Doc.No.: Pxxxx
Project: Programming Language - C++ (WG21)
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2019-05-30
Audience: EWGI
Vehicle: C++23

Proposal of Namespace Templates

1. Abstract

We describe the library configuration problem, current solutions (and why they are lacking), and then propose *initial namespace templates* and *namespace template extensions* as an elegant solution.

2. The Library Configuration Problem

Let us define a *library* as a set of related source files that share a common directory, where all the entities defined in its source files are placed in some exclusive namespace for the library.

eg A directory called `libfoo` containing files:

```
foo1.h
foo1.cc
foo2.h
foo2.cc
foo3.h
foo3.cc
```

...where each `fooi.{h|cc}` contains:

```
/* preamble */

namespace foo {

/* entity declarations and definitions */

}  // namespace foo
```

...we would call a library (“the foo library”).

Many times there are domain-specific cross-cutting characteristics of a library.

For example:

- For a physics simulation library, does it use single-precision or double-precision floating-point arithmetic?
- For a containers library, what kind of memory allocation scheme does it use?
- For an audio signal processing library what frequency and bitrate does it normalize to and operate on?
- For some virtual machine library, what are its limits? Max threads? Max Memory? etc

Let us call a set of answers to such questions about some specific library as a *library configuration*.

Rather than duplicating the code of a library for each possible library configuration, we would like to define the library, once, in terms of an abstract configuration. Users of the library can then select which concrete configuration they would like to use.

There are several different ways this is handled today.

2.1. Hard-code Library Configuration

The simplest solution is to define the library configuration as normal entities in a library config header. Each source file of the library then includes that header and gets the definitions (which it uses):

`libfoo/config.h`

```
#pragma once

namespace foo {

// libfoo configuration
using RealType = float;
constexpr size_t normalized_bitrate = 32’000;
constexpr int max_heap_size = 1’000’000;
/* ... */

}  // namespace foo
```

If the library configuration is changed it can be achieved by modifying the source code of the config header.

This isn't ideal as it requires source code changes whenever the configuration changes.

2.2. Preprocessor-based Library Configuration

To improve on the situation one can use the preprocessor / macros / conditional inclusion to generate the config header based on defines (-Dkey=value) passed into the implementation. There are countless examples of this in the wild, for one example see luaconf.h from liblua.

This isn't great as the data model of defines is not very expressive and is tough to manage.

2.3. Code-generated Library Configuration

Another approach is to generate the config.h header in a pre-build step. Again, there are numerous examples of this. For this we would point at things like the autotools system that generates a config header based on run-time testing of the target system. There are also systems that take a configuration in some declarative format (like JSON) and then code translate them into a C++ header.

All of the approaches mentioned thus far don't allow for multiple instances of the library, each with different configurations, to be included in the same program.

Imagine a situation where a library A and a library B both use some configurable library C, each with a different configuration of C (lets call the two configurations Ca and Cb). Now we want to create a program that uses both library A and library B. Roughly speaking this is an ODR violation and doesn't work, as there are two different definitions of the same entity in the same program.

2.4. Entire Library in Class Template

To solve this problem one could put the entire library into one big source file and wrap it in a class template. Library A could then use C<Ca> and library B could then use C<Cb>, avoiding the linking problem. However placing an entire library into a single source file goes against the best practice of keeping source files small. (There are also a number of features that namespaces have that classes don't.)

2.5. Template Each Entity in Library

Another solution is to make every entity in the library an entity template that is parameterized on the library configuration. This requires an enormous amount of boilerplate as the library configuration is repeated (a DRY violation) not only in every source file, but for every entity in every source file in the library, and, even worse, must be passed around the library. Many times when the definition of an entity X in the library makes use of some entity Y, it must explicitly instantiate Y<Config> with the library configuration.

2.6. Use Namespace Templates (proposed)

This library configuration problem comes up a lot in practice. It has been independently suggested at least four times by four different groups over the years that namespace templates would be an elegant solution to the problem.

The way the solution works is that the library configuration is the template parameter set of the namespace template that encloses the library. The library is defined in terms of those abstract template parameters. Users of the library instantiate the namespace template with the concrete configuration (which are the template arguments in the namespace template-id).

