Proposal for C2y/C3a
WG14

Document Number: N3848
Author: Abdulmalek Almkainzi <aalmkainzi@gmail.com>
Title: Namespacing with prefixes r1
Proposal category: New Features
Target Audience: General Developers

Abstract:
Prefixing identifiers has been the de facto standard method C developers avoid name collisions in their APIs. This paper explores a possible solution to add namespaces to C without implicit name mangling by utilizing prefixes.

Table of Contents

Changes

r0:
- Initial draft.
r1:
- Changed keyword to `_Nameprefix`.
- A namespace can only be associated with one prefix.
- Added `_Global` namespace, replacing C++ style `::` without a name on the left side.
- `_Apply` and `_Capture` keywords instead of `+=` and `-=` syntax.
- Removed `_Using`.
- Added my fork of slimcc [3] as prior art.

Introduction and Rationale

C library developers should prefix their entire exposed API to prevent name collisions. 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]).

The benefits of a possible namespaces feature in C are:

- Make creating a prefixed API more convenient.
- Make using prefixed APIs more convenient.
- Make APIs organizable by names.

The goal isn't to fix existing libraries that already pollute the global namespace with unprefixed identifiers, but to make it less likely to happen with new code.

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.

Example of Usage

For library developers:

C23	this proposal

`#include "lib_math.h"`
 	
`LIBMATH__vec3 LIBMATH__fmulv3(float a, LIBMATH__vec3 b)`
`{`
`    return (LIBMATH__vec3){`
`        b.x * a,`
`        b.y * a,`
`        b.z * a`
`    };`
`}`
 	
`LIBMATH__vec3`
`LIBMATH__v3add(LIBMATH__vec3 a, LIBMATH__vec3 b)`
`{`
`    return (LIBMATH__vec3){`
`        a.x + b.x,`
`        a.y + b.y,`
`        a.z + b.z`
`    };`
`}`
	`#include "lib_math.h"`
	
	`_Apply _Nameprefix Lib::Math`
	`{`
	`    vec3 fmulv3(float a, vec3 b)`
	`    {`
	`        return (vec3){`
	`            b.x * a,`
	`            b.y * a,`
	`            b.z * a`
	`        };`
	`    }`
	
	`    vec3 v3add(vec3 a, vec3 b)`
	`    {`
	`        return (vec3){`
	`            a.x + b.x,`
	`            a.y + b.y,`
	`            a.z + b.z`
	`        };`
	`    }`
	`}`

For library users:

C23	this proposal

`LIBMATH__vec3 vec = {0.1f, 0.2f, 0.3f};`
`vec = LIBMATH__fmulv3(5.0f, vec);`
`vec = LIBMATH__v3add(`
`    vec,`
`    (LIBMATH__vec3){1,2,3}`
`);`
	`_Nameprefix math = Lib::Math;`
	
	`math::vec3 vec = {0.1f, 0.2f, 0.3f};`
	`vec = math::fmulv3(5.0f, vec);`
	`vec = math::v3add(`
	`    vec,`
	`    (math::vec3){1,2,3}`
	`);`

Proposal

Identifiers that can enter a namespace are file scope identifiers. These include ordinary identifiers and tags.

Namespaces must be declared before opening scopes for them. Namespace scopes can only be opened at file scope, and file scope identifiers declared in the namespace scope are still considered to be file scope.

A namespace declaration looks like this:

```
_Nameprefix mylib = "MyLib_";
```

After a namespace has been declared, subsequent declarations of it are allowed as long as they are identical.

There are two ways to add entries to a namespace. The first adds file scope identifiers declared in a scope to a namespace (if it isn't already added) and prefixes the identifiers with the namespace's prefix. This will be referred to as an **apply-prefix** namespace scope:

```
_Nameprefix mylib = "MyLib_";

_Apply _Nameprefix mylib
{
    struct S {
        char c;
    };

    int count;

    struct S foo()
    {
        return (struct S){};
    }
}

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

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

It’s a constraint violation to declare a namespace inside an apply-prefix scope:

```
_Nameprefix A = "A_";

_Apply _Nameprefix A
{
    _Nameprefix B = "B_"; // error
}
```

declared identifiers cannot contain a namespace access `::`.

```
_Nameprefix A = "A_";
_Nameprefix B = "B_";

_Apply _Nameprefix B
{
    struct X { char c; };
    typedef float fl;
}

B::fl f; // ok
struct B::X x; // ok
struct B::X { char c; } x2; // error
```

If the prefixed declarations collide, a constraint violation occurs:

```
_Nameprefix A = "h_";
_Nameprefix B = "h";

_Apply _Nameprefix A
{
    int foo(); // becomes h_foo
    void bar(); // becomes h_bar
}

_Apply _Nameprefix B
{
    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.

