Document Number:	P1473R0
Date:	2019-01-21
Audience:	Library Evolution Working Group
Author:	Casey Carter
Reply to:	casey@carter.net

Shadow namespaces

1 Abstract

Usage of `using namespace std;` means we break existing code in every new C++ standard. Our mechanisms for name collision avoidance reduce but do not eliminate the potential for collisions, do so by reducing the quality of the Standard Library, and are not sustainable. Instead of demanding that users pay the cost of avoiding name collisions by typing `std::` everywhere we should provide an alternative mechanism that is equally convenient. This paper proposes such a mechanism in a familiar guise:

```
using namespace std::cpp17;
```

Table 1 — Tony Table

Before	**After**

`using namespace std;`
`using span = int[42];`
`vector<int> vec;`
// ill-formed; ambiguous:
`span x;`
	`using namespace std::cpp17;`
	`using span = int[42];`
	`vector<int> vec;`
	// Ok:
	`span x;`

`#include <range/v3/core.hpp>`
`using namespace std;`
// ill-formed; ambiguous:
`auto first = ranges::begin(vec);`
	`#include <range/v3/core.hpp>`
	`using namespace std::cpp17;`
	// Ok:
	`auto first = ranges::begin(vec);`

1.1 Revision History

1.1.1 Revision 0

— Sprang fully formed from the forehead of Zeus.

2 Problem description

Name collisions between the Standard Library and user programs that pull the entire standard library into their namespaces - or the global namespace - via `using namespace std;` are a perennial problem for C++. WG21 is divided as to whether we should be responsible for avoiding such collisions. On the one hand no one wants to break existing code, and on the other no one wants to pessimize name choices in the library. This is a constant source of problems for us, with the typical result being that big companies with loud voices in the committee are insulated from name collisions and Joe Programmer is left out in the cold.

We use two mechanisms to avoid collisions: either replace the good name[1] with a bad name - which is problematic for obvious reasons - or relocate the good name into a subnamespace of `std` - which is problematic for less obvious reasons. Subnamespaces make it harder for people who do fully qualify names to use the standard library - they're paying for what they don't use.

Neither mechanism avoids collions entirely: the replacement name may be less likely to cause collisions, but the potential is still there. Neither mechanism is sustainable: as the body of C++ code grows over time,

1) Perhaps "chosen name" would be a better term than "good name" to describe the output of LEWG.

and more names are added to the standard library, the supply of usable names gradually dwindles. We'll eventually run out of names to use directly in `std` either as replacements or subnamespace names, and have to start increasing the nesting depth of subnamespaces rendering the library unusable without namespace gymnastics.

For a recent example, consider P0631R4 "Math Constants" [1] which adds definitions of common mathematical constants to the standard library like `pi` and `e`. The consensus opinion on the reflector was to move the constants into `std::math` to avoid collisions. Note that `math` is far from an uncommon name; we've traded one problem for another.

There have even been rumbles in LEWG about deprecating `using namespace std;` to *force* users to fully qualify names they use from the standard library or write using declarations for individual names. It's hard to envision this kind of adversarial tactic being well-received by users. What if, instead of harming ease of use by trying to get users to stop doing this thing we want them to not do, we gave them an alternative that doesn't require them to do more work?

3 Proposal

If users won't write using declarations for the standard library names they want to use unqualified, why don't we do it for them? This paper proposes adding nested namespace `std::cpp17` and `std::cpp20`. within these nested namespaces `std::cpp`X (where X is either 17 or 20) there shall be a shadow of the entire name structure of the standard library as it was defined in C++ version X.

To be precise, in each standard library header:

— For each namespace `std::`$\cdots$`::`N defined in that header in C++X, there shall be a corresponding shadow namespace `std::cpp`X`::`$\cdots$`::`N.

— For each namespace alias `std::`$\cdots$`::`A defined in that header in C++X that denotes a namespace `std::`$\cdots$`::`N, there shall be a corresponding shadow namespace alias `std::cpp`X`::`$\cdots$`::`A that denotes namespace `std::cpp`X`::`$\cdots$`::`N.

— Each other name E in namespace `std::`$\cdots$`::`N defined in that header in C++X shall have a corresponding *using-declaration* in `std::cpp`X`::`$\cdots$`::`N:

```
using ::std::···::N::E;
```

— Each enumerator E of unscoped enumeration type T defined in namespace `std::`$\cdots$`::`N in that header in C++X shall have a corresponding *using-declaration* in `std::cpp`X`::`$\cdots$`::`N:

```
using ::std::···::N::E;
```

The end result is that `std::cpp`X in each header mirrors the structure and content that existed in that header in `std` in C++X. The meaning of those names may change in future revisions of C++, but the set of visible names will not.

