Project:  ISO JTC1/SC22/WG21: Programming Language C++
Document: P0654R0
Date: 2017/06/05
Reply-to: Loïc Joly (loic.actarus.joly@numericable.fr)
Audience: Evolution

# Explicit struct

Enforcing initialization of member data

## Abstract

This document explores different ways in which the designer of an aggregate type could make sure that all members of the aggregate are initialized by the aggregate user, without manually defining a constructor for it. It then proposes a new syntax to make that easier.

## The problem

Very often, I come up with types that just aggregate some values, without direct relationship between those values. They will be used together, but they have no behaviour, no invariant to enforce. Let's take an example of a drawing software with a special effect algorithm. This algorithm has two parameters: A colour, and an effect radius (in real cases, there would be more than two parameters, which justifies packing them). We can define this in different ways:

```
struct EffectParameters
{
    Colour colour;
    int radius;
};
```
The problem with this writing is that it is very easy for the user of this code to create an `EffectParameters` without initializing one of the member. This is especially true if the code evolves and the algorithm now takes a third parameter (a direction, for instance). How can we make sure this new value is defined everywhere?

Another option is to write the following:

```
struct EffectParameters
{
    Colour colour = red;
    int radius = 5;
};
```

This is better in that we no longer have to deal with undefined behaviour when reading the value of `radius`, but the value may not be the one expected.

Finally, what I consider the best solution in today's C++:

```
struct EffectParameters
{
    EffectParameters(Colour colour, int radius): colour(colour), radius(radius) {}
    Colour colour;
    int radius;
};
```

Now, the user of this struct has to provide values for each member. The main problem with this solution is that the maintainer of this class has to write tedious code to get there. The type of each member has to be repeated in the constructor arguments, and the name of each member is repeated thrice.

The goal of this proposal is to allow for a new, lightweight syntax.

## Proposed solution : Explicit struct

The proposed syntax is the following:

```
explicit struct EffectParameters
{
    Colour colour;
    int radius;
};
```

With this definition, the following code would be:

```
EffectParameters params1{ red }; // Error, diagnostic required
EffectParameters params2{ red, 12 }; // Correct
```

This applies to all kinds of ways of creating an `EffectParameters` instance. For instance, you can no longer use the one argument `resize` of a `vector<EffectParameters>` and if you inherit from `EffectParameters`, you will have to make sure those members are correctly taken care of in the derived class.

Static data members are not concerned with this rule.

If a member data has an initializer, it is not concerned by this rule:

```
explicit struct EffectParameters
{
    int radius;
    Colour colour = red;
};

EffectParameters params1{ 12 }; // Correct, the colour will be red
```

But to take advantage of that, the default member must be at the end:

```
explicit struct EffectParameters
{
    int radius;
    Colour colour = red; // Explicitly initialized member
    double height; // Followed by a non-explicitly initialized member
};

EffectParameters params1{ 12 }; // Error, height not specified
EffectParameters params2{ 12, 15 }; // Error, 15 is not a color, and height not specified
EffectParameters params3{ 12, red, 15 }; // Ok
```

If an explicit struct also defines a constructor, then the constructor has to initialize all members:

```
explicit struct EffectParameters
{
    EffectParameters(Colour colour, int radius):
        colour(colour), radius(radius) {} // Error, direction is not initialized
    Colour colour;
    int radius;
    float direction;
};
```

## Discussion

### Do we really need it?

We don't *need* it (we can always write the constructor manually), but it is a small addition, and I've seen this situation many many times, so I think it is worth adding it to the language.

Moreover, I've seen many people use the first variant to write such structs (no constructor, no default value), just because of the tediousness of writing the constructor. This tendency seems to be increasing since C++11 with the better ways we have to initialize aggregates. So the proposed change would probably lead to more robust code.

### Could it be done with an attribute?

Probably, although I'm not sure it respects the guidelines about attributes. However, I think that the `explicit` keyword clearly captures the intent of the code, and I do not see this usage clashing with other uses of this keyword.

### What about constructors in explicit structs?

There seems to be 3 possibilities:

a. Explicit structs can't have constructors
b. Explicit structs can have constructor, the constructor does not have to initialize all members
c. Explicit structs can have constructors, they must initialize all members (this is the option currently proposed)

*b* seems illogical: Since the struct has a constructor, it's not an aggregate, and if explicit only has an impact on aggregate initialization, it serves no purpose here. Moreover, allowing uninitialized members in a struct marked as explicit seems to defeat the purpose of the keyword.

*a* or *c* are both reasonable. I think that c is consistent with the usage when there is no constructor (the user of the type can be sure all members will be initialized), and might provide a small help to the class designer when it contains many constructors.

I'm ready to revert this decision and go for *a* if it can increase the consensus for the main part of this feature.

### Should this feature be worded in terms of an automatically generated constructor, or in more constraints on the aggregate initialization?

I have no idea, and I'm open to advice by people more experimented in wording than I am.

### Should we require explicit initialization for a member with an initializer?

If we did, the initializer would never be used. So the answer is no.

### Could we decide what needs to be initialized on a member-by-member basis?

I’ve had some requests to force some members to be initialized, but not all. Although it may have some value, I’ve not met such cases quite as often as the case where we want to enforce initialization of all members. So, in order to keep this proposal small and focused, this is not proposed.

In particular, note that members with an initializer already provide a way to exclude some members from the forced initialization. I think this will cover most cases.

However, should there be a need for that in the future, I think this proposal would not be a hindrance for finer-grained control, maybe something like:

```
struct EffectParameters
{
    explicit int radius;
    float direction;
    Colour colour = red;
};
```

### Is a type explicitness inherited from its base classes?

No. But if your base class is explicit, you have two options:

- Your derived class is an aggregate. In that case, the user has to specify initializers for the base class sub-object when aggregate-initializing the derived class
- Otherwise, in your constructor initializer, you need to initialize all the base class members

### Can a template specialization have a different explicitness from the base template?

Yes. This looks consistent with the fact that the specialization can redefine about everything from the base template.

### Has it been implemented?

No. I don’t foresee any issue.

## Thanks

I’d like to thank the people that reviewed this draft, and helped improve its quality: Gabriel Dos Reis, Jean-Marc Bourguet, Joël Falcou, Joël Lamotte, Pascal Menuet and Sylvain Bonnal Conduzorgues.