Document Number: p1701r2
Date: 2021-05-04
To: SC22/WG21 EWG
Reply to: Nathan Sidwell nathan@acm.org

Inline Namespaces: Fragility Bites

Nathan Sidwell

Inline namespaces were added with the goal of allowing vendors to provide different source-compatible and link-interoperable library variants. Unfortunately there was at least one defect with the design, and that has opened the door to a conflicting unexpected use.

Since presenting R0 at the Cologne 2019 meeting, another issue has come to light. Given the guidance from EWG an alternative solution is presented. This R2 update follows an EWG meeting and presents options in Tony-Table form along with suggested wording.

1 Background

Inline namespaces introduce a named scope that is almost invisible. Users do not need to name the scope in order to access members within. Qualified and unqualified namespace-scope name lookup is modified to also search inline namespace nests, adding any found entities to the lookup set.

The intent is to be able to write:

```
namespace std {
#ifdef _SMALL_STRING_OPTIMIZATION
inline namespace __sso
#endif

template <typename T> string
 { /* details unimportant.  */ }

#ifdef _SMALL_STRING_OPTIMIZATION
}
#endif
}
```

The user of the vendor's library can name '`string`' with '`std::string`'. The vendor can provide different flavours of '`string`' depending on `_SMALL_STRING_OPTIMIZATION`. Howard Hinnant noted:

> Despite the weaknesses, I can report the transition period went remarkably smoothly. libstdc++'s COW string never got confused at run-time with libc++'s SSO string.

An unqualified declaration does not redeclare a declaration visible in an inline namespace nest, however a qualified name does:

```
inline namespace A {
  void foo () {} // #1
  void bar () {} // #2
}

void foo () {} // OK, not redefinition of #1
void ::bar () {} // ERROR, redefinition of #2
```

However, template specializations do locate their general template within an inline namespace:

```
inline namespace A {
  template <int I> void foo () {} // #1
}

template<> void foo<1> () {} // OK, specializes #1
```

1.1 DR2061

Core DR2061[1] concerns a problem introduced by resolving DR1795:[2]

> After the resolution of issue 1795, 10.3.1 [namespace.def] paragraph 3 [...] appears to break code like the following:
>
> ```
> namespace A {
>    inline namespace b {
>      namespace C {
>        template<typename T> void f();
>      }
>    }
> }
>
> namespace A {
>    namespace C {
>      template<> void f<int>() { }
>    }
> }
> ```
>
> because (by definition of “declarative region”) `C` cannot be used as an unqualified name to refer to `A::b::C` within `A` if its declarative region is `A::b`.

1 https://wg21.link/cwg2061
2 https://wg21.link/cwg1795

Proposed resolution (September, 2015):

Change 10.3.1 [namespace.def] paragraph 3 as follows:

> In a *named-namespace-definition*, the *identifier* is the name of the namespace. If the *identifier*, when looked up (6.4.1 [basic.lookup.unqual]), refers to a *namespace-name* (but not a *namespace-alias*) **that was** introduced in the ~~declarative region~~ **namespace** in which the *named-namespace-definition* appears **or that was introduced in a member of the inline namespace set of that namespace**, the *namespace-definition* extends the previously-declared namespace. Otherwise, the *identifier* is introduced as a *namespace-name* into the declarative region in which the *named-namespace-definition* appears.

I.e when opening a namespace N, look for Ns indirectly reachable via nested inline namespaces. It is only if there are no such Ns that we create a new namespace.

This behaviour is different to other unqualified declarations, as described in Section 1, where no such inline namespace search occurs.

2 Bug Reports and Surprises

I implemented DR2061 in GCC 8. It caused several bug reports to be raised. During implementation I encountered a difficulty with unnamed namespaces, which as discussed below, is another issue with the DR's resolution.

