Proposal for C2y
WG14

Document Number: N3794
Author: Abdulmalek Almkainzi <aalmkainzi@gmail.com>
Title: Namespacing with prefixes
Proposal category: New Features
Target Audience: General Developers

Abstract:
Prefixing identifiers has been the de facto standard way C developers avoid name collisions in their APIs. This proposal aims to add proper namespaces to C without implicit name mangling by utilizing prefixes.

Table of Contents

Introduction and Rationale

C library developers should prefix their entire exposed API, otherwise name collisions may occur. Having to write the library prefix for every identifier the library exposes can be tedious, both for library developers and their users.

Consequently, it isn't uncommon for some libraries to expose identifiers that aren't properly prefixed, which can be problematic (e.g. X11 [0], raylib [1], json-c [2]).

This proposal has two goals:

- Make creating a prefixed API more convenient.
- Make using prefixed APIs more convenient.

This paper proposes that there should be a language provided construct for namespacing an entire scope, and applying a prefix to all file scope identifiers in that scope. And another construct for converting an already prefixed API into a namespace.

Ultimately, the benefit of namespaces is the ability to make code easier to write and be less noisy:

C23	C2y (with this proposal)

`SDL_Event e;`
 	
`SDL_WaitEvent(&e);`
 	
`if(e.type == SDL_EVENT_QUIT)`
`{`
`    quit = true;`
`}`
 	
`SDL_SetRenderDrawColor(renderer, 0xFF, 0xFF, 0xFF, 0xFF);`
 	
`SDL_RenderClear(renderer);`
 	
`SDL_SetRenderDrawColor(renderer, 0xFF, 0x00, 0x00, 0xFF);`
 	
`SDL_RenderFillRect(renderer, &squareRect);`
 	
`SDL_RenderPresent(renderer);`
	`_Using _Namespace sdl;`
	
	`Event e;`
	
	`WaitEvent(&e);`
	
	`if(e.type == EVENT_QUIT)`
	`{`
	`    quit = true;`
	`}`
	
	`SetRenderDrawColor(renderer, 0xFF, 0xFF, 0xFF, 0xFF);`
	
	`RenderClear(renderer);`
	
	`SetRenderDrawColor(renderer, 0xFF, 0x00, 0x00, 0xFF);`
	
	`RenderFillRect(renderer, &squareRect);`
	
	`RenderPresent(renderer);`

Proposal

Identifiers that can enter a namespace are file scope identifiers. These include struct and union tags, type names, global variables, function names, enumeration constants, and namespaces. Namespace definitions are only allowed at file scope, and identifiers that are declared in the namespace scope are still considered to be file scope.

This paper proposes two ways to create a namespace. The first adds file scope identifiers declared in a scope to a namespace (if it isn't already added) and, except for nested namespace names, prefixes the identifiers with a specified prefix. This will be referred to as an **apply-prefix** namespace scope:

```
_Namespace mylib += "MyLib_"
{
    struct S {
        char c;
    };

    int count;

    struct S foo();
}

int main()
{
    mylib::count = 0;
    mylib::foo();
    struct mylib::S s;

    MyLib_count = 0;
    MyLib_foo();
    struct MyLib_S s2;
}
```

An apply-prefix scope is useful for creating new library code. It makes it so that library developers omit the prefix they would normally include for every file scope identifier.

File scope identifiers declared inside a namespace scope and contain a namespace access are not prefixed by the apply-prefix scope, nor are they added to the namespace:

```
_Namespace B += "B_"
{
    extern int B_f;
}
_Namespace A += "A_"
{
    int ::i;
    extern int B::f;
}

int main()
{
    int a = i;    // ok
    int b = B::f; // ok
    int c = B_f;  // ok
    int d = A::i  // error
    int e = A::f; // error
}
```

In order to make using existing prefixed libraries easier for users, this paper also proposes another way to define namespaces, such that prefixed identifiers in a scope enter a namespace with their prefixes omitted. The idea is to create a mapping from a prefix to a namespace, and matching file scope identifiers in the scope against that mapping. This will be referred to as a **capture-prefix** namespace scope:

```
_Namespace bits -= "stdc_"
{
    #include <stdbit.h>
}

int main()
{
    return bits::leading_zeros_ui(1);
}
```

It’s important to note that a namespace doesn’t have to be only associated with a single scope, users are allowed to reopen the namespace with a different scope:

```
_Namespace mylib += "MyLib_"
{
    void foo();
}

_Namespace mylib -= "MyLib2__"
{
    void MyLib2__bar();
}

int main()
{
    mylib::foo(); // calls MyLib_foo
    mylib::bar(); // calls MyLib2__bar

    MyLib_foo();
    MyLib2__bar();
}
```

If the prefixed declarations collide, a constraint violation occurs:

```
_Namespace A += "h_"
{
    int foo(); // becomes h_foo
    void bar(); // becomes h_bar
}

_Namespace B += "h"
{
    int _foo(); // ok, becomes h_foo
    int _bar(); // error, h_bar declared with different type
}
```

In the above example `A::foo` and `B::_foo` refer to the same function, which is allowed. The error occurred because the function `h_bar` was declared twice with a different type signature, as if they were written fully prefixed without namespaces.

