Back to the basics: Namespaces 101

Sandor Dargo

Meeting C++

7th November 2025, Berlin (Germany)

TWITTER @SandorDargo WEB sandordargo.com

Who Am I?

Sándor DARGÓ

- Senior Engineer at **Spotify**
- Enthusiastic blogger: sandordargo.com
- Curious oenophile
- Fortunate father of two

Why this talk?

Namespaces are a great tool for a cleaner organization

They are often overlooked in terms of usage and teaching

Want to share some best practices I wish I was shared

Agenda

Introduction to Namespaces

Evolution of Namespaces

Namespace Mechanics and Scope Rules

Best Practices for `using namespace`s

Introduction to Namespaces

Introduction to Namespaces

Namespace Evolution

Namespace Mechanics and Scope Rules

Best Practices

A tool for better code organization

Organizes code in logical units

Reduces the risk of naming conflicts

Improves modularity and readability

Supports scalable architecture

Real-World Problems Namespaces Solve

Limits name collision between different libraries

Helps managing large (internal) APIs

Helps creating clear boundaries for modules

Namespace Evolution

Namespaces were introduced by C++98

Already provided most features:

- Name collision prevention
- Logical grouping
- Scope control
- The `std` namespace
- The using directive and declaration

```
#include <iostream>
namespace lib {
int value = 42;
void printValue() {
  std::cout << "value: " << value << '\n';
}
}  // namespace lib

int main() {
  lib::printValue();
  using namespace lib; // using directive
  std::cout << "value: " << value << '\n';
  using std::cout; // using declaration
  cout << "Hello!" << '\n';
  return 0;
}
```

C++11: `inline` Namespaces

Symbols in an `inline` NS are as if they

were part of the enclosing scope

Makes versioning easier

The `inline` version can be directly used

Ideal for the latest version

```
namespace my_library {

namespace v1 {
void printMessage() { /* … */ }
}  // namespace v1

inline namespace v2 {
void printMessage() { /* … */ }
}  // namespace v2
}  // namespace my_library

int main() {
  my_library::v1::printMessage(); // v1
  my_library::v2::printMessage(); // v2
  my_library::printMessage(); // also v2!
}
```

C++17: Nested Namespace Openings

Becomes easier to open nested namespaces

Before:

```
namespace outer {
namespace inner {
class Widget;
}  // namespace inner
}  // namespace outer
```

After:

```
namespace outer::inner {
class Widget;
}  //  namespace outer::inner
```

C++20: Nested Inline Namespaces

Becomes easier to open nested **inline** namespaces

Before:

```
namespace outer::middle {
inline namespace inner {
class Widget;
}  // namespace inner
}  // namespace outer::middle
```

After:

```
namespace
outer::middle::inline inner {
class Widget;
} // namespace outer::middle…
```

Namespace Mechanics and Scope Rules

Introduction to Namespaces

Namespace Evolution

Namespace Mechanics and Scope Rules

Best Practices

Name lookup rules

Qualified name lookup

For symbols on the right hand side of `::`

First, lookup is performed on the left hand side

Happens in the explicitly named scope

Doesn’t go to enclosing scopes

Includes using directives in the named scope

```
#include <print>

namespace c {
int x = 3;
}  // namespace c

namespace a {
int x = 1;
namespace b {
using namespace c;
int x = 2;
}  // namespace b

}  // namespace a

int main() {
    std::println("{}", ::a::b::x);  // 2
}
```

Qualified name lookup

For symbols on the right hand side of `::`

First, lookup is performed on the left hand side

Happens in the explicitly named scope

Doesn’t go to enclosing scopes

Includes using directives in the named scope

```
#include <print>

namespace c {
int z = 13;
}  // namespace c

namespace a {

namespace b {
using namespace c;
}  // namespace b

}  // namespace a

int main() {
    std::println("{}", ::a::b::z);  // 13
}
```

Qualified name lookup

For symbols on the right hand side of `::`

First, lookup is performed on the left hand side

Happens in the explicitly named scope

Doesn’t go to enclosing scopes

Includes using directives in the named scope

```
#include <print>

namespace a {
int w = 66;

namespace b {
}  // namespace b

}  // namespace a

int main() {
    // ERROR below
    // outer namespace not considered
    std::println("{}", ::a::b::w);
}
```

Unqualified name lookup

For symbols not appearing on the right hand side of ::

```
int x = 2;
int func() {
    int x = 3;
    // UNL finds local x = 3
    int y = x;
    // QNL finds global x = 2
    int z = ::x;
    return y * z; // 3 * 2
}
```

Unqualified name lookup

For symbols not appearing on the right hand side of ::

