# 6.2 — User-defined namespaces and the scope resolution operator

ALEX JULY 4, 2021

In lesson 2.8 -- Naming collisions and an introduction to namespaces, we introduced the concept of `naming collisions` and `namespaces`. As a reminder, a naming collision occurs when two identical 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 they do not have enough information to resolve the ambiguity. As programs become larger, the number of identifiers increases linearly, which in turn causes the probability of a naming collision occurring to increase exponentially.

Let's revisit an example of a naming collision, and then show how we can resolve it using namespaces. In the following example, `foo.cpp` and `goo.cpp` are the source files that contain functions that do different things but have the same name and parameters.

foo.cpp:

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

goo.cpp:

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

main.cpp:

```cpp
#include <iostream>

int doSomething(int x, int y); // forward declaration for doSomething

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

If this project contains only `foo.cpp` *or* `goo.cpp` (but not both), it will compile and run without incident. However, by compiling both into 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 linker will issue an error:

```
goo.cpp:3: multiple definition of `doSomething(int, int)'; foo.cpp:3: first defined here
```

Note that this error happens at the point of redefinition, so it doesn't matter whether function `doSomething` is ever called.

One way to resolve this would be to rename one of the functions, so the names no longer collide. But this would also require changing the names of all the function calls, which can be a pain, and is subject to error. A better way to avoid collisions is to put your functions into your own namespaces. For this reason the standard library was moved into the `std` namespace.

## Defining your own namespaces

C++ allows us to define our own namespaces via the `namespace` keyword. Namespaces that you create for your own declarations are called user-defined namespaces. Namespaces provided by C++ (such as the `global namespace`) or by libraries (such as `namespace std`) are not considered user-defined namespaces.

Namespace identifiers are typically non-capitalized.

Here is an example of the files in the prior example rewritten using namespaces:

foo.cpp:

```
namespace foo // define a namespace named foo
{
    // This doSomething() belongs to namespace foo
    int doSomething(int x, int y)
    {
        return x + y;
    }
}
```

goo.cpp:

```
namespace goo // define a namespace named goo
{
    // This doSomething() belongs to namespace goo
    int doSomething(int x, int y)
    {
        return x - y;
    }
}
```

Now `doSomething()` inside of `foo.cpp` is inside the `foo` namespace, and the `doSomething()` inside of `goo.cpp` is inside the `goo` namespace. Let's see what happens when we recompile our program.

main.cpp:

```
int doSomething(int x, int y); // forward declaration for doSomething

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

The answer is that we now get another error!

```
ConsoleApplication1.obj : error LNK2019: unresolved external symbol "int __cdecl doSomething(int,int)" (?doS
omething@@YAHHH@Z) referenced in function _main
```

In this case, the compiler was satisfied (by our forward declaration), but the linker could not find a definition for `doSomething` in the global namespace. This is because both of our versions of `doSomething` are no longer in the global namespace!

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 a later lesson in this chapter).

For the subsequent examples, we'll collapse our examples down to a one-file solution for ease of reading.

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

The best way to tell the compiler to look in a particular namespace for an identifier is to use the scope resolution operator (::). The scope resolution operator tells the compiler that the identifier specified by the right-hand operand should be looked for in the scope of the left-hand operand.

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:

```cpp
#include <iostream>

namespace foo // define a namespace named foo
{
    // This doSomething() belongs to namespace foo
    int doSomething(int x, int y)
    {
        return x + y;
    }
}

namespace goo // define a namespace named goo
{
    // This doSomething() belongs to namespace goo
    int doSomething(int x, int y)
    {
        return x - y;
    }
}

int main()
{
    std::cout << foo::doSomething(4, 3) << '\n'; // use the doSomething() that exists in namespace
foo
    return 0;
}
```

