f3 = {Types<X, Y, Z>{}, X{}, W{}};  // error: conflicting types deduced; operator Y not
considered
```

Document history

— **R0**, 2020-01-13: Initial version.

— **R1**, 2020-02-14: Added more fixes.

Acknowledgements

Many thanks to Jason Merrill and Richard Smith for pointing out the issues, Michael Spertus for helping me understand them properly, and Richard Smith for providing useful code examples and the draft wording.

References

[N4849] Richard Smith. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4849.pdf, 2020-01-14.

[P1021R5] Mike Spertus, Timur Doumler, and Richard Smith. Filling holes in Class Template Argument Deduction. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1021r5.html, 2019-08-15.

[P1816R0] Timur Doumler. Wording for class template argument deduction for aggregates. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1816r0.pdf, 2019-07-70.