If any collisions happen within a namespace, a constraint violation occurs:

```
_Namespace A -= "a_"
{
    int a_foo(); // A::foo
}

_Namespace A -= "A_"
{
    int A_foo(); // error
    // A::foo already exists and refers to a different function
}
```

Name Resolution

Referring to an identifier by using a namespace is done by putting the `::` punctuator between the namespace name and the identifier's unprefixed name. If the namespace name is not provided, then the identifier is searched for in the outermost file scope:

```
constexpr int i = 5;

_Namespace A += "A_"
{
    constexpr int i = 10;

    int foo()
    {
        return ::i; // returns 5
    }

    int bar()
    {
        return i; // returns 10
    }
}
```

When inside a namespace scope, the unprefixed names can be used without a namespace access, except in capture-prefix scopes with identifiers that were captured by the prefix that the scope is

capturing, or with declarations inside a capture-prefix scope. Additionally, identifiers other than nested namespaces can always be referred to with their fully prefixed name.

Identifiers inside a namespace scope with the same unprefixed name as outer file scope identifiers will cause the outer ones to be hidden when inside the namespace scope:

```
int i;
int foo();

_Namespace A += "A_"
{
    int i;
    int foo();

    static inline int bar()
    {
        return i + foo(); // returns A::i + A::foo()
    }
}
```

Essentially, identifiers are resolved by searching from the names available in the innermost nested namespace, continuing outwards if no match found. When there is ambiguity between file scope identifiers at the same nest level, a constraint violation occurs.

Capture-Prefix Scopes

A non-namespace file scope identifier that is declared inside the capture-prefix scope is added to the namespace if it meets the following:

- It does not contain a namespace access `::`
- It starts with the prefix
- It is a valid identifier after the prefix
- It is not already in the namespace

```
_Namespace B -= "b_"
{
    int b_count;  // added to the namespace B
    int b_func(); // added to the namespace B

    int bar();    // not added to the namespace B
    int b_();     // not added to the namespace B
    int b_1();    // not added to the namespace B
}


int main()
{
    return B::count + B::func() + bar() + b_() + b_1();
}
```

When a capture-prefix scope adds an identifier to a namespace, the identifier is added without the prefix that was used to capture it. However, that prefix cannot be omitted when inside a capture-prefix scope that maps from that prefix to the namespace:

```
_Namespace A -= "A_"
{
    void A_foo();

    static inline void A_bar()
    {
        A_foo(); // ok
        foo(); // error
    }
}

_Namespace A -= "AAA_"
{
    static inline void AAA_baz()
    {
        A_foo(); // ok, full prefixed name
        foo(); // ok since this scope doesn't capture "A_"
    }
}
```

When an identifier is added to a namespace, the things that need to be taken into account are:

- The name that enters the namespace (the unprefixed name)
- The fully prefixed name
- The prefix that was used to capture it (if it was added via a capture-prefix scope)

File scope identifiers declared inside a capture-prefix scope are not allowed to use the namespace's available names, the names declared must be as if in outermost global file scope:

```
_Namespace A += "A_"
{
    _Namespace B += "B_"
    {
        void foo();
    }
}

_Namespace A -= "A_"
{
    void A_bar();
    void foo(); // not A_foo, but a new foo declaration ::foo

    void A_bar()
    {
        foo(); // since this isn't a declaration, this calls A::foo
               // because it's at a deeper nest level than ::foo
    }
}
```

The reason for disallowing omitting the prefix when inside a capture-prefix scope that captures the same prefix is backwards compatibility, for example:

```
// lib.h

#ifndef LIB_H
#define LIB_H

#include <errno.h>

extern int lib_errno;

static inline int lib_get_errno()
{
    return lib_errno;
}

static inline int lib_get_std_errno()
{
    return errno;
}

#endif
```

```
// main.c

_Namespace lib -= "lib_"
{
    #include "lib.h"
}

int main()
{
    return lib::get_std_errno();
}
```

If the requirement to not allow omitting the prefix inside a capture-prefix scope wasn't implemented, then main would return the value of `lib_errno` instead of the standard `errno`.