In order to make existing prefixed libraries benefit from namespaces, this paper proposes another way to add identifiers to namespaces, such that prefixed identifiers declared in a scope enter a namespace with their prefixes omitted. The idea is to match file scope identifiers in the scope against a namespace’s prefix. This will be referred to as a **capture-prefix** namespace scope:

```
_Nameprefix sqlite3 = "sqlite3_";

_Capture _Nameprefix sqlite3
{
    #include <sqlite3.h>
}

int main()
{
    _Nameprefix sql = sqlite3; // alias

    sqlite3 *db;
    sql::open(":memory:", &db);
    sql::exec(db, "CREATE TABLE test(id INT)", 0,0,0);
    sql::close(db);
}
```

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.

When inside an apply-prefix scope, the names of the identifiers in the namespace can be used without a prefix and without a namespace access, including nested namespaces.

```
constexpr int i = 5;

_Nameprefix A = "A_";

_Apply _Nameprefix A
{
    int bar()
    {
        return i; // returns 5
    }

    constexpr int i = 10;

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

Essentially, in apply-prefix scopes, identifiers are resolved by searching from the names available in the innermost nested namespace, continuing outwards if no match found:

```
_Nameprefix A = "A__";
_Nameprefix A::B = "B__";

_Apply _Nameprefix A
{
    constexpr int i = 1;
}

_Apply _Nameprefix A::B
{
    int foo()
    {
        return i; // returns 1
    }

    constexpr int i = 2;

    int bar()
    {
        return i; // returns 2
    }
}
```

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 starts with the prefix.
- It is a valid identifier after the prefix.

```
_Nameprefix B = "b_";

_Capture _Nameprefix B
{
    int b_count;  // added to the namespace B

    int b_func()  // added to the namespace B
    { return 0; }
    int bar()     // not added to the namespace B
    { return 0; }
    int b_()      // not added to the namespace B
    { return 0; }
    int b_1()     // not added to the namespace B
    { return 0; }
}


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

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

```
_Nameprefix A = "A_";

_Apply _Nameprefix A
{
    int i;
}

_Capture _Nameprefix A
{
    int A_foo()
    {
        return i; // error, must use A::i, or A_i
    }
}
```

Basically, a capture-prefix scope acts like the outermost global scope. Because of that, declaring a namespace inside it is allowed:

```
_Nameprefix A = "A_";

_Capture _Nameprefix A
{
    _Nameprefix B = "B_"; // ok
}

// B now exists and can be used
_Apply _Nameprefix B
{
    void foo();
}
```

When a capture-prefix scope is opened inside an apply-prefix scope, file scope identifiers inside the capture-prefix scope are global, and not prefixed nor added to the apply-prefix scope’s namespace:

```
_Nameprefix A = "A_";
_Nameprefix B = "B_";

_Apply _Nameprefix A
{
    int i;
    _Capture _Nameprefix B
    {
        int B_i;
    }
}

int main()
{
    return A::i + B::i;
}
```

It is allowed to specify multiple namespaces when creating a capture-prefix scope:

```
_Nameprefix A = "A_";
_Nameprefix B = "B_";
_Nameprefix C = "C_";

_Capture _Nameprefix A, B, C
{
    int A_func(); // enters the namespace A
    int B_func(); // enters the namespace B
    int C_func(); // enters the namespace C
}
```

The purpose of this is to handle headers that apply different prefixes for different categories of their API. It's also useful for transitive includes, where a header includes another library, and the user wants to namespace both libraries.

When an identifier is encountered in a capture-prefix scope with multiple namespaces, it is only added to one of them. The process of determining which namespace an identifier maps to is sequential. A file scope identifier would be checked against the first namespace's prefix, if it doesn't match, it's checked against the one after it, and so on.

```
_Nameprefix A = "lib__";
_Nameprefix B = "lib_";
_Nameprefix C = "";

_Capture _Nameprefix A, B, C
{
    void baz();
    void lib__bar();
    void lib_foo();
    void lib_();
}

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

If an empty string is specified as the prefix, 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` before `A`, then `A` would be empty. A warning diagnostic could be raised when a namespace is guaranteed to never match with any identifier in a capture-prefix scope.

When a file scope identifier is declared inside a capture-prefix scope, it is considered for all capture-prefix scopes enclosing it:

```
_Nameprefix A  = "A_";
_Nameprefix A2 = "A__";
_Nameprefix A3 = "A___";

_Capture _Nameprefix A
{
    _Capture _Nameprefix A2
    {
        _Capture _Nameprefix A3
        {
            int A___foo() { return 0; }
        }
    }
}

int main()
{
    return A::__foo() + A2::_foo() + A3::foo();
}
```

Nesting

To declare a namespace as nested, it must be declared with a namespace access:

```
_Nameprefix sdl = "SDL_";
_Nameprefix sdl::gpu = "SDL_GPU";
```

After that, scopes for the nested namespace can be opened. When inside an apply-prefix namespace scope, the names of the namespaces nested under it can be used:

```
_Nameprefix A = "A_";
_Nameprefix A::B = "A_B_";

_Apply _Nameprefix A
{
    _Apply _Nameprefix B
    {
        void foo();
    }

    static inline void bar()
    {
        B::foo();
    }
}
```

Nested namespace scopes can also be opened like this:

```
_Nameprefix mylib = "MyLib_";
_Nameprefix mylib::math = "MyLib_Math_";

