DigiPen
INSTITUTE OF TECHNOLOGY

# Chapter 4 | Namespaces

CS185

# Namespaces

*"The road to hell is paved with global variables."* – Steve McConnell

- Every type, function, object, or template declared at the global scope must have a unique name.
- The global scope is also referred to as the *global namespace scope*. (To distinguish it from other namespaces.)
- In small programs, this is not a problem, since one person may produce all of these names.
- With very large programs, however, it can be a problem, especially with libraries from other sources or programs written by many programmers.

A simple program with 3 global symbols:

```
#include <iostream> // cout, endl

int foo = 1;        // foo in global namespace
int bar = 2;        // bar in global namespace

int Div2(int value) // Div2 in global namespace
{
	return value / 2;
}

int main(void)
{
	std::cout << foo << std::endl;     // use global foo
	std::cout << bar << std::endl;     // use global bar
	std::cout << Div2(8) << std::endl; // use global Div2
	return 0;
}
```

If another source file (.cpp) has **foo, bar**, or **Div2**, it will cause problems.

One solution is to make them static:

```
static int foo = 1;  // file scope
static int bar = 2;  // file scope

static int Div2(int value) // file scope
{
	return value / 2;
}
```

but this will limit their use to this file only. A better solution is to put them in a unique namespace:

```
namespace IntroCppProgramming
{
	int foo = 1;
	int bar = 2;
	int Div2(int value)
	{
		return value / 2;
	}
}
```

However, placing these symbols in a namespace will prevent the program from compiling:

```
int main(void)
{
	std::cout << foo << std::endl;     //error, foo is undeclared
	std::cout << bar << std::endl;     // error, bar is undeclared
	std::cout << Div2(8) << std::endl; // error, Div2 is undeclared
	return 0;
}
```

We need to *qualify* the symbols in the namespace:

```
int main(void)
{
	std::cout << IntroCppProgramming::foo << std::endl;
	std::cout << IntroCppProgramming::bar << std::endl;
	std::cout << IntroCppProgramming::Div2(8) << std::endl;
	return 0;
}
```

The general form of a namespace definition is:

```
namespace user-defined-name
{
  declaration/definition
  declaration/definition
  ...
}
```

- The *user-defined-name* must be unique in the global namespace (or else it will be part of an existing namespace).
- Any declaration that can appear in the global namespace scope can appear in a user-defined namespace.
- This includes classes, variables (declared/defined), functions (declared/defined), and templates.
- The names within the namespace include the namespace (e.g. *IntroCppProgramming::Div2* is **not** the same as *Div2* by itself.)

- Namespace definitions do not have to be contiguous:

```
namespace IntroCppProgramming
{
	int foo = 1;
	int bar = 2;
}

// Lots of other code here ...

namespace IntroCppProgramming
{
	int Div2(int value)
	{
		return value / 2;
	}
}
```

- However, if there are definitions needed by the program, they must still be seen by the compiler before they are used:

```
namespace IntroCppProgramming
{
	int foo = 1;
	int bar = 2;
}

int main(void)
{
	// Ok
	std::cout << IntroCppProgramming::foo << std::endl;
	// Ok
	std::cout << IntroCppProgramming::bar << std::endl;
	// error, Div2 is not part of namespace
	std::cout << IntroCppProgramming::Div2(8) << std::endl;
	return 0;
}

namespace IntroCppProgramming
{
	int Div2(int value)
	{
		return value / 2;
	}
}
```

- We can declare portions in one namespace definition and define them in another. This is now OK:

```
namespace IntroCppProgramming
{
	int foo = 1;
	int bar = 2;
	int Div2(int value); // Declaration/prototype
}

int main()
{
	// Ok
	std::cout << IntroCppProgramming::foo << std::endl;
	// Ok
	std::cout << IntroCppProgramming::bar << std::endl;
	// Ok, compiles and links
	std::cout << IntroCppProgramming::Div2(8) << std::endl;

	return 0;
}

namespace IntroCppProgramming
{
	int Div2(int value)  // Definition
	{
		return value / 2;
	}
}
```

- What is the specific problem if the second namespace definition (for `Div2`) above is missing? (Hint: Will it *compile* without it?)
- Note also that the separate definitions of the same namespace (as above) can be in separate files as well. They don't have to be in the same physical source file (and often they won't be).
- This gives you the flexibility to put the *interface* for your code into the public files (header files) where your users can see it, and keep the *implementation* hidden in the .cpp files:

**Helpers.h**

```
namespace Helpers
{
	extern int Counter;
	int FooFn(void);
	int BarFn(void);
}
```

**Helpers.cpp**

```
namespace Helpers
{
	int Counter = 25;
	int FooFn(void)
	{
		return 123;
	}

	int BarFn(void)
	{
		return 456;
	}
}
```

You can use these from `main.cpp` like this:

```
#include <iostream> // cout, endl
#include "Helpers.h"

int main(void)
{
	std::cout << FooFn() << std::endl;		// error, FooFn undeclared
	std::cout << Helpers::FooFn() << std::endl;	// Ok
	std::cout << Helpers::BarFn() << std::endl;	// Ok
	std::cout << Helpers::Counter << std::endl;	// Ok
	return 0;
}
```

Why do we use the **extern** keyword in the header file?

