4.3b — Namespaces

BY ALEX ON AUGUST 17TH, 2007 | LAST MODIFIED BY ALEX ON FEBRUARY 27TH, 2018

In lesson **1.8a -- Naming conflicts and the std namespace**, we introduced the concept of naming conflicts and namespaces. This lesson builds upon those topics.

A naming conflict occurs when two identifiers are introduced into the same scope, and the compiler can't disambiguate which one to use. When this happens, compiler or linker will produce an error because it does not have enough information to resolve the ambiguity. As programs get larger and larger, the number of identifiers increases linearly, which in turn causes the probability of naming collisions to increase exponentially.

Let's take a look at an example of a naming collision. In the following example, foo.h and goo.h are the header files that contain functions that do different things but have the same name and parameters.

foo.h:

```
// This doSomething() adds the value of its parameters
int doSomething(int x, int y)
{
    return x + y;
}
```

goo.h:

```
// This doSomething() subtracts the value of its parameters
int doSomething(int x, int y)
{
    return x - y;
}
```

main.cpp:

```
#include "foo.h"
#include "goo.h"
#include <iostream>

int main()
{
    std::cout << doSomething(4, 3); // which doSomething will we get?
    return 0;
}
```

If foo.h and goo.h are compiled separately, they will each compile without incident. However, by including them in the same program, we have now introduced two different functions with the same name and parameters into the same scope (the global scope), which causes a naming collision. As a result, the compiler will issue an error:

```
c:\VCProjects\goo.h(4) : error C2084: function 'int __cdecl doSomething(int,int)' already has a bo
```

In order to help address this type of problem, the concept of namespaces was introduced.

What is a namespace?

A namespace defines an area of code in which all identifiers are guaranteed to be unique. By default, global variables and normal functions are defined in the **global namespace**. For example, take a look at the following snippet:

```
int g_x = 5;

int foo(int x)
{
    return -x;
}
```

Both global variable g_x and function foo() are defined in the global namespace.

In the example program above that had the naming collision, when main() #included both foo.h and goo.h, both versions of doSomething() were included into the global namespace, which is why the naming collision resulted.

In order to help avoid issues where two independent pieces of code have naming collisions with each other when used together, C++ allows us to declare our own namespaces via the **namespace** keyword. Anything declared inside a user-defined namespace belongs to that namespace, not the global namespace.

Here is an example of the headers in the first example rewritten using namespaces:

foo.h:

```
namespace Foo
{
    // This doSomething() belongs to namespace Foo
    int doSomething(int x, int y)
    {
        return x + y;
    }
}
```

goo.h:

```
namespace Goo
{
    // This doSomething() belongs to namespace Goo
    int doSomething(int x, int y)
    {
        return x - y;
    }
}
```

Now the doSomething() inside of foo.h is inside the Foo namespace, and the doSomething() inside of goo.h is inside the Goo namespace. Let's see what happens when we recompile main.cpp:

```
int main()
{
    std::cout << doSomething(4, 3); // which doSomething will we get?
    return 0;
}
```

The answer is that we now get another error!

```
C:\VCProjects\Test.cpp(15) : error C2065: 'doSomething' : undeclared identifier
```

What happened is that when we tried to call the doSomething() function, the compiler looked in the global namespace to see if it could find a definition of doSomething(). However, because neither of our doSomething() functions live in the global namespace any more, it failed to find a definition at all!

There are two different ways to tell the compiler which version of doSomething to use, via the scope resolution operator, or via using statements (which we'll discuss in the next lesson).

Accessing a namespace with the scope resolution operator (::)

The first way to tell the compiler to look in a particular namespace for an identifier is to use the scope resolution operator (::). This operator allows you to prefix an identifier name with the namespace you wish to use.

Here is an example of using the scope resolution operator to tell the compiler that we explicitly want to use the version of doSomething that lives in the Foo namespace:

```
int main(void)
{
    std::cout << Foo::doSomething(4, 3);
    return 0;
}
```

This produces the result:

```
7
```

If we wanted to use the version of doSomething() that lives in Goo instead:

```
int main(void)
{
    std::cout << Goo::doSomething(4, 3);
    return 0;
}
```

This produces the result:

```
1
```

The scope resolution operator is very nice because it allows us to specifically pick which namespace we want to look in. It even allows us to do the following:

```
int main(void)
{
    std::cout << Foo::doSomething(4, 3) << '\n';
    std::cout << Goo::doSomething(4, 3) << '\n';
    return 0;
}
```

This produces the result:

```
7
1
```

It is also possible to use the scope resolution operator without any namespace (eg. `::doSomething`). In that case, it refers to the global namespace.

