# 6.17 — Unnamed and inline namespaces

BY ALEX ON JANUARY 3RD, 2020 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 8TH, 2020

C++ supports two permutations on regular namespaces that are worth at least knowing about. We won't build on these, so consider this lesson optional for now.

## Unnamed (anonymous) namespaces

An **unnamed namespace** (also called an **anonymous namespace**) is a namespace that is defined without a name, like so:

```
#include <iostream>

namespace // unnamed namespace
{
    void doSomething() // can only be accessed in this file
    {
        std::cout << "v1\n";
    }
}

int main()
{
    doSomething(); // we can call doSomething() without a namespace prefix

    return 0;
}
```

This prints:

v1

All content declared in an unnamed `namespace` is treated as if it is part of the parent namespace. So even though function `doSomething` is defined in the unnamed `namespace`, the function itself is accessible from the parent namespace (which in this case is the `global namespace`), which is why we can call `doSomething` from `main` without any qualifiers.

This might make unnamed `namespaces` seems useless. But the other effect of unnamed `namespaces` is that all identifiers inside an unnamed `namespace` are treated as if they had `internal linkage`, which means that the content of an unnamed `namespace` can't be seen outside of the file in which the unnamed `namespace` is defined.

For functions, this is effectively the same as defining all functions in the unnamed `namespace` as `static functions`. The following program is effectively identical to the one above:

```
#include <iostream>

static void doSomething() // can only be accessed in this file
{
    std::cout << "v1\n";
}

int main()
{
    doSomething(); // we can call doSomething() without a namespace prefix

```

```
    return 0;
}
```

Unnamed namespaces are typically used when you have a lot of content that you want to ensure stays local to a given file, as it's easier to cluster such content in an unnamed namespace than individually mark all declarations as static. Unnamed namespaces will also keep user-defined types (something we'll discuss in a later lesson) local to the file, something for which there is no alternative equivalent mechanism to do.

---

## Inline namespaces

Now consider the following program:

```
#include <iostream>

void doSomething()
{
    std::cout << "v1\n";
}

int main()
{
    doSomething();

    return 0;
}
```

This prints:

v1

Pretty straightforward, right?

But let's say you're not happy with doSomething, and you want to improve it in some way that changes how it behaves. But if you do this, you risk breaking existing programs using the older version. How do you handle this?

One way would be to create a new version of the function with a different name. But over the course of many changes, you could end up with a whole set of almost-identically named functions (doSomething, doSomething_v2, doSomething_v3, etc...).

An alternative is to use an inline namespace. An **inline namespace** is a namespace that is typically used to version content. Much like an unnamed namespace, anything declared inside an inline namespace is considered part of the parent namespace. However, inline namespaces don't give everything internal linkage.

To define an inline namespace, we use the inline keyword:

```
#include <iostream>

inline namespace v1 // declare an inline namespace named v1
{
    void doSomething()
    {
        std::cout << "v1\n";
    }
}

namespace v2 // declare an normal namespace named v2
{
    void doSomething()
    {
```

```
        std::cout << "v2\n";
    }
}

int main()
{
    v1::doSomething(); // calls the v1 version of doSomething()
    v2::doSomething(); // calls the v2 version of doSomething()

    doSomething(); // calls the inline version of doSomething() (which is v1)

    return 0;
}
```

This prints:

```
v1
v2
v1
```

In the above example, callers to doSomething will get the v1 (the inline version) of doSomething. Callers who want to use the newer version can explicitly call `v2::dosomething()`. This preserves the function of existing programs while allowing newer programs to take advantage of newer/better variations.

Alternatively, if you want to push the newer version:

```
#include <iostream>

namespace v1 // declare an normal namespace named v1
{
    void doSomething()
    {
        std::cout << "v1\n";
    }
}

inline namespace v2 // declare an inline namespace named v2
{
    void doSomething()
    {
        std::cout << "v2\n";
    }
}

int main()
{
    v1::doSomething(); // calls the v1 version of doSomething()
    v2::doSomething(); // calls the v2 version of doSomething()

    doSomething(); // calls the inline version of doSomething() (which is v2)

    return 0;
}
```

This prints:

```
v1
v2
v2
```

In this example, all callers to `doSomething` will get the v2 version by default (the newer and better version). Users who still want the older version of `doSomething` can explicitly call `v1::doSomething()` to access the old behavior. This means existing programs who want the v1 version will need to globally replace `doSomething` with `v1::doSomething`, but this typically won't be problematic if the functions are well named.

**6.x -- Chapter 6 summary and quiz**

**Index**

**6.16 -- Explicit type conversion (casting) and static_cast**

C++ TUTORIAL | PRINT THIS POST

## 10 comments to 6.17 — Unnamed and inline namespaces

fighter_fish
January 29, 2020 at 8:47 pm · Reply

I only noticed now, but congrats to nascardriver for being promoted. Great job in editing the article!

nascardriver
January 30, 2020 at 2:33 am · Reply

Thanks :)

Suyash
January 15, 2020 at 6:36 pm · Reply

Thanks for going through these special types of namespaces... Namespaces are one of the things that I find unique to C++, so it's always good to learn more about it...

While I do find some use cases of anonymous namespaces, inline namespaces don't interest me that much because the main purpose of their existence is to provide some primitive form of versioning (which while useful) will never be used by any actual developer due to the presence of Version Control Systems like Git...

nascardriver
January 16, 2020 at 2:23 am · Reply

You're mixing up versioning of source code (git) with versioning of a library. Take the standard library for example. There's only 1 library, but it serves all versions of the standard (C++98 through C++2a). There's a neat **example, including comments**, on stackoverflow. It's going to be hard to understand the example, you might want to revisit it later.

Loïc
January 10, 2020 at 3:13 pm · Reply

If unnamed namespaces have internal linkage, why does this code doesn't compile :

```
1  namespace a
2  {
3      namespace
4      {
5          int test()
6          {
7              return 5;
8          }
9      }
10 }
11
12 int main()
13 {
14     std::cout << test() << std::endl;
15     return 0;
16 }
```

nascardriver
January 12, 2020 at 3:26 am · Reply

Linkage has nothing to do with whether or not you can access something that's in the same file. Linkage defines if you can access something that's in a different file.
`a::test()` is inaccessible in files other than the file it's defined in.
You still have to use the `a::` prefix to access it in `main`.

Fan
January 9, 2020 at 5:36 am · Reply

What is the difference between putting something in an inline namespace vs. in the global scope?

nascardriver
January 9, 2020 at 5:48 am · Reply

Name collisions are easy to solve with inline namespaces:

```
inline namespace FansLibrary
{
  void fn(){}
}

int main()
{
  fn(); // ok
}
```

Now you include something else that also has an inline namespace with the same function:

```
inline namespace NascardriversLibrary
{
  void fn(){}
}

inline namespace FansLibrary
{
  void fn(){}
}

int main()
{
  fn(); // oops, now what?

  FansLibrary::fn(); // Ok
}
```

If both library authors put the functions in the global namespace, you'd have to edit the libraries or do some dirty workarounds.

hausevult
January 6, 2020 at 11:31 am · Reply

> Unnamed namespaces will also keep user-defined type definitions (something we'll discuss in a later lesson) local to the file, something for which there is no alternative equivalent mechanism to do.

Type definitions were discussed in a previous lesson, rather than a later one.

nascardriver
January 8, 2020 at 5:09 am · Reply

User-defined types (`struct`, `class`) are covered later. Either lessons were moved around and you read the lesson on `struct` already, or you're thinking of something else.