# Unnamed Namespaces

What happens when we run out of unique names for namespaces?

- It's unlikely to happen, but as more and more code uses namespaces, the chances for a collision are high.
- Namespace names are global, so there's no way to protect them from other global names. (Think **static**)
- This is a problem if code uses lots of small namespaces.
- We could come up with some kind of GUID (Globally Unique ID) scheme to guarantee unique namespaces:

```
namespace NS_1E266980_A661_48B6_94D1_C9DEA80A328B
{
	// stuff
}

namespace NS_6FB60AE7_AEEE_4285_88A7_6F0C28B34B5B
{
	// other stuff
}
```

- A better approach is *unnamed namespaces*.

```
#include <iostream> // cout, endl

namespace
{
	double sqrt(double x) { return x; }
}

int main(void)
{
	std::cout << sqrt(25.0) << std::endl;	// No qualification needed
	return 0;
}
```

- There is only one ***sqrt*** function in our program, and it is in an unnamed namespace.
- No qualification (couldn't even if you wanted to because it has no name.)
- Symbols in the unnamed namespace are local to this file (similar to the **static** keyword).

If we have a symbol in an unnamed namespace that is the same as a global symbol in our program, we won't be able to access the symbol in the unnamed namespace.

```
#include <iostream> // cout, endl
#include <cmath>    // sqrt

namespace
{
	double sqrt(double x)
	{
		return x;
	}
}

// global
double sqrt(double x)
{
	return x;
}

int main()
{
	// Global sqrt function defined in this file
	std::cout << ::sqrt(25.0) << std::endl;
	// sqrt from std namespace
	std::cout << std::sqrt(25.0) << std::endl;
	// Line 15: Ambiguous (from global or unnamed namespace?)
	std::cout << sqrt(25.0) << std::endl;

	return 0;
}
```

This is there error messages from the Microsoft compiler:

| Severity | Code | Description |
|---|---|---|
| Error | C2668 | 'sqrt': ambiguous call to overloaded function |

# Scope Resolution Operator

Example:

```
#include <iostream> // cout, endl

int foo = 1; // global
int bar = 2; // global

void fn1(void)
{
	int foo = 10;    // local foo #1 hides global foo
	int bar = foo;   // local bar #1 hides global bar (set to local foo)
	int baz = ::foo; // local baz #1 is set to global foo

	if (bar == 10)
	{
		int foo = 100; // local foo #2 hides local #1 and global
		bar = foo;     // local bar #1 is set to local foo #2
		foo = ::bar;   // local foo #2 is set to global bar
	}

	::foo = foo;   // global foo is set to local foo #1
	::bar = ::foo; // global bar is set to global foo

	std::cout << "foo is " << foo << std::endl;     // local foo #1 is 10
	std::cout << "bar is " << bar << std::endl;     // local bar #1 is 100
	std::cout << "::foo is " << ::foo << std::endl; // global foo is 10
	std::cout << "::bar is " << ::bar << std::endl; // global bar is 10
}
```

Notes:

- When the *scope resolution operator* is placed before a symbol (as above), it indicates that the symbol should be accessed from the global namespace. (Always global)
- Within the **`if`** statement above, the newly defined *foo* hides local *foo* #1.
- There is no way to access local *foo* #1 within the **`if`** statement.
- This means that if you hide a symbol in an outer scope, you can never refer to it unless the hidden symbol was global. (No way to access any symbols in any *intermediate* scope.)
- In C, if you hide a global variable, there is no way to access the global variable. (There is no scope resolution operator in C.)