2.1 PR90291

Bug report PR90291[3] was raised. The bug reporter relates that their software's organization has the following hierarchy:

```
inline namespace A {
  namespace detail  {   // #1
    void foo() {} // #3
  }
}

namespace detail  {   // #2
  inline namespace C {
    void bar() {} // #4
  }
}
```

The intent is to have functions `A::detail::foo` (#3) and `detail::C::bar` (#4). However, with DR2061 implemented, the namespace declaration #2 no longer creates a new top-level

3 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90291

namespace, but locates the previously opened A::detail at #1. Thus the second function's fully qualified name is A::detail::C::bar.

As the reporter expands in comments 6 & 7, A is a utility component name:

```
// header file
namespace component {
  inline namespace utility {
    namespace detail {
      // stuff
    }
  }
}

// source file
#include "header file"
namespace component {
  namespace detail {
    // oops, component::utility::detail
  }
}
```

If two different headers use the same hierarchy, but with different 'utility' names, a user that includes both will discover that detail has become a poisoned namespace, as any attempt to open it will result in an ambiguous lookup.

This problem was discussed on the core mailing list.[4] Gaby dos Reis commented that while DR2061 is addressing the issue it intends to address:

> However, this is already extremely fragile: if the namespace is also opened
> before including the header [example] ... then this doesn't work: #2 reopens #3 instead of
> #1.
>
> However, inline namespaces have *also* been adopted for another behavior
> entirely unrelated to versioning: as a way of providing an optional
> namespace name component (eg, std::inline literals::inline chrono_literals,
> or the example in that GCC bug report). In that guise, it is not reasonable
> to look through the inline namespace set when considering reopening a
> namespace.

Davis Herring suggested:

> ... any namespace declaration that would cause a subsequent (fully-qualified) namespace
> lookup to be ambiguous due to inline namespaces should be rejected immediately.

4 http://lists.isocpp.org/core/2019/04/6102.php

That is, not accepting DR2061, but making namespace definition #2 in the bug report example above ill-formed due to it (also) matching definition #1.

GCC 8 was released in May 2018, PR90291 was filed in April 2019. I note the following related PRs, both fallout from implementing DR2061

- 87155,[5] anonymous namespaces inside inline namespaces (see Section 2.2)
- 81064,[6] libstdc++ breakage, because it had exactly this structure. The library was changed.

Given those issues, and Richard Smith's comment that:

> Clang intends to implement DR2061, but it looks like we get it wrong in some ways …

perhaps DR2061's direction is suboptimal?

2.2 Unnamed Namespaces

The standard specifies:

> An *unnamed-namespace-definition* behaves as if it were replaced by
>
> ```
> inline_opt namespace unique { /* empty body */ }
> using namespace unique ;
> namespace unique { namespace-body }
> ```
>
> … all occurrences of **unique** in a translation unit are replaced by the same identifier, and this identifier differs from all other identifiers in the translation unit
>
> [namespace.unnamed]

This wording means that placing an unnamed namespace inside an inline namespace could cause issues with other unnamed namespaces within the same inline namespace nest:

```
namespace {}
inline namespace bob {
  namespace {}
}

namespace {} // error, ambiguous
```

In addressing PR87155 (& PR89068) I accepted the above by not searching an inline namespace nest when opening an unnamed namespace. Again, this was discussed on the core mailing list.[7] That discussion concluded this was well-formed, but it predates the above-mentioned DR2061 discussion, and I now consider the argument incomplete.

5 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=87155
6 https://gcc.gnu.org/bugzilla/show_bug.cgi?id=81064
7 http://lists.isocpp.org/core/2018/08/4912.php

2.3 Modules

Consider:

```
export module Impl;
inline namespace A {
  namespace B {
    export void Widget ();
 }
}

export module User;
import Impl; // Internal use
namespace B {
  export void Frobber (); // #2
}
```

User has a *silent* ABI dependence on Impl – Frobber is actually in A::B::Frobber. If User no longer needs to import Impl, the code will continue to compile without error, but User's ABI stability will have been broken.

The only way User may defend against this is by not importing any modules, and not including anything in its Global Module Fragment.

