Prohibit aggregate types with user-declared constructors

Timur Doumler (papers@timur.audio)
Arthur O'Dwyer (arthur.j.odwyer@gmail.com)
Richard Smith (richardsmith@google.com)
Howard E. Hinnant (howard.hinnant@gmail.com)

Document #:	P1008R0
Date:	2018-05-07
Project:	Programming Language C++
Audience:	Evolution Working Group, Core Working Group

Abstract

C++ currently allows some types with user-declared constructors to be initialized via aggregate initialization, bypassing those constructors. The result is code that is surprising, confusing, and buggy. This paper proposes a fix that makes initialization semantics in C++ safer, more uniform, and easier to teach. We implemented the proposed change in Clang and ran it on existing codebases. We also discuss possible alternatives and related work.

1 Motivation

Today, the rules of initialization in C++ are frequently cited as one of the most confusing and hard-to-learn aspects of the language. A particular problem occurs with aggregate types with user-declared constructors. The current rule says that an aggregate class type cannot have user-provided, inherited, or `explicit` constructors, but explicitly deleted or defaulted constructors are allowed. This rule has many surprising and unfortunate consequences.

1.1 Unintended instantiation

Consider:

```
struct X {
  X() = delete;
};

int main() {
  X x1;     // ill-formed - default c'tor is deleted
  X x2{};   // compiles!
}
```

Clearly, the intent of the deleted constructor is to prevent the user from initializing the class. However, contrary to intuition, this does not work: the user can still initialize `X` via aggregate initialization because this completely bypasses the constructors. The author could even explicitly delete all of default, copy, and move constructor, and still fail to prevent the client code from instantiating `X` via aggregate initialization as above. Most C++ developers are surprised by the current behaviour when shown this code.

The author of class `X` could alternatively consider making the default constructor `private`. But if this constructor is given a defaulted definition, this again does not prevent aggregate initialization (and thus, instantiation) of the class:

```
struct X {
private:
  X() = default;
};

int main() {
  X x1;     // ill-formed - default c'tor is private
  X x2{};   // compiles!
}
```

Because of the current rules, aggregate initialization allows us to "default-construct" a class even if it is not, in fact, default-constructible:

```
static_assert(!std::is_default_constructible_v<X>);
```

would pass for both definitions of `X` above.

The programmer has to know way too many details about both the class and about arcane initialization rules to be able to understand this behaviour. `X x1;` doesn't compile because the user-declared constructor inhibits the compiler from supplying a default constructor. However, `X x2{};` compiles because despite the rule that says the compiler doesn't supply a default constructor, the class has no user-provided, `explicit`, or inherited constructors, no private or protected non-static data members, no virtual functions, and no virtual, private, or protected base classes, and so the compiler allows the aggregate initialization syntax. The majority of C++ developers are unlikely to ever master those rules. As a result, we end up with code that gets used in unexpected ways, even after passing good unit tests such as the `static_assert` above. This leads to bugs and unintended behaviour.

1.2 Unintended member initialization

Aggregate initialization creates even more surprises if the type in question has non-static data members. Consider:

```
struct X {
  int i{4};
  X() = default;
};

int main() {
  X x1(3);  // ill-formed - no matching c'tor
  X x2{3};  // compiles!
}
```

Here, `X` has a default constructor, and the member `int i` is given a default member initializer to ensure that it will always hold the value `4` after default-construction. But surprisingly, aggregate initialization allows the user to bypass this interface and initialize `X` with a different value for `i`, potentially breaking class invariants. This can easily cause bugs and unintended behaviour.

It gets worse. We could try to make the intent even more clear by explicitly deleting the constructor that takes an `int`. But, surprisingly, this still doesn't make the class non-aggregate and therefore accomplishes nothing:

```
struct X {
  int i{4};
  X(int) = delete;
};
```

```
int main() {
  X x1(3);  // ill-formed - using deleted c'tor
  X x2{3};  // compiles!
}
```

This behaviour is confusing, not useful, and potentially dangerous.

1.3 C++17 workaround: `explicit`

The need to control instantiation of such simple types often arises in practice. One example are tag types like `std::piecewise_construct_t`, which the user should not be able to instantiate. How can we accomplish this with the current rules? C++17 introduced the following workaround. If we add `explicit` to the deleted default constructor, this causes the type to be non-aggregate:

```
struct X {
  X() = delete;
};

struct Y {
  explicit Y() = delete;
};

int main() {
  X x{};  // compiles
  Y y{};  // ill-formed because of explicit!
}
```

Unfortunately, this particular use of the `explicit` keyword is yet another rule that is not well-known and can cause a lot of confusion. It is also a workaround that fails to address the core of the problem.