_Apply _Nameprefix mylib::math
{
    int div(int,int);
}
```

The _Global Namespace

`_Global` is a special pre-declared namespace. It has nested under it all outer-level namespaces, and it contains all names that can be used in the outer-most global scope without a namespace access:

```
int main()
{
    // any file scope identifier that can be used here without
    // a namespace access, is inside _Global
}
```

The purpose of it is to be able to use global names when inside an apply-prefix scope:

```
_Nameprefix A = "A_";

constexpr int i = 1;

_Apply _Nameprefix A
{
    int foo()
    {
        return i; // returns 1
    }

    constexpr int i = 2;

    int bar()
    {
        return i; // returns 2
    }

    int baz()
    {
        return _Global::i; // returns 1
    }
}

int main()
{
    // can also use _Global here
    return _Global::i + _Global::A::i + _Global::A_i;
}
```

It is similar to using `::` without a namespace name on the left side in C++.

Capture-prefix scopes can be opened for _Global.

```
_Nameprefix A = "A_";

_Apply _Nameprefix A
{
    int i;
    _Capture _Nameprefix _Global
    {
        int i;
    }
}
```

There is no real reason why it is called “_Global” and not “global”. Namespace names and aliases are in their own name space, so it won’t break backwards compatibility to name it “global”. I named it “_Global” just in case there is a library named “global”.

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, a capture-prefix scope with multiple mappings can be used:

```
_Nameprefix sdl = "SDL_";
_Nameprefix sdl::gpu = "SDL_GPU";

_Capture _Nameprefix sdl::gpu, 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.

Aliasing Namespaces

Namespaces can be aliased in a scope with `_Nameprefix alias = existing_namespace;`:

```
_Nameprefix posix = "posix_";
_Nameprefix posix::thread = "pthread_";

_Capture _Nameprefix posix::thread
{
    #include <pthread.h>
}

void *proc(void *arg);

int main()
{
    _Nameprefix pt = posix::thread;

    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:

```
_Nameprefix pt = "pthread_";
_Capture _Nameprefix pt
{
    #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 declaration and a capture-prefix scope inside the header in a future version, this code will still work as intended. This is why capture-prefix scopes were designed as-if global scope, and why file scope identifiers are considered for all enclosing 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)

#if IS_C2Y
_Nameprefix MyLib = "MYLIB_";
_Nameprefix MyLib::Math = "MYLIBMATH_";

_Capture _Nameprefix MyLib::Math, 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 nameprefix _Nameprefix
#define apply_nameprefix _Apply _Nameprefix
#define capture_nameprefix _Capture _Nameprefix
```

Prior Art

Many other programming languages provide namespaces, or a similar feature (e.g. modules, packages). The most similar existing language feature 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.

there doesn’t appear to be prior art for capturing prefixes into namespace. However, I have implemented this proposal in my fork of slimcc [3].

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].

Possible Extensions

Ideas I have not included in the proposal and I have not implemented in my slimcc fork, as I'm not sure if they are the desired approaches:

Restricted Namespaces

Jakub Łukasiewicz mentioned to me that he wants namespaces to be closed for addition. A possible solution for this is to specify all entries that could be added when declaring a namespace:

```
restrict _Nameprefix Lib(struct Ctx, Ctx, union Result, Math::) = "Lib__";
```

In the above example, the namespace `Lib` can only ever contain a tag named `Ctx`, an ordinary identifier named `Ctx`, a tag named `Result`, and a nested namespace named `Math`.

If a nested namespace is declared but was not listed in the restrict namespace declaration, a constraint violation occurs:

```
_Nameprefix Lib::Util = "LibUtil__"; // error
```

When opening scopes for a restricted namespace, it must be specified that it is restrict:

```
_Apply restrict _Nameprefix Lib // ok
{
}

_Apply _Nameprefix Lib // error
{
}
```

file scope identifiers declared inside a restrict namespace scope and aren't listed in the restrict namespace declaration do not enter the namespace, and do not get prefixed (in case of apply-prefix scope).

No-Extern Apply-Prefix Scopes

A possible solution to attempt to fix existing libraries that expose file scope identifiers that are not properly prefixed:

```
_Nameprefix raylib = "RL_";

_Apply _Nameprefix(no_extern) raylib
{
    #include <raylib.h>
}
```

External linkage identifiers in the scope would still enter the namespace, but won't get prefixed. This should be paired with restricted namespace to prevent applying the prefix to transitive includes.

Alternative Approaches

Why Not a Preprocessor-based Approach?

A preprocessor-based solution would add locals, parameters, and member names to the namespace, even though they can't be accessed at file scope, which would risk name collisions within the namespace. The current design avoids this. For example:

```
_Nameprefix A = "A__";

_Apply _Nameprefix A {
    struct S // added to A
    {
        int i; // not added
    };
}

_Capture _Nameprefix A {
    static inline int A__foo(int A__i /* not added */ )
    {
        int A__i2; // not added
        A__i2 = A__i + 1;
        return A__i2;
    }
}
```

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
[3]: https://github.com/aalmkainzi/slimcc