Using Namespaces

By Alex Allain

One of C++'s less heralded additions is addition of *namespaces*, which can be used to structure a program into "logical units". A namespace functions in the same way that a company division might function -- inside a namespace you include all functions appropriate for fulfilling a certain goal. For instance, if you had a program that connected to the Internet, you might have a namespace to handle all connection functions:

```
namespace net_connect
{
  int make_connection();
  int test_connection();
  //so forth...
}
```

You can then refer to functions that are part of a namespace by prefixing the function with the namespace name followed by the scope operator -- *::*. For instance,

```
net_connect::make_connection()
```

By enabling this program structure, C++ makes it easier for you to divide up a program into groups that each perform their own separate functions, in the same way that classes or structs simplify object oriented design. But namespaces, unlike classes, do not require instantiation; you do not need an object to use a specific namespace. You only need to prefix the function you wish to call with *namespace_name*:: -- similar to how you would call a static member function of a class.

Another convenience of namespaces is that they allow you to use the same function name, when it makes sense to do so, to perform multiple different actions. For instance, if you were implementing a low-level IO routine and a higher level IO routine that uses that lower level IO, you might want to have the option of having two different functions named "input" -- one that handles low-level keyboard IO and one that handles converting that IO into the proper data type and setting its value to a variable of the proper type.

So far, when we've wanted to use a namespace, we've had to refer to the functions within the namespace by including the namespace identifier followed by the scope operator. You can, however, introduce an entire namespace into a section of code by using a using-directive with the syntax

```
using  namespace namespace_name;
```

Doing so will allow the programmer to call functions from within the namespace without having to specify the namespace of the function while in the current scope. (Generally, until the next closing bracket, or the entire file, if you aren't inside a block of code.) This convenience can be abused by using a namespace globally, which defeats some of the purpose of using a namespace. A common example of this usage is

```
using namespace std;
```

which grants access to the std namespace that includes C++ I/O objects cout and cin.

Finally, you can introduce only specific members of a namespace using a using-declaration with the syntax

```
using namespace_name::thing;
```

One trick with namespaces is to use an unnamed namespace to avoid naming conflicts. To do so, simply declare a namespace with the normal syntax, but leave off the identifier; when this is done, you will have

```
namespace
{
  //functions
}
```

and within the namespace you are assured that no global names will conflict because each namespace's function names take precedence over outside function names.

Now, you might ask, how can you actually use anything in that namespace? When your program is compiled, the "anonymous" namespace you have created will be accessible within the file you created it in. In effect, it's as though an additional "using" clause was included implicitly. This effectively limits the scope of anything in the namespace to the file level (so you can't call the functions in that namespace from another other file). This is comparable to the effect of the static keyword.

Renaming namespaces

Finally, if you just don't feel like typing the entire name of namespace, but you're trying to keep to a good style and not use the *using* keyword, you can rename a namespace to reduce the typing:

```
namespace <new> = <old>
```

Related articles
The many uses of the static keyword in C and C++

Namespaces

Namespaces provide a method for preventing name conflicts in large projects.

Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes.

Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

Syntax

namespace *ns_name* { *declarations* }	(1)	
inline namespace *ns_name* { *declarations* }	(2)	(since C++11)
namespace { *declarations* }	(3)	
ns_name`::`*name*	(4)	
using namespace *ns_name*`;`	(5)	
using *ns_name*`::`*name*`;`	(6)	
namespace *name* `=` *qualified-namespace* `;`	(7)	
namespace *ns_name*`::`*name*	(8)	(since C++17)

1) Named namespace definition for the namespace *ns_name*.

2) Inline namespace definition for the namespace *ns_name*. Declarations inside *ns_name* will be visible in its enclosing namespace.

3) Unnamed namespace definition. Its members have potential scope from their point of declaration to the end of the translation unit, and have internal linkage.

4) Namespace names (along with class names) can appear on the left hand side of the scope resolution operator, as part of qualified name lookup.