This produces the expected result:

```
7
```

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

```cpp
#include <iostream>

namespace foo // define a namespace named foo
{
    // This doSomething() belongs to namespace foo
    int doSomething(int x, int y)
    {
        return x + y;
    }
}

namespace goo // define a namespace named goo
{
    // This doSomething() belongs to namespace goo
    int doSomething(int x, int y)
    {
        return x - y;
    }
}

int main()
{
    std::cout << goo::doSomething(4, 3) << '\n'; // use the doSomething() that exists in namespace goo
    return 0;
}
```

**This produces the result:**

```
1
```

**The scope resolution operator is great because it allows us to*explicitly* pick which namespace we want to look in, so there's no potential ambiguity. We can even do the following:**

```cpp
#include <iostream>

namespace foo // define a namespace named foo
{
    // This doSomething() belongs to namespace foo
    int doSomething(int x, int y)
    {
        return x + y;
    }
}

namespace goo // define a namespace named goo
{
    // This doSomething() belongs to namespace goo
    int doSomething(int x, int y)
    {
        return x - y;
    }
}

int main()
{
    std::cout << foo::doSomething(4, 3) << '\n'; // use the doSomething() that exists in namespace foo
    std::cout << goo::doSomething(4, 3) << '\n'; // use the doSomething() that exists in namespace goo
    return 0;
}
```

**This produces the result:**

```
7
1
```

## Using the scope resolution operator with no name prefix

**The scope resolution operator can also be used in front of an identifier without providing a namespace name (e.g `::doSomething`). In such a case, the identifier (e.g. `doSomething`) is looked for in the global namespace.**

```
#include <iostream>

void print() // this print lives in the global namespace
{
 std::cout << " there\n";
}

namespace foo
{
 void print() // this print lives in the foo namespace
 {
  std::cout << "Hello";
 }
}

int main()
{
 foo::print(); // call print() in foo namespace
 ::print(); // call print() in global namespace (same as just calling print() in this
case)

 return 0;
}
```

**In the above example, the `::print()` performs the same as if we'd called `print()` with no scope resolution, so use of the scope resolution operator is superfluous in this case. But the next example will show a case where the scope resolution operator with no namespace can be useful.**

## Identifier resolution from within a namespace

**If an identifier inside a namespace is used and no scope resolution is provided, the compiler will first try to find a matching declaration in that same namespace. If no matching identifier is found, the compiler will then check each containing namespace in sequence to see if a match is found, with the global namespace being checked last.**

```cpp
#include <iostream>

void print() // this print lives in the global namespace
{
 std::cout << " there\n";
}

namespace foo
{
 void print() // this print lives in the foo namespace
 {
  std::cout << "Hello";
 }

 void printHelloThere()
 {
  print(); // calls print() in foo namespace
  ::print(); // calls print() in global namespace
 }
}

int main()
{
 foo::printHelloThere();

 return 0;
}
```

This prints:

```
Hello there
```

In the above example, `print()` is called with no scope resolution provided. Because this use of `print()` is inside the `foo` namespace, the compiler will first see if a declaration for `foo::print()` can be found. Since one exists, `foo::print()` is called.

If `foo::print()` had not been found, the compiler would have checked the containing namespace (in this case, the global namespace) to see if it could match a `print()` there.

Note that we also make use of the scope resolution operator with no namespace (`::print()`) to explicitly call the global version of `print()`.

## Multiple namespace blocks are 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 are considered part of the namespace.

**circle.h:**

```cpp
#ifndef CIRCLE_H
#define CIRCLE_H

namespace basicMath
{
    constexpr double pi{ 3.14 };
}

#endif
```

**growth.h:**

```cpp
#ifndef GROWTH_H
#define GROWTH_H

namespace basicMath
{
    // the constant e is also part of namespace basicMath
    constexpr double e{ 2.7 };
}

#endif
```

**main.cpp:**

```cpp
#include "circle.h" // for basicMath::pi
#include "growth.h" // for basicMath::e

#include <iostream>

int main()
{
    std::cout << basicMath::pi << '\n';
    std::cout << basicMath::e << '\n';

    return 0;
}
```