It is allowed to specify multiple mappings when defining a capture-prefix scope:

```
_Namespace
A -= "A_",
B -= "B_",
C -= "C_"
{
    int A_func(); // enters the namespace A
    int B_func(); // enters the namespace B
    int C_func(); // enters the namespace C
}
```

In such case, the scope belongs to all the namespaces specified. The purpose of this is to handle libraries that apply different prefixes. It's also useful for transitive includes, where a header includes another library, and the user wants to namespace both libraries. Example of usage:

```
_Namespace
A -= "A_",
B -= "B_",
C -= "C_"
{
    void A_foo();
    void B_foo();
    void C_foo();
}

// A contains foo which is A_foo
// B contains foo which is B_foo
// C contains foo which is C_foo

_Namespace
A -= "AA_",
B -= "BB_",
C -= "CC_"
{
    static inline void AA_bar()
    {
        A::foo(); // ok
        // A:: not found at current nest level
        // search continues next level up
        // finds A:: and no other A:: exists at that level
    }
    static inline void BB_bar()
    {
        B_foo(); // ok
        // fully prefixed name
    }
    static inline void CC_bar()
    {
        foo(); // error, ambiguous
    }
}
```

The process of determining which namespace an identifier maps to is sequential. A file scope identifier would be checked against the first mapping, if it doesn't match, it's checked against the one after it, and so on.

```
_Namespace
A -= "lib__",
B -= "lib_",
C -= ""
{
    void baz();
    void lib__bar();
    void lib_foo();
    void lib_();
}
```

If an empty string is specified as the prefix in a capture-prefix mapping, all file scope identifiers will match.

In the above example the namespace `A` will contain `A::bar`, `B` will contain `B::foo`, and `C` will contain both `C::baz` and `C::lib_`. If the order of the specified mapping was `B -=

`"lib_"` before `A -= "lib__"`, then `A` would be empty. A warning diagnostic could be raised when a capture-prefix mapping is guaranteed to never match with any identifier. Perhaps an alternative design is preferred, where the longer prefixes have higher priority.

Nesting

In case of an apply-prefix scope being declared directly inside another apply-prefix scope, the inner namespace is added as a member to the outer one, and the prefixes are applied inner prefix first:

```
_Namespace Lib += "lib_"
{
    _Namespace Math += "math_"
    {
        float mul(float a, float b);
        // lib_math_mul == Lib::Math::mul
    }
}
```

In case of a capture-prefix scope being declared inside another namespace scope, the inner namespace will not be added to the outer one.

```
_Namespace S += "S_"
{
    _Namespace A += "A_"
    {
        _Namespace B -= "B_"
        {
            int B_foo();
            int bar();
        }
    }
}
int main()
{
    return B::foo() + B_foo() + bar();
}
```

A capture-prefix scope is never nested by scoping, nor is it ever a parent by scoping. The only way to make nested namespaces with scopes declared inside each other is direct chains of apply-prefix scopes:

```
_Namespace A -= "P_"
{
    void P_foo();
    _Namespace A += "A_"
    {
        void bar();
        _Namespace B += "B_"
        {
            void baz();

            _Namespace C -= "C_"
            {
                _Namespace D += "D_"
                {
                    void foo();
                }
                void C_foo();
                void K_foo2();
                void P_baz();
            }
        }
    }
}

int main()
{
    A::foo();
    A::bar();
    A::B::baz();
    C::foo();
    D::foo();
}
```

In the above example, the namespaces created are as follows:

Namespace	Identifiers (not including nested namespaces)	Fully prefixed
A	foo	P_foo
	bar	A_bar
	baz	P_baz
A::B	baz	A_B_baz
C	foo	C_foo
D	foo	D_foo

Note how the only nesting that happened is `A::B`, that is because they are both apply-prefix scopes, where `B` is directly declared in the scope of `A`.

The reason why `A` contains `baz` (`P_baz`) is because file scope identifiers declared inside a capture-prefix scopes are also candidates to all outer capture-prefix scopes if the identifiers aren’t already members of the current scope’s namespace:

```
_Namespace A3 += "A_"
{
    int bar();
}

_Namespace A -= "A_"
{
    _Namespace A2 -= "A_"
    {
        _Namespace A3 -= "A_"
        {
            int A_foo();
            int A_bar();
            // since A3::bar() already exists,
            // it isn't considered for outer scopes
        }
    }
}
```

The above code results in the following namespaces:

Namespace	Identifiers	Fully prefixed
A	foo	A_foo
A2	foo	A_foo
A3	foo	A_foo
	bar	A_bar

Sub Prefixes

In many cases, a library declares identifiers with sub prefixes (e.g. enumerations prefixed by the enum type name, a category of functions, etc.).

