Trailing Commas in Base-clauses and Ctor-initializers

Document number: **P0562R2**
Date: 2024-04-15
Audience: Evolution Working Group, Core Working Group
Reply to: **Alan Talbot**
cpp@alantalbot.com

Abstract and Tony Table

One of the requirements of writing solid C++ is the maintenance of member initializer lists in constructors. These lists must match the declarations in the class body both in content and order. Neither is enforced by the compiler, and yet the hazards of omitting an entry or getting them out of order are serious and well known.[1] Initializers in the class body eliminate the problem in some situations, but not in the most common case where the initializers are dependent on constructor arguments.

Unfortunately, maintaining these lists is made more difficult by a small irregularity in the language. Unlike all the other initialization productions, member initializer lists do not allow a terminating comma. This proposal adds that (redundant) trailing comma.

The same issue exists for base class lists, so for similar reasons and for consistency, it also adds a trailing comma to these lists.

These small changes may not seem very exciting, and they don't change the functional capabilities of the language. But I believe they will save the millions of programmers who use C++ a noticeable amount of time and energy, and even more importantly help prevent a very insidious source of bugs (see Motivation).

C++23	Proposed
`foo::foo(int x, int y, int z) :` `a(x),` `b(y),` `c(z)` `{...}`	`foo::foo(int x, int y, int z) :` `a(x),` `b(y),` `c(z),` `{...}`
`class bar :` `public base,` `public mixin` `{};`	`class bar :` `public base,` `public mixin,` `{};`

History

R0

The first version of this paper was reviewed by EWG in Kona in 2017. It received a mixed response (6|9|7|6|2), and we agreed that there was not sufficient support to continue with it at that time.

R1

So why bring it back now? I believe there are two good reasons:

- In C++20 we have added yet another feature which wisely recognizes that allowing trailing commas makes code maintenance easier, namely designated initializers.
- We have expressed a renewed focus on the 5-7 million C++ programmers who are not language experts, and who are trying to understand and use a very complicated language.

This version is largely rewritten with greatly expanded motivation and better wording.

R2

Added trailing commas to base class lists based on strong support from EWG. Removed redundant Annex A wording. Changed the paper name per CWG request.

Motivation

Most initialization contexts accept commas as terminators (rather than the more restrictive delimiters). This convenience is welcome and valuable in my opinion, and I strongly doubt that anyone would wish it gone. Here is a review of the contexts where this question arises:

Enums

Enums have always allowed each entry to be terminated by a comma. The importance of maintaining the correct order of enums depends on whether the numbers are meaningful and/or persistent.

Array Initializers

Array initializers have always allowed each entry to be terminated by a comma (and examples of this can be found in the Standard). The importance of maintaining the correct order of array initializers is usually high.

Initializer Lists

Initializer lists have always allowed each entry to be terminated by a comma. The importance of maintaining the correct order in an initializer list is usually high.

Designated Initializers

Designated initializers also allow each entry to be terminated by a comma. It is necessary to maintain the correct order of designated initializers (the language requires them to match the declaration order) but getting it wrong cannot cause a bug because it won’t compile.

Member Initializer Lists

Member initializer lists do not allow a final terminating comma. This makes formatting them for maximum readability and maintainability something of a quandary. I have tried several different formats and have discovered no perfect answer. The best format I have found (except for very trivial cases) is the one shown above.

The problem is the last line. Every time an order change involves the last line, a comma must be added and another one deleted. This may not seem like much work, but it adds up over time and it's easy to forget, which means a compile-time error that wastes even more time.

However, the real concern is that because it's a bit fussy and annoying to rearrange the list, *people won't do it.* Getting the initializer order wrong does not cause a compile-time error, but easily could cause a quiet and subtle bug, which makes this maintenance extremely important. (The bugs caused by the order problems mentioned above are usually not quiet or subtle.)

Base Class Lists

The list of base classes in a class definition does not allow a final terminating comma. While this is not an initialization context, it poses a similar maintenance task for those who use base classes liberally. For this reason, and for consistency, this paper also proposes allowing commas at the end of base class lists.

Function and Template Parameters

Function and template parameters do not allow a final terminating comma, but these do not trouble me in practice. (Arguably any function or template with enough parameters to be much of a maintenance issue seems like it is ripe for refactoring.) This paper does not propose changing function or template parameters.

Notes

1. For example (and please note the date):
   Scott Meyers. *Effective C++,* p. 41-42. Addison-Wesley Publishing Company, 1992.

Acknowledgements

Thanks to Daveed Vandevoorde for confirming that the syntax is possible and for suggesting that base class lists are also a case worth addressing.

Proposed Wording

11.7.1 General [class.derived.general]

1 A list of base classes can be specified in a class definition using the notation:

> *base-clause:*
>
> > *: base-specifier-list ,$_{opt}$*
>
> *base-specifier-list:*
>
> > *base-specifier . . .* $_{opt}$
> >
> > *base-specifier-list , base-specifier . . .* $_{opt}$
>
> *[…]*

11.9.3 Initializing bases and members [class.base.init]

1 In the definition of a constructor for a class, initializers for direct and virtual base subobjects and non-static data members can be specified by a *ctor-initializer*, which has the form

> *ctor-initializer:*
>
> > *: mem-initializer-list ,$_{opt}$*
>
> *mem-initializer-list:*
>
> > *mem-initializer . . .* $_{opt}$
> >
> > *mem-initializer-list , mem-initializer . . .* $_{opt}$
>
> *[…]*