**This works exactly as you would expect:**

```
3.14
2.7
```

**The standard library makes extensive use of this feature, as each standard library header file contains its declarations inside a `namespace std` block contained within that header file. Otherwise the entire standard library would have to be defined in a single header file!**

**Note that this capability also means you could add your own functionality to the `std` namespace. Doing so causes undefined behavior most of the time, because the `std` namespace has a special rule, prohibiting extension from user code.**

**Warning**

**Do not add custom functionality to the std namespace.**

**When you separate your code into multiple files, you'll have to use a namespace in the header and source file.**

**add.h**

```cpp
#ifndef ADD_H
#define ADD_H

namespace basicMath
{
    // function add() is part of namespace basicMath
    int add(int x, int y);
}

#endif
```

**add.cpp**

```cpp
#include "add.h"

namespace basicMath
{
    // define the function add()
    int add(int x, int y)
    {
        return x + y;
    }
}
```

**main.cpp**

```cpp
#include "add.h" // for basicMath::add()

#include <iostream>

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

    return 0;
}
```

**If the namespace is omitted in the source file, the linker won't find a definition of `basicMath::add`, because the source file only defines `add` (global namespace). If the namespace is omitted in the header file, "main.cpp" won't be able to use `basicMath::add`, because it only sees a declaration for `add` (global namespace).**

## Nested namespaces

**Namespaces can be nested inside other namespaces. For example:**

```cpp
#include <iostream>

namespace foo
{
    namespace goo // goo is a namespace inside the foo namespace
    {
        int add(int x, int y)
        {
            return x + y;
        }
    }
}

int main()
{
    std::cout << foo::goo::add(1, 2) << '\n';
    return 0;
}
```

**Note that because namespace `goo` is inside of namespace `foo`, we access `add` as `foo::goo::add`.**

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

```cpp
#include <iostream>

namespace foo::goo // goo is a namespace inside the foo namespace (C++17 style)
{
  int add(int x, int y)
  {
    return x + y;
  }
}

int main()
{
    std::cout << foo::goo::add(1, 2) << '\n';
    return 0;
}
```

## Namespace aliases

**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, which allow us to temporarily shorten a long sequence of namespaces into something shorter:**

```cpp
#include <iostream>

namespace foo::goo
{
    int add(int x, int y)
    {
        return x + y;
    }
}

int main()
{
    namespace active = foo::goo; // active now refers to foo::goo

    std::cout << active::add(1, 2) << '\n'; // This is really foo::goo::add()

    return 0;
} // The active alias ends here
```

One nice advantage of namespace aliases: If you ever want to move the functionality within `foo::goo` to a different place, you can just update the `active` alias to reflect the new destination, rather than having to find/replace every instance of `foo::goo`.

```cpp
#include <iostream>

namespace foo::goo
{
}

namespace v2
{
    int add(int x, int y)
    {
        return x + y;
    }
}

int main()
{
    namespace active = v2; // active now refers to v2

    std::cout << active::add(1, 2) << '\n'; // We don't have to change this

    return 0;
}
```

It’s worth noting that namespaces in C++ were not originally 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 library lives under the singular namespace `std::` (with some nested namespaces used for newer library features). Some newer languages (such as C#) differ from C++ in this regard.

In general, you should avoid deeply nested namespaces.

## When you should use namespaces

In applications, namespaces can be used to separate application-specific code from code that might be reusable later (e.g. math functions). For example, physical and math functions could go into one namespace (e.g. `math::`). Language and localization functions in another (e.g. `lang::`).

When you write a library or code that you want to distribute to others, always place your code inside a namespace. The code your library is used in may not follow best practices -- in such a case, if your library’s declarations aren’t in a namespace, there’s an elevated chance for naming conflicts to occur. As an additional advantage, placing library code inside a namespace also allows the user to see the contents of your library by using their editor’s auto-complete and suggestion feature.

Next lesson
6.3 Local variables

Back to table of contents

Previous lesson
6.1 Compound statements (blocks)

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

DP N N FOUT

Newest