7.13 — Unnamed and inline namespaces

👤 **ALEX**   🕐 **OCTOBER 24, 2023**

C++ supports two variants of 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:

```cpp
#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 seem useless. But the other effect of unnamed namespaces is that all identifiers inside an unnamed namespace are treated as if they have 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 a single unnamed namespace than individually mark all declarations as `static`. Unnamed namespaces will also keep program-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, unlike unnamed namespaces, inline namespaces don't affect 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 a 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 a 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.

Mixing inline and unnamed namespaces `Optional`

A namespace can be both inline and unnamed:

```cpp
#include <iostream>

namespace V1 // declare a normal namespace named V1
{
    void doSomething()
    {
        std::cout << "V1\n";
    }
}

inline namespace // declare an inline unnamed namespace
{
    void doSomething() // has internal linkage
    {
        std::cout << "V2\n";
    }
}

int main()
{
    V1::doSomething(); // calls the V1 version of doSomething()
    // there is no V2 in this example, so we can't use V2:: as a namespace prefix

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

    return 0;
}
```

However, in such cases, it's probably better to nest an anonymous namespace inside an inline namespace. This has the same effect (all functions inside the anonymous namespace have internal linkage by default) but still gives you an explicit namespace name you can use:

```
#include <iostream>

namespace V1 // declare a normal namespace named V1
{
    void doSomething()
    {
        std::cout << "V1\n";
    }
}

inline namespace V2 // declare an inline namespace named V2
{
    namespace // unnamed namespace
    {
        void doSomething() // has internal linkage
        {
            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;
}
```

• • •

Leave a comment...

Name*

Notify me about replies: 🔔 POST COMMENT

@ Email* ?

Find a mistake? Leave a comment above!?

Avatars from https://gravatar.com/ are connected to your

provided email address.

Newest ▾

We and our partners share information on your use of this website to help improve your experience.

Do not sell my info:

OKAY

✕