How can this prevent name collisions for programs that `using namespace std;`? Simple - it can't. It *can* prevent name collisions from names newly added in C++20 for programs that `using namespace std::cpp17;`, however, and prevent name collisions from names newly added in C++23 for programs that `using namespace std::cpp20;`. These shadow namespaces are exactly the mechanism required. Replacing `using namespace std;` with `using namespace std::cpp17;` requires almost no additional effort from users - it's 7 more keytrokes per TU, rather than 5 keystrokes per reference to the standard library - and the replacement is easily toolable - a `sed` script could update an entire source tree in no time with almost no chance for error.

The simultaneous presence of multiple shadow namespaces has the additional benefit of allowing for gradual transition of a codebase from an old standard library to a new one. If you replace `using namespace std;` with `using namespace std::cpp17;`, your program is substantially more likely to compile after throwing the C++20 compiler switch. You are then free to add references to / *using-declarations* for fully-qualified names in `std` to get at new features without touching the rest of the TU, change 17 to 20 TU-by-TU, or even stay at `cpp17` indefinitely while continuing to upgrade compilers and core language versions in the expectation that your standard library implementation will continue to support `std::cpp17`.

3.1 What about Argument Dependent Lookup?

Argument Dependent Lookup ([basic.lookup.argdep]) pokes some holes in the proposal. For example, this C++20 program:

```
#include <vector>
using namespace std::cpp20;
int frobnozzle(vector<int>&) { /**/ }
int main() {
  vector<int> vec;
  frobnozzle(vec);
}
```

would be broken if we add a function with the signature `std::frobnozzle(vector<int>&)` in C++23. Since `std::cpp20::vector` and `std::vector` are the same template, ADL for the unqualified call to `frobnozzle` in `main` will find the new signature in `std` - shadow namespaces or not - resulting in overload resolution ambiguity. This proposal greatly reduces the surface for name collisions, but doesn't eliminate them completely when ADL is a factor.

There is ongoing work in both EWG and LWG to address issues with ADL which will likely help with this problem if not solve it outright.

3.2 Messaging

Together with making this change, we as a community need to change our messaging on *using-directive*s. The Standard should provide normative encouragement for implementations to diagnose `using namespace std;` as a dangerous practice and suggest `using namespace std::cpp`*X* instead. WG21 should change its guidance from "never write `using namespace std`" and/or "write `using namespace std;` in CPP files" to "write `using namespace std::cpp`*X* in CPP files if you like, and never ever write `using namespace std;` or we will gleefully break your program."

3.3 Brains...

We should simultaneously zombify all names matching the regular expression `cpp\d+` for past and future standardization to defend against macroization. This both prevents defining `cpp23` in C++20 programs which would then fail to compile in C++23, and allows vendors to indefinitely maintain support for older versions of the standard in e.g. `std::cpp11` or `std::cpp98`.[2]

3.4 How many versions?

How far back should the required shadow namespaces go? The proposal above obviously suggests at least coverage of "current" and "previous" Standard revisions. Should we go back further? Should we start at two and grow to N? Grow indefinitely? This seems to be a question of (a) the cost to implementations to maintain (or implement, for fresh implementations) old shadow namespaces and (b) the cost to WG21 to maintain the specifications of such in the IS.

3.5 IS Maintenance

WG21 maintenance costs depend on the form of wording for this feature that's acceptable to LWG and the Editor. Wording could be as simple as listing the above bullets that describe the proposal in the library front matter and indicating the number of required shadow namespaces, which would require no maintenance, or as detailed as realizing the complete set of required declarations concretely in each header. The latter would require per-IS-cycle maintenaince in which we strike the oldest shadow namespace from each header synopsis - if we choose not to maintain them indefinitely - and duplicate the most recent shadow namespace into the shadow namespace for the current standard. It would also require careful monitoring by LWG to ensure that each proposal that adds a new name to `std` simultaneously adds that same name to the current shadow namespace.

3.6 Painting the bikeshed

Obviously `std::cpp`*X* is not the only possible naming scheme for the shadow namespaces. `std`*X* has already been suggested to the author, which has two benefits:

— those namespaces have been reserved since C++17, and

2) At least until 2098 when WG21 will need to decide whether to drop support for C++98 or define the new shadow namespace as `std::cpp2098`, but hey - that's not my problem!

— `using namespace std17;` is 5 keystrokes shorter than `using namespace std::cpp17;`

although it seems a shame to paint ourselves into a corner and kill the dream of `std2` die completely.

4 Technical specifications

Wording for this proposal will be straightforward - but possibly voluminous - once LWG and the Editor decide what form the wording should take. Rather than proving page upon page of wording for alternative forms here which would only contribute to the word cloud for the pre-meeting mailing, the author awaits direction on the desired form.

Bibliography

[1] Lev Minkovsky and John McFarlane. P0631R4: Math constants, 08 2018. https://wg21.link/p0631r4.