5) using-directive: From the point of view of unqualified name lookup of any name after a using-directive and until the end of the scope in which it appears, every name from *ns_name* is visible as if it were declared in the nearest enclosing namespace which contains both the using-directive and *ns_name*.

6) using-declaration: makes the symbol *name* from the namespace *ns_name* accessible for unqualified lookup as if declared in the same class scope, block scope, or namespace as where this using-declaration appears.

7) *namespace-alias-definition*: makes *name* a synonym for another namespace: see namespace alias

8) nested namespace definition: `namespace A::B::C {` is equivalent to `namespace A { namespace B { namespace C {`

Explanation

Namespaces

`inline`(optional) **`namespace`** *attr*(optional) *identifier* **`{`**
namespace-body **`}`**

`inline`	-	if present, makes this an inline namespace (see below). Cannot appear on the *extension-namespace-definition* if the *original-namespace-definition* did not use `inline`
attr(C++17)	-	optional sequence of any number of attributes
identifier	-	either a previously unused identifier, in which case this is *original-namespace-definition* or the name of a namespace, in which case this is *extension-namespace-definition* or a sequence of enclosing namespace specifiers separated by ::, ending with *identifier*, in which case this is a *nested-namespace-definition* (since C++17)
namespace-body	-	possibly empty sequence of declarations of any kind (including class and function definitions as well as nested namespaces)

Namespace definitions are only allowed at namespace scope, including the global scope.

To reopen an existing namespace (formally, to be an *extension-namespace-definition*), the lookup for the *identifier* used in the namespace definition must resolve to a namespace name (not a namespace alias), that was declared as a member of the enclosing namespace or of an inline namespace within an enclosing namespace.

The *namespace-body* defines a namespace scope, which affects name lookup.

All names introduced by the declarations that appear within *namespace-body* (including nested namespace definitions) become members of the namespace *identifier*, whether this namespace definition is the original namespace definition (which introduced *identifier*), or an extension namespace definition (which "reopened" the already defined namespace)

A namespace member that was declared within a namespace body may be defined or redeclared outside of it using explicit qualification

```
namespace Q {
  namespace V { // V is a member of Q, and is fully defined within Q
// namespace Q::V { // C++17 alternative to the above two lines
    class C { void m(); }; // C is a member of V and is fully defined within
V
                           // C::m is only declared
    void f(); // f is a member of V, but is only declared here
  }
  void V::f() // definition of V's member f outside of V
              // f's enclosing namespaces are still the global namespace, Q,
and Q::V
  {
      extern void h(); // This declares ::Q::V::h
  }
  void V::C::m() // definition of V::C::m outside of the namespace (and the
class body)
                 // enclosing namespaces are the global namespace, Q, and
Q::V
  {
  }
}
```

Out-of-namespace definitions and redeclarations are only allowed after the point of declaration, only at namespace scope, and only in namespaces that enclose the original namespace (including the global namespace) and they must use qualified-id syntax (since C++14)

```
namespace Q {
  namespace V { // original-namespace-definition for V
    void f(); // declaration of Q::V::f
  }
  void V::f() {} // OK
  void V::g() {} // Error: g() is not yet a member of V
  namespace V { // extension-namespace-definition for V
    void g(); // declaration of Q::V::g
  }
}
```

```
namespace R { // not a enclosing namespace for Q
   void Q::V::g() {} // Error: cannot define Q::V::g inside R
}
void Q::V::g() {} // OK: global namespace encloses Q
```

Names introduced by friend declarations within a non-local class X become members of the innermost enclosing namespace of X, but they do not become visible to lookup (neither unqualified nor qualified) unless a matching declaration is provided at namespace scope, either before or after the class definition. Such name may be found through ADL which considers both namespaces and classes.