# Namespaces Aliases

Given these namespaces:

```
namespace AdvancedProgramming
{
	int foo = 11;
	int bar = 12;
	int f1(int x) { return x / 2; }
}

namespace IntroductoryProgramming
{
	int foo = 21;
	int bar = 22;
	int Div2(int x) { return x / 2; }
}
```

using them requires a lot of typing:

```
int main(void)
{
	std::cout << AdvancedProgramming::foo << std::endl;
	std::cout << IntroductoryProgramming::Div2(8) << std::endl;

	return 0;
}
```

To allow unique namespaces and to shorten the names, you can create a *namespace alias*

```
// Declare these after the namespace definitions above
namespace AP = AdvancedProgramming;
namespace IP = IntroductoryProgramming;

int main(void)
{
	// Now, use the shorter aliases
	std::cout << AP::foo << std::endl;
	std::cout << IP::foo << std::endl;

	std::cout << AP::f1(8) << std::endl;
	std::cout << IP::Div2(8) << std::endl;

	return 0;
}
```

**Class Design Tip: Don't create very terse namespaces (like std). Create unique and meaningful namespaces and let the user create shorthand notation with aliases.**

# Using Directives

A *using directive* allows you to make all of the names in a namespace visible at once:

- Assume we have these symbols in a namespace:

```
namespace Stuff
{
	int foo = 11;		// Stuff::foo
	int bar = 12;		// Stuff::bar
	int baz = 13;		// Stuff::baz
}
```

- We can make them all accessible with a using directive:

```
// Everything in Stuff (foo, bar, baz) is visible from here down in the file
using namespace Stuff;

int main()
{
	std::cout << foo << std::endl;  // Stuff::foo
	std::cout << bar << std::endl;  // Stuff::bar
	std::cout << baz << std::endl;  // Stuff::baz

	return 0;
}
```

- Using directives are scoped; apply only within block where directive is used:

```
int main()
{
	// Everything in Stuff (foo, bar, baz) is visible only in main, now
	using namespace Stuff;

	std::cout << foo << std::endl;  // Stuff::foo
	std::cout << bar << std::endl;  // Stuff::bar
	std::cout << baz << std::endl;  // Stuff::baz

	return 0;
}

// Unqualified members in Stuff not available here.
```

- Ambiguity errors are detected when an ambiguous name is referenced, not when the directive is encountered.
- Qualified names can override the using directive.

More detailed example:

```
namespace Stuff
{
	int foo = 11;		// Stuff::foo
	int bar = 12;		// Stuff::bar
	int baz = 13;		// Stuff::baz
}

void f1(void)
{
	int foo = 3;		// local, hides nothing
	int x = Stuff::foo; // OK
	int y = bar;		// error, bar is undeclared
}

int foo = 20;		// global ::foo

int main(void)
{
	// Stuff's members are accessible without Stuff:: qualifier
	using namespace Stuff;

	// no problem, global
	std::cout << ::foo << std::endl;
	// no problem, Stuff::foo
	std::cout << Stuff::foo << std::endl;
	// error, foo is ambiguous (global or Stuff::foo?)
	std::cout << foo << std::endl;

	std::cout << bar << std::endl;  // Stuff::bar
	std::cout << baz << std::endl;  // Stuff::baz

	int foo = 3;		// OK, hides Stuff::foo and global ::foo
	int x = Stuff::foo; // OK, use qualified name
	x = foo;			// OK, local foo above
	x = ::foo;			// OK, global foo

	return 0;
}
```

Summary:

- Using directives were created to help migrate existing (pre-namespace) code.
- It is not meant to be used to make it "easier" on the programmer (by saving keystrokes).
- Many using directives will cause the global namespace to be polluted, which is the primary purpose of namespaces to begin with.
- It's best to avoid using directives, but may be useful if you are dealing with a lot of legacy code (old code that you didn't write).
- **Never** use them in header files that are meant to be used by others. (Aren't all header files for others to use?)

# The Standard Namespace

Now that we've seen some of the details of how namespaces are created and used, we can see how they can be applied.

- This code should be easy to understand now:

```
#include <iostream>  // For cout and endl
using namespace std; // For access to *all* names inside std namespace

int main(void)
{
	cout << "Hello" << endl;
	return 0;
}
```

- A better way to write the above:

```
#include <iostream> // For cout and endl
using std::cout;    // using declaration, global scope
using std::endl;    // using declaration, global scope

int main(void)
{
	cout << "Hello" << endl;  // std::cout and std::endl
	return 0;
}
```

- An even better way to write the above:

```
#include <iostream> // For cout and endl

int main(void)
{
	using std::cout;    // using declaration, local scope
	using std::endl;    // using declaration, local scope

	cout << "Hello" << endl;  // std::cout and std::endl
	return 0;
}
```

- Yet the preferred way to write code that uses the C++ standard library:

```
#include <iostream>  // For cout and endl

int main(void)
{
	std::cout << "Hello" << std::endl;
	return 0;
}
```

- There are thousands of names in the C++ global namespace, so the chances you collide with one is pretty good.
- You should learn to take control over when, where, and how names are introduced into your programs (don't introduce names "accidentally").
- C++ gives the programmer complete control, but this power is often abused (or not understood) by beginning C++ programmers.