3. Namespace Template Design

The obvious first move here (and the least surprising) is to say that a namespace template is "just like" a class template:

A class template…

```
template<typename T>
class C {
  /* ... */
};
```

A namespace template...

```
template<typename T>
namespace N {
  /* ... */
};
```

We are left with a couple of design issues.

The first is that this doesn't (yet) address the problem we are trying to solve. The key feature we want here is to be able to define the abstract library configuration (once) in one source file, and then use that abstract library configuration from the multiple other library source files. That's the main reason we can't just use a class template in the first place.

So what we propose is that while there must be a single initial definition, of the form above, of a namespace template, we call the *initial namespace template*, there can be further definitions, that are logically concatenated during translation (in a similar fashion to normal namespace definitions). We call these further definitions, *namespace template extensions*. So that we don't need to keep redeclaring the same template parameters, we shorten the namespace template extension syntax so that this is done implicitly (based on the *initial namespace template*).

```
// initial namespace template
template<typename T>
namespace N {
    /* ... */
}

// namespace template extension
namespace template N {
    /* ... */
}
```

The intent here is that:

1. The initial namespace template appears in the library configuration header file.
2. Each of the other source files of the library include that header file.
3. Each other source file of the library uses a namespace template extension.

A small toy program to illustrate the proposed mechanics:

```
template<int i>
namespace n {
    constexpr int x = i + 2;
}

namespace template n {
    constexpr int y = x + i + 3; // both x and i are in scope
}

namespace template n {
    constexpr int z = 2 * y;
}

int main() {
    static_assert(n<10>::x == 10 + 2);

    using namespace n<100>;
    static_assert(y == 102 + 100 + 3);
    static_assert(z == 410);
}
```

4. Issues

4.1. Namespace template parameters

The new entity that is proposed in this paper is called a “namespace template” (the concatenation of an initial namespace template and any namespace template extensions). It is a template used to create one or more namespaces. A usage of that template would be a “namespace template instantiation” which is a namespace. This is like how a usage of a “class template” is a “class template instantiation” which is a class type.

Separately, a “namespace template parameter” would be a template parameter that is a namespace:

```
// NOT proposed in this paper.
template<namespace N>  // a namespace template parameter
class C {
    /*...*/
}
```

Template parameters can currently be type template parameters, non-type template parameters and template template parameters. A namespace template parameter would be a fourth kind of template parameter in this list.

The relationship between a namespace template and a namespace template parameter, is similar to the relationship between a variable template and a non-type template parameter - in that the template/template-parameter pair both relate to the same kind of underlying entity. The first of each pair is a template of the entity, the second is when the entity is used as a parameter to a template.

Namespace template parameters are not proposed in this paper. During initial discussions of namespace templates, namespace template parameters came up.

The proposal author's position on namespace template parameters is that the motivation is completely different for them to namespace templates. This can be illustrated by the fact that we had non-type template parameters long before we had variable templates. We can easily imagine a C++ with any of:

- namespace templates and not namespace template parameters, or
- namespace template parameters and not namespace templates, or
- both namespace templates and namespace template parameters.

For this reason we suggest that namespace template parameters should be advanced separately, and independently, by a separate line of proposals.

4.2. Out-of-class definitions of members of a class template.

It has been suggested that another possible motivation for something like namespace templates is cutting down the boilerplate on out-of-class definitions of class template members....

ie Instead of:

```
template<typename T>
class C {
    void f();
    void g();
    void h();
}

template<typename T>
void C<T>::f() { /* … */ }
template<typename T>
void C<T>::g() { /* … */ }
template<typename T>
void C<T>::h() { /* … */ }
```

Maybe we could write something like:

```
template<typename T>
class C {
    void f();
    void g();
    void h();
}

// NOT PROPOSED
template<typename T> {
void C<T>::f() { /* … */ }
void C<T>::g() { /* … */ }
void C<T>::h() { /* … */ }
}
```

..to save the boilerplate of repeating the template prefix.

While saving on boilerplate is certainly interesting it pales in comparison to the utility of namespace templates vs the “Template Each Entity in Library” solution (and other solutions) to address the library configuration problem.

The proposal author (at the time of writing) does not currently see a unified design for namespace templates that both addresses the library configuration problem and addresses this out-of-class use case. Therefore, while we remain open to ideas about this, we are not proposing addressing this use case at this time with namespace templates.