Only the innermost enclosing namespace is considered by such friend declaration when deciding whether the name would conflict with a previously declared name.

```
void h(int);
namespace A {
  class X {
    friend void f(X); // A::f is a friend
    class Y {
        friend void g(); // A::g is a friend
        friend void h(int); // A::h is a friend, no conflict with ::h
    };
  };
  // A::f, A::g and A::h are not visible at namespace scope
  // even though they are members of the namespace A
  X x;
  void g() {  // definition of A::g
     f(x); // A::X::f is found through ADL
  }
  void f(X) {}       // definition of A::f
  void h(int) {}     // definition of A::h
  // A::f, A::g and A::h are now visible at namespace scope
  // and they are also friends of A::X and A::X::Y
}
```

Inline namespaces

(since C+

An inline namespace is a namespace that uses the optional keyword `inline` in its *original-namespace-definition*. (+11)

Members of an inline namespace are treated as if they are members of the enclosing namespace in many situations (listed below). This property is transitive: if a namespace N contains an inline namespace M, which in turn contains an inline namespace O, then the members of O can be used as though they were members of M or N.

- A *using-directive* that names the inline namespace is implicitly inserted in the enclosing namespace (similar to the implicit using-directive for the unnamed namespace)
- In argument-dependent lookup, when a namespace is added to the set of associated namespaces, its inline namespaces are added as well, and if an inline namespace is added to the list of associated namespaces, its enclosing namespace is added as well.
- Each member of an inline namespace can be partially specialized, explicitly instantiated or explicitly specialized as if it were a member of the enclosing namespace.
- Qualified name lookup that examines the enclosing namespace will include the names from the inline namespaces even if the same name is present in the enclosing namespace.

```
{ // in C++14, std::literals and its member namespaces are inline
   using namespace std::string_literals; // makes visible operator""s
                                         // from
std::literals::string_literals
   auto str = "abc"s;
}
{
   using namespace std::literals; // makes visible both
                                  //
std::literals::string_literals::operator""s
                                  // and
std::literals::chrono_literals::operator""s
   auto str = "abc"s;
   auto min = 60s;
}
{
   using std::operator""s; // makes both
```

```
std::literals::string_literals::operator""s
                           // and
std::literals::chrono_literals::operator""s visible
   auto str = "abc"s;
   auto min = 60s;
}
```

Note: the rule about specializations allows library versioning: different implementations of a library template may be defined in different inline namespaces, while still allowing the user to extend the parent namespace with an explicit specialization of the primary template.

Unnamed namespaces

The *unnamed-namespace-definition* is a namespace definition of the form

inline(optional) **namespace** *attr*(optional) **{**
namespace-body **}**

attr(C++17)	-	optional sequence of any number of [attributes]

This definition is treated as a definition of a namespace with unique name and a *using-directive* in the current scope that nominates this unnamed namespace.

```
namespace {
    int i;  // defines ::(unique)::i
}
void f() {
    i++;  // increments ::(unique)::i
}

namespace A {
    namespace {
        int i; // A::(unique)::i
        int j; // A::(unique)::j
    }
    void g() { i++; } // A::unique::i++
}
```

```
using namespace A; // introduces all names from A into global namespace
void h() {
    i++;    // error: ::(unique)::i and ::A::(unique)::i are both in scope
    A::i++; // ok, increments ::A::(unique)::i
    j++;    // ok, increments ::A::(unique)::j
}
```

Even though names in an unnamed namespace may be declared with external linkage, they are never accessible from other translation units because their namespace name is unique. (until C++11)

Unnamed namespaces as well as all namespaces declared directly or indirectly within an unnamed namespace have internal linkage, which means that any name that is declared within an unnamed namespace has internal linkage. (since C++11)

Using-declarations

Introduces a name that is defined elsewhere into the declarative region where this using-declaration appears.

using typename(optional) *nested-name-specifier unqualified-id* **;**	(until C++17)
using *declarator-list* **;**	(since C++17)

- *nested-name-specifier* - a sequence of names and scope resolution operators `::`, ending with a scope resolution operator. A single `::` refers to the global namespace.
- *unqualified-id* - an id-expression
- *typename* - the keyword `typename` may be used as necessary to resolve dependent names, when the using-declaration introduces a member type from a base class into a class template
- *declarator-list* - comma-separated list of one or more declarators of the form **typename**(optional) *nested-name-specifier unqualified-id*. The last declarator may be an ellipsis, although that form is only meaningful in derived class definitions

Using-declarations can be used to introduce namespace members into other namespaces and block scopes, or to introduce base class members into derived class definitions.