1.4 Out-of-line default

There are more bear traps that emerge from the current rules. Because a type with a *user-declared* defaulted constructor can be an aggregate, but a type with a *user-provided* defaulted constructor cannot, it depends on the position of the `= default` whether aggregate initialization is ill-formed or not:

```
struct X {
   int i;
   X() = default;
};

struct Y {
  int i;
  Y();
};
Y::Y() = default;

int main() {
  X x{4};  // compiles
  Y y{4};  // ill-formed - not an aggregate!
}
```

2 Proposed wording

The solution to all of the above is to make classes with explicitly deleted or defaulted constructors non-aggregate. This can be accomplished via the following minimal wording change.

Modify **[dcl.init.aggr]** as follows:

> An *aggregate* is an array or a class (Clause 12) with
>
> — no ~~user-provided, `explicit`,~~user-declared or inherited constructors (15.1),
>
> — no private or protected non-static data members (Clause 14),
>
> — no virtual functions (13.3), and
>
> — no virtual, private, or protected base classes (13.1).

The proposed change is relative to the C++ working paper [Smith2018].

3 Impact on the standard

3.1 Simpler and more uniform model for initialization semantics

The proposed change would make aggregate initialization ill-formed for any class that has an explicitly defaulted or deleted constructor. This removes all cases in which aggregate initialization could bypass the declared initialization interface of a class. Not only does this make all the code from section 1 behave according to most people's expectations, but it also simplifies the rules, making initialization semantics in C++ more uniform and easier to teach.

The underlying mental model is as follows. A class, by default, gets a set of constructors: a default constructor, and aggregate "constructors" that initialize the fields. If the user declares any constructors themselves, they are taking ownership of the initialization semantics for the class, and the compiler no longer provides these defaults.

This mental model matches most users' expectations. The proposed change would align C++ initialization semantics with this model and remove the currently existing harmful exceptions. We won't have to teach unintuitive rules like "a `=deleted` or not-out-of-line-`=defaulted` constructor, if not marked `explicit`, does not suppress aggregate initialization, but a constructor without these properties does." Instead, we will be able to teach a simple and intuitive rule: "if a class declares any constructors, it will always be initialized through one of those constructors".

Copy and move semantics are notionally distinct from initialization semantics as described above, and we do not propose any changes for the generation of default copy and move constructors.

3.2 Eliminating "accidental aggregates"

The proposed change addresses a particular family of bugs. It often happens that the author of a library type intends to control its instantiation by declaring a constructor, but is not aware of the fact that this does not actually prevent aggregate initialization. This, in turn, allows clients of this type to use it in unintended ways and invoke unintended behaviour. This proposal eliminates such "accidental aggregates".

We found that there are many more occurrences of "accidental aggregates" that our proposal would fix than there are cases of code that our proposal would break (3.3 and 3.4). Aggregate types are sometimes useful and have their place in C++. However we believe that they are useful much less often than they are an unwanted mechanism that provides "constructors" that the programmer did not intend.

3.3 Preventing uninitialized data members

There is a certain hypothetical constellation that would become ill-formed under this proposal.

Consider an aggregate type with data members of built-in type. An expert programmer who is fully aware of the current rules might choose to delete the constructor of such a type to force the data

members to always be initialized, thus preventing undefined behaviour. The only way to initialize the object would then be aggregate initialization, which ensures that data members will always be initialized:

```
struct X {
  int i;
  X() = delete;
};

int main() {
  X x1;      // ill-formed
  X x2{};    // zero-initializes i. OK in C++17, ill-formed with this proposal
  X x3{4};   // initializes i to 4. OK in C++17, ill-formed with this proposal
}
```

This proposal would make both `x2` and `x3` ill-formed, because `X` would no longer be an aggregate type. To achieve guaranteed initialization of members, the user would have to remove the deleted constructor and add default member initializers instead. This would make the class aggregate again:

```
struct X {
  int i{0};
};

int main() {
  X x1;      // zero-initializes i. OK in C++17 and with this proposal
  X x2{} ;   // zero-initializes i. OK in C++17 and with this proposal
  X x3{4};   // initializes i to 4. OK in C++17 and with this proposal
}
```

Note that with this definition of `X`, both `x1` and `x2` are valid and do the same thing. Arguably, the resulting code is cleaner and easier to understand. It would also make `X` conform to the often recommended "rule of zero".

3.4 Redundant default/delete

There is a case of arguably valid code that would break under this proposal.

