Document Number: p1701r0
Date: 2019-06-14
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.

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
```

2 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.

3 PR90291

I implemented DR2061 in GCC 8. 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 namespace, but locates the previously opened A::detail at #1. Thus the second function's fully qualified name is A::detail::C::bar.

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

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.

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.

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

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 4)
- 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?

4 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

5 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.

Questions:

1. Should inline namespaces be searched when opening the namespace of a namespace definition? This agrees with DR2061's resolution but breaks the use case of PR90291.
2. 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.
3. (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?

6 Revision History

R0 First version