A using-declaration with more than one using-declarator is equivalent to a corresponding sequence of using-declarations with one using- (since C++17)

declarator.

For the use in derived class definitions, see using declaration.

Names introduced into a namespace scope by a using-declaration can be used just like any other names, including qualified lookup from other scopes:

```
void f();
namespace A {
    void g();
}
namespace X {
    using ::f; // global f is now visible as ::X::f
    using A::g; // A::g is now visible as ::X::g
    using A::g, A::g; // (C++17) OK: double declaration allowed at namespace
scope
}
void h()
{
    X::f(); // calls ::f
    X::g(); // calls A::g
}
```

If, after the using-declaration was used to take a member from a namespace, the namespace is extended and additional declarations for the same name are introduced, those additional declarations do not become visible through the using-declaration (in contrast with using-directive). One exception is when a using-declaration names a class template: partial specializations introduced later are effectively visible, because their lookup proceeds through the primary template.

```
namespace A {
    void f(int);
}
using A::f; // ::f is now a synonym for A::f(int)

namespace A { // namespace extension
   void f(char); // does not change what ::f means
}
void foo() {
```

```
    f('a'); // calls f(int), even though f(char) exists.
}
void bar() {
   using A::f; // this f is a synonym for both A::f(int) and A::f(char)
   f('a'); // calls f(char)
}
```

Using-declarations cannot name template-id, namespace, or a scoped enumerator. Each declarator in a using-declaration introduces one and only one name, for example using-declaration for an enumeration does not introduce any of its enumerators.

All restrictions on regular declarations of the same names, hiding, and overloading rules apply to using-declarations:

```
namespace A {
    int x;
}
namespace B {
    int i;
    struct g { };
    struct x { };
    void f(int);
    void f(double);
    void g(char); // OK: function name g hides struct g
}
void func() {
    int i;
    using B::i; // error: i declared twice

    void f(char);
    using B::f; // OK: f(char), f(int), f(double) are overloads
    f(3.5); // calls B::f(double)

    using B::g;
    g('a');      // calls B::g(char)
    struct g g1; // declares g1 to have type struct B::g
```

```
    using B::x;
    using A::x;  // OK: hides struct B::x
    x = 99;      // assigns to A::x
    struct x x1; // declares x1 to have type struct B::x
}
```

If a function was introduced by a using-declaration, declaring a function with the same name and parameter list is ill-formed (unless the declaration is for the same function). If a function template was introduced by a using-declaration, declaring a function template with the same name, parameter type list, return type, and template parameter list is ill-formed. Two using-declarations can introduce functions with the same name and parameter list, but if a call to that function is attempted, the program is ill-formed.

```
namespace B {
    void f(int);
    void f(double);
}
namespace C {
    void f(int);
    void f(double);
    void f(char);
}
void h() {
    using B::f; // introduces B::f(int), B::f(double)
    using C::f; // introduces C::f(int), C::f(double), and C::f(char)
    f('h');      // calls C::f(char)
    f(1);        // error: B::f(int) or C::f(int)?
    void f(int); // error: f(int) conflicts with C::f(int) and B::f(int)
}
```

If an entity is declared, but not defined in some inner namespace, and then declared through using-declaration in the outer namespace, and then a definition appears in the outer namespace with the same unqualified name, that definition is a member of the outer namespace and conflicts with the using-declration: (since C++14)

```
namespace X {
  namespace M {
     void g(); // declares, but doesn't define X::M::g()
  }
  using M::g;
  void g();   // Error: attempt to declare X::g which conflicts with
X::M::g()
}
```

More generally, a declaration that appears in any namespace scope and introduces a name using an unqualified identifier always introduces a member into the namespace it's in and not to any other namespace. The exceptions are explicit instantiations and explicit specializations of a primary template that is defined in an inline namespace: because they do not introduce a new name, they may use unqualified-id in an enclosing namespace.