Consider a type that *relies on being an aggregate* and has an implicitly deleted default constructor (for example, because of a reference member). The programmer may choose to explicitly declare this property for "documentation purposes", although this is redundant. Consider:

```
struct Y {
  int& i;
  Y() = delete;  // redundant declaration; default c'tor would be implicitly deleted
};
```

The same applies to a compiler-provided default constructor, which the programmer may choose to explicitly declare as defaulted:

```
struct Z {
  int i;
  Z() = default;  // redundant declaration; trivial default c'tor is compiler-generated
};
```

With the changes proposed here, both `Y` and `Z` would cease to be aggregates, because they have a user-declared constructor:

```
int main() {
  int val = 4;
  Y y{val};  // OK in C++17, ill-formed with this proposal
  Z z{val};  // OK in C++17, ill-formed with this proposal
}
```

In both cases, it is possible to get back the old behaviour by simply removing the redundant constructor declaration:

```
struct Y {
  int& i;
};

struct Z {
  int i;
};

int main() {
  int val = 4;
  Y y{val};  // OK in C++17 and with this proposal
  Z z{val};  // OK in C++17 and with this proposal
}
```

Again, `Y` and `Z` now also have a simpler interface and conform to the "rule of zero".

It is important to note that, conceptually, an implicitly deleted constructor is actually not the same as an explicitly deleted one. By using the latter, the programmer *takes ownership* of the type's initialization semantics. Consider:

```
struct Foo; // not default-constructible

struct Bar {
  Foo foo;
};
```

Here, the compiler generates an implicitly deleted constructor for `Bar` because `Foo` is not default-constructible. If `Foo` is later changed to be default-constructible, the compiler will make `Bar` default-constructible as well — the initialization semantics are *implicit*. On the other hand, if you declare `Bar() = delete;`, then changing `Foo` to be default-constructible will *not* make `Bar` default-constructible as well — the initialization semantics are *user-defined*. Due to this conceptual difference we believe that adding redundant constructor declarations for "documentation purposes" is not a very strong usecase.

3.5 No change in behaviour of well-formed code

The change proposed here makes some constructions ill-formed, as described above. However, adopting this proposal will not introduce any changes in behaviour to well-formed C++17 code that would remain well-formed. In particular, adopting this proposal can never silently turn aggregate-initialization into a constructor call.

There is one case where well-formed C++17 code would remain well-formed and would change its *notional* meaning, but without any change in *behaviour*.

Consider a class with data members and a `=default`ed default constructor that would be an aggregate in C++17, but stop being an aggregate with this proposal. Now, consider initializing the class with empty `{}` braces:

```
struct X {
  int i;
  X() = default;
}

int main() {
  X x{}; // aggregate initialization in C++17; value-initialization with this proposal
}
```

With this proposal, `X` would no longer be aggregate. Therefore, `X x{};` would no longer aggregate-initialize `x`. For a non-aggregate class with a `=defaulted` default constructor, `X x{};` performs *value-initialization*, which zero-initializes member `x.i` (the same as what aggregate initialization with empty braces would do). We do not propose any changes to this rule. Therefore, the observable behaviour of `X x{};` would be exactly the same as before.

Omitting the braces and writing `X x;` would *default-initialize* `x`, leaving member `x.i` uninitialized, both in C++17 and with this proposal, so there would be no change here either.

4 Implementation and testing on real-world codebases

We have implemented a Clang patch[1] that implements the proposed change. Additionally, it can flag with a warning any type that is currently an aggregate but would cease to be an aggregate with the proposed change.

4.1 LLVM/Clang

We have run the patch on the LLVM/Clang codebase. The proposed change *broke no code*. We found no occurrences of the use cases 3.3 or 3.4.

Six types were flagged as no longer aggregate. Nothing in the code depended on them being aggregate. All six cases were different flavours of "accidental aggregates". Two had a `=deleted` constructor and four had a `=defaulted` one. Among those was one particularly insidious case. Consider:

```
struct X {
   X() {}
   // some data members...
};
```

This class was initially not an aggregate, because it has a user-provided constructor. Later, someone decided to "modernize" the code base by changing `X() {}` to `X() = default;`. Unbeknownst to them, this silently turned the class into an aggregate, changing its initialization semantics.

4.2 Other codebases

We have not yet run the patch on other large codebases at this time. We might do such a study in a future revision of this paper.

5 Alternative approach

We believe that this proposal is the clearest, most easily teachable, and least disruptive way to fix the issues with initialization semantics described in section 1. However, one alternative approach was suggested, which we discuss here.

The idea of this alternative is to leave the definition of aggregate types unchanged, but instead change the selection rule for braced initializers. Instead of performing list-initialization if possible, and otherwise call constructors (status quo), they could try a constructor call first, and only perform list-initialization if no matching constructor can be found[2].

We do not think this is a viable route for several reasons.

[1] https://reviews.llvm.org/D44948

[2] Note that such a rule would require an exception that `X x{};` would not match a `=defaulted` constructor. Otherwise this could introduce undefined behaviour in existing code by silently turning aggregate initialization into a constructor call, i.e. silently turning zero-initialization of data members into default-initialization. This cannot happen with our proposal (see 3.5).