If DR2061 was not in effect, there would be no such ABI fragility. A failure mode would manifest as compilation errors due to the ambiguity of ::B and ::A::B during name lookup. This could occur in User, but only in importers of User if they also imported Impl (or namespace ::A becomes visible to name lookup via some other import or declaration).

3 Discussion

The use shown in PR90291 conflicts with the direction taken in DR2061. The user's rationale is reasonable. That the report was nearly a year after compiler release is probably indicative of the user's compiler-update cadence (rather than bug obscurity). As G dos Reis notes, a scheme with similar behaviour is now used in the STL. GCC encountered a few other bug reports related to the DR2061 change, and has implemented a workaround for that change in the unnamed namespace case.

3.1 Cologne 2019

At the Cologne meeting, EWG gave guidance that neither of the two options presented:

Although no formal polls were taken, the opinion was expressed that unnamed namespaces at different levels of an inline hierarchy should continue to work as they had done prior to DR2061. Also, there was no preference for either of the two options presented:

1. When a namespace definition uses a qualified name, should lookup of the qualifying names search inline namespaces? That would match the behaviour of other qualified-name declarations, but break the equivalence between using a qualified name, or an explicit nest of namespace definitions.
2. (If answer 1 is ‘no’), should an approach suggested by D Herring be taken, and prevent creating new namespaces whose name matches an existing namespace within their local inline namespace nest?

Both were equally disliked.

3.2 Kobayashi Maru

Given the dislike of solutions modifying DR2061’s behaviour, let us reexamine the motivation for DR2061. DR2061 is motivated by the following user code, specializing a standard library template:

```
namespace std {
namespace ranges {
template<> constexpr bool disable_sized_range<MyType> = true;
}
}
```

Prior to DR2061, the user code will fail to compile if the library has inserted an inline namespace between std and ranges. Should the user have used the qualified name std::ranges::disable_sized_range, all would be well.

DR2061 has exchanged a fixable compilation error, for the possibility of silent ABI changes and latent poisoning of namespace names. Further it has added an exception to the rule that unqualified names that can declare new named entities only consider the current namespace. These do not seem equitable transactions.

3.3 Summer ’20

Revision 1 of this paper was presented to EWG virtual meeting in the summer of 2020. In addition to the simple reversal of DR2061, EWG would like to consider more closely, primarily via a Tony Table, a few alternatives, also raised at the Cologne ‘19 meeting.

0	Status Quo	C++ 20
1	Revert DR2061	Pre-dr2061 behaviour, ‘namespace Foo’ ignores any inline namespace set of the current namespace.
2	Open qualified namespace	As #1 but when opening a qualified namespace name behave more like other qualified name lookups and consider inline namespace sets for each component of the name lookup.

3	Check Inline Nest	It is ill-formed to create a namespace such that an inline-namespace set contains an ambiguous namespace lookup. (From D Herring)

Any combination of #1-#3 could be applied.

#3 prevents inadvertently creating ambiguous lookups, at the point of creation, rather than discovery.

#2 is somewhat orthogonal, but has been mentioned in discussions of this paper. In the absence of #1, it is the status quo. It is intended to make using a qualified name to declare a namespace consistent with uses of qualified names in other declarations. As opening a qualified namespace name is specified as a sequence of unqualified openings, the existing behaviour of searching inline namespace sets provides this behaviour. If #1 is accepted, additional wording is needed to retain this behaviour. Some finessing may be required in avoiding using directives and how to handle creation to avoid breaking existing source.