Using-directives

A *using-directive* is a block-declaration with the following syntax:

attr(optional) **using namespace** *nested-name-specifier*(optional) *namespace-name* **;**	(1)

attr(C++11)	- any number of attributes that apply to this using-directive
nested-name-specifier	- a sequence of names and scope resolution operators `::`, ending with a scope resolution operator. A single `::` refers to the global namespace.
namespace-name	- a name of a namespace. When looking up this name, lookup considers namespace declarations only

Using-directives are allowed only in namespace scope and in block scope. From the point of view of unqualified name lookup of any name after a using-directive and until the end of the scope in which it appears, every name from *namespace-name* is visible as if it were declared in the nearest enclosing namespace which contains both the using-directive and *namespace-name*.

Using-directive does not add any names to the declarative region in which it appears (unlike the using-declaration), and thus does not prevent identical names from being declared.

Using-directives are transitive for the purposes of unqualified lookup: if a scope contains a using-directive that nominates a *namespace-name*, which itself contains using-directive for some *namespace-name-2*, the effect is as if the using directives from the second namespace appear within the first. The

order in which these transitive namespaces occur does not influence name lookup.

```
namespace A {
    int i;
}
namespace B {
    int i;
    int j;
    namespace C {
        namespace D {
            using namespace A; // all names from A injected into global
namespace
            int j;
            int k;
            int a = i; // i is B::i, because A::i is hidden by B::i
        }
        using namespace D; // names from D are injected into C
                           // names from A are injected into global namespace
        int k = 89; // OK to declare name identical to one introduced by a
using
        int l = k;  // ambiguous: C::k or D::k
        int m = i;  // ok: B::i hides A::i
        int n = j;  // ok: D::j hides B::j
    }
}
```

If, after a using-directive was used to nominate some namespace, the namespace is extended an additional members and/or using-directives are added to it, those additional members and the additional namespaces are visible through the using-directive (in contrast with using-declaration)

```
namespace D {
   int d1;
   void f(char);
}
using namespace D; // introduces D::d1, D::f, D::d2, D::f,
                   //  E::e, and E::f into global namespace!
```

```
int d1; // OK: no conflict with D::d1 when declaring
namespace E {
    int e;
    void f(int);
}
namespace D { // namespace extension
    int d2;
    using namespace E; // transitive using-directive
    void f(int);
}
void f() {
    d1++; // error: ambiguous ::d1 or D::d1?
    ::d1++; // OK
    D::d1++; // OK
    d2++; // OK, d2 is D::d2
    e++; // OK: e is E::e due to transitive using
    f(1); // error: ambiguous: D::f(int) or E::f(int)?
    f('a'); // OK: the only f(char) is D::f(char)
}
```

Notes

The using-directive `using namespace std;` at any namespace scope introduces every name from the namespace `std` into the global namespace (since the global namespace is the nearest namespace that contains both `std` and any user-declared namespace), which may lead to undesirable name collisions. This, and other using directives are generally considered bad practice at file scope of a header file.

Example

This example shows how to use a namespace to create a class that already has been named in the `std` namespace.

Run this code

```
#include <vector>


namespace vec {
```

```
    template< typename T >
    class vector {
        // ...
    };

} // of vec

int main()
{
    std::vector<int> v1; // Standard vector.
    vec::vector<int> v2; // User defined vector.

    v1 = v2; // Error: v1 and v2 are different object's type.

    {
        using namespace std;
        vector<int> v3; // Same as std::vector
        v1 = v3; // OK
    }

    {
        using vec::vector;
        vector<int> v4; // Same as vec::vector
        v2 = v4; // OK
    }

    return 0;
}
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.