To add those to their own namespace, nested under the library's main namespace, the user can specify a namespace as nested when defining it:

```
_Namespace
sdl::gpu -= "SDL_GPU_",
sdl::gpu -= "SDL_GPU",
sdl -= "SDL_"
{
    #include <SDL3/SDL.h>
}
```

As mentioned previously, the order of capture-prefix mappings matters. In this example, if the order was reversed, the `sdl::gpu` namespace would be empty.

The _Using Keyword

This paper also proposes adding the `_Using` keyword. Its purpose is to allow names from a namespace to be used within the current scope without requiring a namespace access or a prefix.

It can be used to either allow a single name, or all the names from a namespace:

```
_Namespace
stdc::bits -= "stdc_",
stdc::mem  -= "mem",
stdc::str  -= "str"
{
    #include <stdbit.h>
    #include <string.h>
}

int main()
{
    _Using _Namespace stdc::bits; // introduces all names from stdc::bits
into this scope

    unsigned i = leading_zeros_ui(1);

    _Using stdc::mem::cmp; // introduces only stdc::mem::cmp

    int c = cmp("a", "b", 1); // calls stdc::mem::cmp (aka memcmp)

    _Using stdc::str::cmp;

    int c2 = cmp("a", "b"); // error, ambiguous
    // either use fully prefixed name, or access via namespace
}
```

Aliasing Namespaces

namespaces can be aliased in a scope with `_Namespace alias = existing_namespace;`:

```
_Namespace pthread -= "pthread_"
{
    #include <pthread.h>
}

void *proc(void *arg);

int main()
{
    _Namespace pt = pthread;

    pthread_t thread;

    pt::create(&thread, 0, proc, 0);

    pt::join(thread, 0);
}
```

Backward/Forward Compatibility

Since this proposal doesn't require any modification to existing code to become useful, nor does it break any old code, backwards compatibility isn't a concern for the most part.

Forward compatibility is also maintained:

```
_Namespace pt -= "pthread_"
{
    #include <pthread.h>
}

void *proc(void *arg);

int main()
{
    pthread_t thread;
    pt::create(&thread, 0, proc, 0);
    pt::join(thread, 0);
}
```

Even if pthread.h adds a namespace scope in a future version, this code will always work as intended. This is why capture-prefix scopes were designed such that they can never nest by scoping, and why file scope identifiers are considered for all outer capture-prefix scopes.

Compatibility with C++

Many libraries aim to have their header files be compatible with both C and C++. With this proposal, library header files can remain as-is, and the responsibility to namespace the library can be left to the user of the header.

Alternatively, if the library wants to convenience C2y users, it can use conditional compilation to only use namespaces if the language is C2y:

```
#define IS_C2Y (__STDC_VERSION__ >= 202ymmL) // whatever the number ends up
being

#if IS_C2Y
_Namespace
MyLib::Math -= "mylib_math_",
MyLib -= "mylib_"
{
#elif defined(__cplusplus)
extern "C" {
#endif

// declarations

#if IS_C2Y || defined(__cplusplus)
}
#endif
```

Convenience Header

A new header <stdnamespace.h> might be desired to define the macros:

```
#define namespace _Namespace
#define using _Using
```

Prior Art

Many other programming languages provide namespaces, or a similar feature that eliminates the need for users to prefix all file scope identifiers. The most similar prior art to this proposal is C++ namespaces.

The biggest difference between what this paper proposes and prior art is that this proposal entirely avoids implicit name mangling. It instead requires the user to specify a prefix. And there appears to be no prior art for converting a prefixed API into a namespace.

Limitations

The biggest limitation of this proposal is that macros can't get namespaced (since namespaces are processed after all macros have been expanded). Another limitation is that it doesn't help when working with libraries that already pollute the global namespace [0] [1] [2]. It only helps make using prefixed libraries easier, and helps make prefixing new libraries easier.

A future draft may include an extension to the apply-prefix scope to specify that it should only apply the prefix to non-external linkage file scope identifiers. That would help with libraries that declare file scope identifiers with internal linkage or no linkage without a proper prefix:

```
_Namespace(no_extern) raylib += "RL_"
{
    #include <raylib.h>
}
```

External linkage identifiers in the scope would still enter the namespace, they just don't get prefixed.

References

[0]: https://www.reddit.com/r/cpp_questions/comments/lxjhgg/how_do_you_deal_with_c_libraries_which_pollute/
[1]: https://github.com/raysan5/raylib/issues/1217
[2]: https://github.com/json-c/json-c/issues/621