Lookup goes outward to find a declaration:

Unqualified name lookup

For symbols not appearing on the right hand side of ::

Lookup goes outward to find a declaration:

Local scope(s): innermost to extern

```
int func() {
  int x = 3;
  int y;
  {
    // No x here; lookup continues
    // to the enclosing function scope
    y = x;
  }
  return y;
}
```

Unqualified name lookup

For symbols not appearing on the right hand side of ::

Lookup goes outward to find a declaration:

- Local scope(s): innermost to extern
- Class/struct scope

```
struct Widget {
  int func() {
    // No local x; lookup finds member x
    int y = x;
    return y;
  }
  int x = 3;
};
```

Unqualified name lookup

For symbols not appearing on the right hand side of ::

Lookup goes outward to find a declaration:

- Local scope(s): innermost to extern
- Class/struct scope
- Namespaces: innermost to extern

```
#include <print>

namespace a {

int n = 1;

namespace b {

int f() {
    return n;  // Finds a::n (outer namespace)
}

}  // namespace b
}  // namespace a
int main() {
    std::println("{}", a::b::f());  // 1
}
```

Unqualified name lookup

For symbols not appearing on the right hand side of ::

Lookup goes outward to find a declaration:

- Local scope(s): innermost to extern
- Class/struct scope
- Namespaces: innermost to extern
- Global scope

```
#include <print>

int n = 1;

namespace a {

namespace b {

int f() {
    return n;  // Finds n in the global scope
}

}  // namespace b
}  // namespace a
int main() {
    std::println("{}", a::b::f());  // 1
}
```

Unqualified name lookup

For symbols not appearing on the right hand side of ::

Lookup goes outward to find a declaration:

- Local scope(s): innermost to extern
- Class/struct scope
- Namespaces: innermost to extern
- With the same priority:
  - Global scope
  - Namespaces introduced via using directives

```
#include <print>

int x = 2;
namespace c {
int x = 8, z = 7;
}  // namespace c
namespace a {
namespace b {
int g() {
    using namespace c;
    return z;
}
int h() {
    // using namespace c;
    // If uncommented, ambiguity between ::x and c::x
    return x;
}
}  // namespace b
}  // namespace a
int main() {
    std::println("{}", a::b::g());  // 7
    std::println("{}", a::b::h());  // 2
}
```

Argument-Dependent Lookup

Extends unqualified name lookup

Only for functions calls

Makes function calls "just work"

Brings into scope the function arguments' namespaces

Especially helpful for operator overloading or generic code

```
namespace math {
struct Vec {};
void normalize(Vec) {}
bool operator==(Vec, Vec) {
 return true;
}
}  // namespace math

void foo() {
  math::Vec a, b;
  normalize(a);   // OK via ADL
  if (a == b) {}  // OK via ADL
}
```

Argument-Dependent Lookup

Extends unqualified name lookup

Only for functions calls

Makes function calls "just work"

Brings into scope the function arguments' namespaces

Especially helpful for operator overloading or generic code

```
namespace math {
struct Vec {};
struct Bar {
    Bar(Vec v): _v(v){}
    Vec _v;
};
void normalize(Vec) {}
bool operator==(Vec, Vec) {
 return true;
}
}  // namespace math

void foo() {
  math::Vec a, b;
  normalize(a);   // OK via ADL
  if (a == b) {}  // OK via ADL
  // ERROR: ADL does not apply to class names
  // Bar c(a);
}
```

Bringing namespaces into scope

Using directives: `using namespace ns;`

Brings all the symbols of a namespace into the current scope

It might decrease readability as who knows what comes from where

A double edged sword, as it pollutes the namespace

What is namespace pollution?

When you have too many identifiers in a namespace, causing conflicts and confusion

Namespace pollution is a problem

With name clashing you can get

- Compilation errors
- Worse, unexpected behaviour

Maintenance is difficult due to potentially even more naming conflicts

Readability suffers as you don’t see what comes from where

Using-declarations: `using ns::symbol;`

Instead of bringing in all symbols, it brings only the specified ones

- Classes
- Functions
- Variables
- Enums
- Even enumerators since C++20

Better than the directive, but it can still pollute

Namespaces can be aliased

Shortens long namespace names

Might improve readability

Doesn’t pollute directly

```
#include <print>

namespace a::b::c {
    int x = 42;
}

int main() {
    std::println("{}", a::b::c::x);
    namespace abc = a::b::c;
    std::println("{}", abc::x);
}
```

Namespaces can be aliased

Shortens long namespace names

Might improve readability

Doesn’t pollute directly

Yet, using them in headers are risky

Namespaces can be aliased

Shortens long namespace names

Might improve readability

Doesn’t pollute directly

Yet, using them in headers are risky

Best to use it in `.cpp` files to avoid

shadowing or redefinition

```
// lib.hpp
#pragma once
namespace mylib::utils {
    void do_something();
}

// Alias defined in header
namespace mu = mylib::utils;

// main.cpp
#include "lib.hpp"

// This conflicts with the alias!
namespace mu {
    void other_thing();
}

int main() {
    // Which 'mu' is this?
    mu::do_something();
}
```

Impacts on readability and maintainability

Using directives and declarations pollute namespaces

Using directives *might* decrease readability

Literals are still good!