First, while this would make aggregate initialization ill-formed if the arguments inside `{...}` match an explicitly deleted constructor[3], it would not fix the other issues that our proposal fixes. Notably, it would fix neither the first example in 1.2 nor any other "accidental aggregates" with `=default`ed constructors. Second, instead of making C++ simpler and more uniform, this would make the rules more complex. Third, whenever this alternative rule of "constructors first" would apply, the types would still be aggregates. This would have unwanted and surprising side effects. Currently, aggregate initialization is either available (if the type is an aggregate), or it is not, and our proposal does not change that. With this alternative approach however, aggregate initialization would become available for some combinations of an incomplete initializer list, but not others:

```
struct X {
  int a, b, c;
  X() = delete;
}

struct Y {
  int a, b, c;
  Y(int) = delete;
}

int main() {
  X x{};         // would be ill-formed - matches deleted c'tor
  X x{1};        // OK
  X x{1, 2};     // OK
  X x{1, 2, 3}; // OK

  Y y{};         // OK
  Y y{1};        // would be ill-formed - matches deleted c'tor
  Y y{1, 2};     // OK
  Y y{1, 2, 3}; // OK
}
```

There are more surprising interactions between constructors and aggregate initialization that such an alternative rule would create. Consider:

```
struct A {
  int x, y, z;
  A(int) = delete;
};

A a1 = {0};        // would be ill-formed - matches deleted c'tor
A a2 = {1, 2, 3}; // OK
```

Later, the programmer implements the constructor `A(int)`, changing the class to:

```
struct A {
  int x, y, z;
  A(int i) : x(n), y(n), z(n) {}
};
```

This would now have an unexpected side effect: changing the deletedness of `A(int)` would result in the unrelated initialization `A a2 = {1, 2, 3};` becoming ill-formed:

```
A a1 = {0};        // OK
A a2 = {1, 2, 3}; // would become ill-formed!
```

We consider such unexpected interaction between constructors and aggregate initialization highly undesirable. The goal of the present proposal is to eliminate such issues, instead of creating new ones.

[3]Note that aggregate initialization is already ill-formed in all cases where the arguments match an *implicitly* deleted constructor.

6 P0960 and other directions

[P0960], mostly orthogonal to the present paper, proposes to allow aggregate initialization with the parenthesized syntax in addition to the braced syntax. Consider an aggregate type:

```
struct A {
  int i;
};

int main() {
  A a1{4};  // OK
  A a2(4);  // ill-formed; P0960 would allow this
}
```

Additionally, P0960 mentions the possibility of allowing the `(...)` syntax to initialize arrays.

As of revision R0 (reviewed in Jacksonville 2018), P0960 does not consider the issues that we propose to fix here. We believe that this is necessary, because otherwise with P0960 these issues could show up with the `(...)` syntax in addition to `{...}`, potentially creating even more surprising interactions between the different kinds of initialization semantics (constructors vs. aggregate initialization).

Apart from the need to address this, we would like to emphasize that the present paper does not preclude the direction of P0960 in any way. It is actually very interesting to consider what would happen if both the present paper and P0960 were adopted. The result would make braced and parenthesized initializers do the exact same thing in almost all cases. The only two exceptions would be list-initialization in the presence of constructors taking a `std::initializer_list` (which could be selected with braces but not parentheses) and narrowing conversions (which could be selected with parentheses but not braces).

More broadly, there seem to be three theoretically possible, mutually exclusive directions:

1. `(...)` should call constructors, `{...}` should be uniform initialization (status quo). Fix the current issues with this approach as far as possible, and otherwise leave it as is.
2. Both `(...)` and `{...}` should be uniform initialization (with a few differences such as `std::initializer_list` and narrowing conversions). This is the direction of P0960.
3. `(...)` should call constructors, `{...}` should perform list-initialization. Abandon the idea of uniform initialization. Stop teaching that `{...}` should be used to call constructors and even deprecate and remove this functionality.

We would like to emphasize that the present paper does not preclude any of these three directions. We do not propose any changes on what syntax can perform what kinds of initialization. We merely propose to fix issues with aggregate initialization that exist today, and if not fixed will keep existing regardless of the direction taken.

Acknowledgements

Many thanks to Nico Josuttis, Igor Akhmetov, Dmitry Kozhevnikov, Ville Voutilainen, Nicole Mazzuca, Peter Dimov, Ben Craig, Kévin Alexandre Boissonneault, Michał Dominiak, Mark Zeren, Kate Gregory, Barry Revzin, Agustín Bergé, Tony Van Eerd, Jens Maurer, Gabriel Dos Reis, and Daveed Vandevoorde for their helpful comments and code examples.

References

[P0960] Ville Voutilainen. Allow initializing aggregates from a parenthesized list of values. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0960r0.html, 2018 (accessed 2018-05-07).

[Smith2018] Richard Smith. Working Draft, Standard for Programming Language C++. https://github.com/cplusplus/draft, 2018 (accessed 2018-05-07).