Namespace Poisoning	**0**	**1**	**1+2**	**3**
	Status Quo	**Revert DR2061**	**+Qualified Lookup**	**Check Nest**

`namespace B {`
`inline namespace V1 {} // B::V1`
`namespace C {} // B::C`
`}`				

`namespace B::V1::C {}`	Create B::V1::C			Error, ambiguates B::C

`namespace B {`
`namespace C { … }`
`}`
	Error ambiguous	Enter B::C	Enter B::C	Enter B::C (There is no B::V1::C)

`namespace B::C {...}`			Error ambiguous	

Intermediate inline namespace	**0**	**1**	**1+2**	**3**
	Status Quo	**Revert DR2061**	**Qualified Lookup**	**Check Nest**

namespace std {
inline namespace V1 { // std::V1
namespace ranges { // std::V1::ranges
template <typename T> class X {…};
}}}				

namespace std {
namespace ranges {
template<>
class X<myclass> {…};
}
}
	Enter std::V1::ranges, specialize X	Error, creates std::ranges, cannot find X	Error, creates std::ranges, cannot find X	(if #1 or #2 in play)
				
				Error, ambiguates std::V1::ranges

namespace std::ranges {
template<>
class X<myclass> {…};
}
			Enter std::V1::ranges, specialize X	(if only #1 in play)
				
				Error, ambiguates std::V1::ranges

template<>
class
std::ranges::X<myclass>
{…};
	Specialize std::V1::ranges::X			

Innermost inline namespace	**0**	**1**	**1+2**	**3**
	Status Quo	**Revert DR2061**	**Qualified Lookup**	**Check Nest**

namespace std {
inline namespace V1 { // std::V1
template <typename T> class Y {…};
}				

namespace std {
template<>
class Y<myclass> {…};
}
	Enter std, specialize V1::Y			

template<> class std::Y<myclass> {…};	Specialize std::V1::Y			

To address the confusion about the uniqueness of the fabricated name for an *unnamed-namespace-definition* there are two options:

(a) All *unnamed-namespace-defnitions* use the same fabricated name, or

(b) All *unnamed-namespace-definitions* with different containing scopes use different fabricated names.

(a) means that it is possible to construct unreopenable inaccessible inline unnamed namespace sets (unless D Herring's suggestion prohibits them). (b) removes the possibility of that happening.

I believe (a) is the Status Quo, and its implication is that, in general, there can be at most one unnamed namespace with, or reachable from, an inline namespace set, if one wishes to reopen that anonymous namespace. There was considerable dissatisfaction with the observed failure mode, as mentioned above.

		(a)	**(b)**	**(a) + #3**
	Status quo	**Same Unique Identifier**	**Scope-dependent Unique Identifier**	**Same Unique Identifier, No Ambiguous Hierarchies**
inline namespace {}				
namespace { namespace {} }	Constructs <anon>::<anon>			Error, would create ambiguous hierarchy
namespace {}	?	Ambiguous <anon> or <anon>::<anon>?	Enters <anon>	

4 Proposal

1) Revert DR2061, restoring the prior behaviour of only searching the current namespace. Specifically:

Change 9.8.2.1 [namespace.def.general] paragraph 2 as follows:

In a *named-namespace-definition D*, the *identifier* is the name of the namespace. The *identifier* is looked up by searching for it in the scopes of the namespace A in which D appears ~~and of every element of the inline namespace set of A~~. If the lookup finds a *namespace-definition* for a namespace N, D *extends* N*, and the target scope of D is the scope to which N belongs*. If the lookup finds nothing, the *identifier* is introduced as a *namespace-name* into A.

2) Add a library note that as specializations of templates may be declared 'anywhere the primary template may be defined' [tmpl.expl.spec]/3, user specializations of library templates should be declared using a qualified name so that inline namespace sets are searched for the template. (There appears to be compiler divergence on permitting such out-of-scope declarations.)

Alter [namespace.std]/2:

> Unless explicitly prohibited, a program may add a template specialization for any standard library class template to namespace std provided that (a) the added declaration depends on at least one program-defined type and (b) the specialization meets the standard library requirements for the original template.167 ***[Note: As vendors may insert inline namespaces inside std, user specializations of library-provided templates should avoid using nested [unqualified][1] named-namespace-definitions to enter the template's scope.***
>
> ***[Example:***
>
> ```
> template<>
> constexpr bool std::ranges::disable_sized_range<MyType>
>   = true;
> ```
>
> ***– end example] – end note]***

[1] If the qualified namespace opening change is accepted, this note should be restricted to nested unqualified namespace declarations.

