6.13 — Unnamed and inline namespaces

ALEX JUNE 18, 2021

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:

```
#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 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 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.

Next lesson
6.x Chapter 6 summary and quiz

Back to table of contents

Previous lesson
6.12 Using declarations and using directives

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

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

Notify me about replies: POST COMMENT

51 COMMENTS

Newest