Using declarations are better for readability

Aliases can improve readability

Anonymous Namespaces

Not every namespace has a name

Limits the scope to internal linkage

Makes symbols private to a translation unit

Helps avoiding clashes

Preferred to static functions

The right place for implementation details

```
namespace {
std::string millisecondsToString(
    std::chrono::milliseconds ms) {
  return std::to_string(ms.count());
}
}  // namespace

namespace rns {
void Widget::update() {
  print(millisecondsToString(time))
}
}  // namespace rns
```

Best Practices for `using` namespaces

Introduction to Namespaces

Namespace Evolution

Namespace Mechanics and Scope Rules

Best Practices

What is covered in coding standards and style guides?

Core C++ Guidelines

C.5: Place helper functions in the same namespace as the class they support

C.168: Define overloaded operators in the namespace of their operands

T.47: Avoid highly visible unconstrained templates with common names?

SF.6: Use using namespace directives for transition, for foundation libraries (such as std), or within a local scope (only)

SF.7: Don't write using namespace at global scope in a header file

SF.20: Use namespaces to express logical structure

SF.21: Don't use an unnamed (anonymous) namespace in a header

SF.22: Use an unnamed (anonymous) namespace for all internal/non-exported entities

SL.3: Do not add non-standard entities to namespace std

C.5: Place helper functions in the same namespace as the class they support

Functions are seen as part of the class interface

Makes the relationship obvious

Allows benefiting from ADL

```
namespace Chrono {
// here we keep time-related
services

class Time { /* ... */ };
class Date { /* ... */ };

// helper functions:
bool operator==(Date, Date);
Date next_weekday(Date);
// ...
}
```

C.168: Define overloaded operators in the namespace of their operands

Allows benefiting from ADL

Increases readability

No inconsistent definitions in different namespaces

```
namespace N {
struct S { };

// OK: in the same namespace as S
// and even next to S
S operator+(S, S);
}  // namespace N

N::S s;
// finds N::operator+() by ADL
S r = s + s;
```

T.47: Avoid highly visible unconstrained templates with common names

Such template arguments match anything

Annoying and even dangerous when ADL is used

“The committee cannot agree to exclude unconstrained templates from ADL”

```
namespace Bad {
struct S { int m; };

template<typename T1, typename T2>
bool operator==(T1, T2) {
  cout << "Bad\n"; return true;
}
}  // namespace Bad

namespace T0 {
bool operator==(int, Bad::S) {
  cout << "T0\n"; return true;
 }  // compare to int
void test() {
  Bad::S bad{ 1 };
  vector<int> v(10);
  bool b = 1 == bad;  // T0
  bool b2 = v.size() == bad; // Bad
}
}  // namespace T0
```

SF.6: Use using namespace directives for transition, for foundation libraries, or within a local scope

`using namespace` can lead to name clashes

Even `std` can lead to name clashes

But sometimes consistent qualification is verbose and distracting

```
#include <cmath>
using namespace std;

int g(int x) {
    int sqrt = 7;
    // …
    return sqrt(x); // error
}
```

SF.7: Don’t write using namespace at global scope in a header file

Pollutes global namespace

Might introduce order dependency

Exception is `using namespace std::literals`

```
// bad.h
#include <iostream>
using namespace std; // bad

// user.cpp
#include "bad.h"
// some function that happens to be named copy
bool copy(/*... some parameters ...*/);

int main() {
    // now overloads local ::copy
    // and std::copy, could be ambiguous
    copy(/*...*/);
}
```

SF.20: Use namespaces to express logical structure

SF.20: Use `namespace`s to express logical structure

Reason ???

Example

```
???
```

Enforcement ???