Multiple namespace blocks with the same name allowed

It's legal to declare namespace blocks in multiple locations (either across multiple files, or multiple places within the same file). All declarations within the namespace block are considered part of the namespace.

add.h:

```
namespace BasicMath
{
    // function add() is part of namespace BasicMath
    int add(int x, int y)
    {
        return x + y;
    }
}
```

subtract.h:

```
namespace BasicMath
{
    // function subtract() is also part of namespace BasicMath
    int subtract(int x, int y)
    {
        return x - y;
    }
}
```

main.cpp:

```
#include "add.h" // import BasicMath::add()
```

```
#include "subtract.h" // import BasicMath::subtract()

int main(void)
{
    std::cout << BasicMath::add(4, 3) << '\n';
    std::cout << BasicMath::subtract(4, 3) << '\n';

    return 0;
}
```

This works exactly as you would expect.

The standard library makes extensive use of this feature, as all of the different header files included with the standard library have their functionality inside namespace std.

Nested namespaces and namespace aliases

Namespaces can be nested inside other namespaces. For example:

```
#include <iostream>

namespace Foo
{
    namespace Goo // Goo is a namespace inside the Foo namespace
    {
        const int g_x = 5;
    }
}

int main()
{
    std::cout << Foo::Goo::g_x;
    return 0;
}
```

Note that because namespace Goo is inside of namespace Foo, we access g_x as Foo::Goo::g_x.

In C++17, nested namespaces can also be declared this way:

```
#include <iostream>

namespace Foo::Goo // Goo is a namespace inside the Foo namespace (C++17 style)
{
    const int g_x = 5;
}

int main()
{
    std::cout << Foo::Goo::g_x;
    return 0;
}
```

Because typing the fully qualified name of a variable or function inside a nested namespace can be painful, C++ allows you to create namespace aliases.

```
namespace Foo
{
    namespace Goo
    {
        const int g_x = 5;
    }
}

namespace Boo = Foo::Goo; // Boo now refers to Foo::Goo

int main()
{
```

```
13      std::cout << Boo::g_x; // This is really Foo::Goo::g_x
14      return 0;
15  }
```

It's worth noting that namespaces in C++ were not designed as a way to implement an information hierarchy -- they were designed primarily as a mechanism for preventing naming collisions. As evidence of this, note that the entirety of the standard template library lives under the singular namespace std::. Some newer languages (such as C#) differ from C++ in this regard.

In general, you should avoid nesting namespaces if possible, and there are few good reasons to nest them more than 2 levels deep. However, in later lessons, we will see other related cases where the scope resolution operator needs to be used more than once.


4.3c -- Using statements


Index


4.3a -- Scope, duration, and linkage summary

161 comments to 4.3b — Namespaces

« Older Comments 1 2 3

Jesper Nielsen
April 12, 2018 at 10:45 am · Reply

As a long time C# and before that Java programmer I find it hard to understand namespaces (and folders) are not used a lot more.
But on the other hands Visual Studio really starts fighting you if you want to arrange your code...

Matt
February 25, 2018 at 5:26 pm · Reply

What would be considered a rule of thumb for "best practice" when it comes to namespaces, multiple files, etc? To clarify my question (and to highlight why I am starting to get nervous at this point in the tutorial series): Every time I write a program from this point forward, should I be separating my individual functions into their own .cpp files, should I be giving all of my functions their own namespace, making ample use of .header files, using global const variables as often as possible, et cetera in the name of practicing the concepts? Or, in contrast, should a 'minimalist' approach be used, using as few files as necessary for disambiguation, convenience, and clarity?

Clearly, C++ has a ton of functionality and there are several ways to accomplish the same goal. In the last chapter or so worth of lessons we have introduced a ton of tools/concepts that seem, if you'll excuse my saying so, almost unnecessary (considering what they offer can be accomplished without them using what we've learned prior).

Perhaps I'm writing this because I'm feeling a bit overwhelmed and instead of learning simple exercises in syntax we're being familiarized with a myriad of options/tools - thanks for your time!