3) If the qualified namespace opening change (#2) is accepted, add wording to alter opening a nest of namespaces using a qualified name:

Replace [namespace.def.general]/8:

> ~~A *nested-namespace-definition* with an *enclosing-namespace-specifier* E, *identifier* I and *namespace-body* B is equivalent to~~
>
> ~~namespace E { inline~~$_{opt}$ ~~namespace I { B } }~~
>
> ~~where the optional inline is present if and only if the *identifier* I is preceded by inline.~~
>
> ~~*[Example 2 :*~~
>
> ```
> ~~namespace A::inline B::C {
>  int i;
> }~~
> ```
>
> ~~The above has the same effect as:~~
>
> ```
> ~~namespace A {
>  inline namespace B {
>   namespace C {
>    int i;
>   }
>  }
> }~~
> ```

~~*— end example]*~~

with:

> **A *nested-namespace-definition* performs a series of lookups for each component of the *enclosing-namespace-specifier*, and the final *identifier*. The inline namespace sets of each searched namespace are also searched. If the lookup finds nothing, the component is introduced as a namespace member of the directly searched namespace. The final namespace is extended. If a component of the *enclosing-namespace-specifier*, or the final *identifier*, is preceded by `inline`, the found namespace (if any) must be an inline namespace. When creating a new namespace, a preceding `inline` causes an inline namespace to be created.**
>
> **[*Example 2:***
>
> ```
> namespace A::inline B::C {
>   int i;
> }
> namespace A::C {
>   int i; // error, redefinition of A::B::C::i
> }
> ```
>
> ***– end example*]**

4) If D Herring's suggestion (#3) is accepted, add wording to make it ill-formed to create ambiguous namespace hierarchies:

Add new paragraph [namespace.def.general]/7½:

> **When a new namespace is to be created, it is ill-formed if the new namespace name is the same as the name of an existing member[1] of the innermost enclosing non-inline namespace or its inline-namespace set.**

[1] Should 'member' be restricted to members that are namespaces?

5) Adjust [namespace.unnamed]/1 To make it clear that either all unnamed namespaces use the same *unique* identifier, or that unnamed namespaces with different containing scopes use different *unique* identifiers.

5.1) Option A: they are all the same:

Alter [namespace.unnamed]/1

> An *unnamed-namespace-definition* behaves as if it were replaced by
>
> inline$_{opt}$ namespace *unique* { /* empty body */ }
> using namespace *unique* ;
> namespace *unique* { *namespace-body* }

where inline appears if and only if it appears in the *unnamed-namespace-definition* and **every *unnamed-namespace-definition* in a translation unit replaces** ~~all occurrences of~~ *unique* ~~in a translation unit are replaced by~~**with** the same identifier, and this identifier differs from all other identifiers in the translation unit. The optional *attribute-specifier-seq* in the *unnamed-namespace-definition* appertains to *unique*.

5.2 Option B: they are all different, but scope-dependent:

Alter [namespace.unnamed]/1

An *unnamed-namespace-definition* behaves as if it were replaced by

inline$_{opt}$ namespace *unique* { /* empty body */ }
using namespace *unique* ;
namespace *unique* { *namespace-body* }

where inline appears if and only if it appears in the *unnamed-namespace-definition* and all occurrences of *unique* **in each scope** in a translation unit are replaced by the same scope-specific identifier, and this identifier differs from all other identifiers in the translation unit. The optional *attribute-specifier-seq* in the *unnamed-namespace-definition* appertains to *unique*.

4.1 Ship Vehicle

A DR against C++20

5 Revision History

R0 First version

R1 Added Modules case, documented Cologne guidance, suggested resolution.

R2 Added EWG-requested Tony Table, alternative suggestions and wording options.