SF.20: Use namespaces to express logical structure

The Guidelines have nothing to say

Maybe it’s considered too obvious

Namespaces are not only for avoiding name clashes

But to represent logical and/or domain structure

And improve readability and modularity

Some practical advice will come later

SF.21: Don’t use unnamed namespaces in headers

Anonymous namespaces give internal linkage

Each cpp file including such a header gets its own version of the “contents”

- Code bloat
- ODR violations
- Possible bugs with static data

Instead

- Use normal namespaces
- Use inlining to avoid ODR violation
- Use detail namespaces

```
// file foo.h:
namespace { // bad
    const double x = 1.234;
    const double y = 2.345;
    double foo(double y) {
        return y + x;
    }
}  // namespace
namespace foo { //good
    const double x = 1.234;
    namespace detail {
        const double y = 2.345;
    }  // detail
    inline double foo(double y){
        return y + x;
    }
} // namespace foo
```

SF.22: Use an unnamed (anonymous) namespace for all internal/non-exported entities

Nothing external can use symbols in an unnamed namespace

Unless a definition is exported, put it in an anonymous namespace

```
// Instead of

static int f();
int g();
static bool h();
int k();

// Prefer using

namespace {
    int f();
    bool h();
}  // namespace
int g();
int k();
```

SL.3: Do not add entities to namespace std

Adding to `std` might change the meaning of otherwise standards conforming code.

Additions to `std` might clash with future versions of the standard.

It is undefined behavior to add declarations or definitions to namespace `std`

Except for some template specializations

```
namespace std {
// UB: not allowed
void my_utility() { /* ... */ }

// UB!
bool operator==(const vector<int>&,
const vector<int>&);

}  // namespace std
```

SL.3: Do not add entities to namespace std (cont)

Except for some template specializations

- If the declaration depends on at least one program-defined type
- And the specialization satisfies all requirements for the original template

```
template<>
struct std::hash<MyType> {
  std::size_t operator()(
         const MyType& t) const {
      return t.hash();
  }
};
```

Google C++ Style Guide

✅ Use **namespaces** generously

❌ *using directives* are forbidden, use namespace aliases

✅ *using declarations* are fine

❌ *inline namespaces* are forbidden (allowed only for versioning)

❌ No anonymous namespaces in header files

Google C++ Style Guide

Namespace names should use `snake_case`

Top-level namespaces must be globally unique and recognizable

Nested namespaces should avoid well-known top-level namespaces

The contents of namespaces are not indented

Use closing comments, for anonymous namespaces leave the name empty

Google C++ Style Guide - Namespace Formatting

The contents of namespaces are not indented.

```
namespace {

  // Wrong!
  // Indented when it should not be.
  void foo() {
     // …
  }

}  // namespace
```

```
namespace {

// Correct.
// No extra indentation within namespace.
void foo() {
  ...
}

}  // namespace
```

Personal and organizational experience

❌No using directives and limit using declarations to `.cpp` files

(✅ *using declarations* are fine in `.cpp` files)

❌Avoid namespace aliases, especially in header files

❌No more than 4 level nested namespaces (the top one is all the same)

⚠️Differentiate between impl, detail, potentially detail or api

⚠️Always fully qualify in macros

⚠️Opening up a foreign namespace is often a code smell (it can happen sometimes)

Limit using declarations to `.cpp` files

Using directives are forbidden in header files

They are better to be avoided in `.cpp` files as they worsen readability

Use using declarations as much as you need if they improve readability

Don’t use them in the header files to avoid namespace pollution

No more than 4 level nested namespaces

Too deeply nested namespaces can lead to name clashes due UNL

- The risk can be eliminated if only leaf NSs are used for symbols
- And intermediary NSs are used for code organization

Too deep NSs are also too verbose and hard to read

Usually there are three levels

There might be an extra level

Always fully qualify in macros

First of all, don’t use macros! 😉

Macros are never part of namespaces

They are expanded before compilation

The preprocessor knows nothing about namespaces

Always fully qualify symbols in macros to avoid lookup issues

Opening up a foreign namespace is often a code smell

Each usage is a sign of dependency

Some are essential...

Too many should probably make you think

Industry best practices are less permissive then Core Guidelines (SF6, SF7)

Core guidelines protects mostly the global namespace

Most coding guidelines protect user defined namespaces as well

Time to conclude!

Key Takeaways

Namespaces prevent name collisions and organize code

Avoid polluting the namespaces

Prefer fine-grained and nested namespaces over large monolithic ones

Keep naming consistent and meaningful

A Quick Namespace “Do & Don’t” Checklist

✅ Do

Use namespaces to group related code

Use `namespace alias` when helpful

Scope `using` declarations appropriately

Might use `inline namespace` for versioning

❌ Don't

Dump everything into `namespace util`

Use `using namespace std;` in headers

Assume readers know where symbols come from

Over-nest deeply without reason

Back to the basics: Namespaces 101

Sandor Dargo

Meeting C++

7th November 2025, Berlin (Germany)

TWITTER @SandorDargo WEB sandordargo.com