4.3. Integration with nested namespaces

We propose a syntax for nested namespace templates, based on syntactic equivalence, such that

```
template<typename T> namespace A::B {}
```

...is equivalent to...

```
namespace A {
template<typename T>
namespace B {}
}
```

And such that:

```
namespace A::template B::C::template D::E {}
```

is equivalent to:

```
namespace A {
namespace template B {
namespace C {
namespace template D {
namespace E {
}}}}}
```

This means that:

- An initial namespace template can only be the last namespace in a nested namespace definition
- A namespace template extension can appear anywhere in a nested namespace definition

So if you wanted to introduce two nested namespace templates you can write:

```
template<int a>
namespace A {};

template<int b>
namespace template A::B {
    constexpr int c = a + b;
}
```

4.4. Namespace Templates and Modules

What best practices surrounding modules that will evolve in the wild after C++20 is released isn’t entirely clear at the time of writing. We assume source files will remain small and for the use of namespaces to enclose libraries to persist. That given, the intent is that an initial namespace template should continue to be possible to define on it’s own in one source file, and for the other source files to be able to extend on it with a namespace template extension.

To demonstrate using the previous toy example **without modules** might look like:

Example 1 - WITHOUT MODULES

```
// FILE X.H
#pragma once
template<int i>
namespace n {
    constexpr int x = i + 2;
}

// FILE Y.H
#pragma once
#include “X.H”

namespace template n {
    constexpr int y = x + i + 3; // both x and i are in scope
}

// FILE Z.H
#pragma once
#include “Y.H”
namespace template n {
    constexpr int z = 2 * y;
}
```

```
// FILE MAIN.CC
#include “Z.H”
int main() {
    static_assert(n<10>::x == 10 + 2);

    using namespace n<100>;
    static_assert(y == 102 + 100 + 3);
    static_assert(z == 410);
}
```

And then here is the same example with modules:

Example 2 - WITH MODULES

```
// FILE X.H
export module X;
template<int i>
namespace n {
    constexpr int x = i + 2;
}

// FILE Y.H
export module Y;
import module X;

namespace template n {
    constexpr int y = x + i + 3; // both x and i are in scope
}

// FILE Z.H
export module Z;
import module Y;
namespace template n {
    constexpr int z = 2 * y;
}

// FILE MAIN.CC
import module Z;
int main() {
    static_assert(n<10>::x == 10 + 2);

    using namespace n<100>;
    static_assert(y == 102 + 100 + 3);
    static_assert(z == 410);
}
```

4.5 Explicit specializations, partial specializations, and explicit instantiations

Given the problem we are trying to solve (library configuration), we don't think having multiple specializations of a namespace template is well-motivated. So we propose for the first version of the feature that only a primary namespace template may be defined. We can always add partial specializations and explicit specializations as an extension later.

Explicit instantiations are however well-motivated. You may want to have a specific list of library configurations that you want to instantiate for the usual compile-time savings. So we propose that explicit instantiations work as expected:

```
   extern template namespace N<args>;
```

declares the explicit namespace template instantiation N<args> and

```
   template namespace N<args>;
```

defines it. The declaration suppresses implicit instantiation, and the definition explicitly instantiates all the content of the original namespace template and any namespace template extensions available to it.

4.6 Entities that can be in namespaces but not classes

There are a number of constructs that can appear in namespace definitions but not within class definitions. As this proposal progresses we intend to go through each one to make a design decision as to whether each such construct may or may not also appear within namespace templates. In general this is a balance between implementation feasibility and whether there is strong motivation to have each construct within a namespace template.

5 Proposal Roadmap

Our goal is to get feedback from WG21 on the described motivation and design decisions we have put forward to determine whether there is sufficient support to justify an investment in preparing wording and implementation.

6 References

2014 first std-proposals thread started by dgutson
https://groups.google.com/a/isocpp.org/d/msg/std-proposals/8IDbb1L5_UA/TVdjru4mFCEJ

2017 second std-proposals thread started by Jake Arkinstall
https://groups.google.com/a/isocpp.org/d/msg/std-proposals/3MR8IhevevU/HQNEFn5fAwAJ