Namespace in C++

Lee S. Koh

- We use function to group code into a module. We use class to group functions, data and types into a type. Both function and class do two things for us:
  - Allow us to define various "entities" without worrying that their names clash with other names in our program.
  - Give us a name to refer to what we have defined.
- What we still need is something to group classes, functions, data and types into an identifiable and named part of a program without defining a type. C++ provides such a mechanism through the `namespace` feature:
  - A *namespace* groups together related declarations (of names) for the purpose of avoiding name clashes.
  - For example, suppose that we buy two different general-purpose class libraries from two different vendors, and each has some features that we'd like to use. We'd include the header files for both libraries accordingly:

```
#include "lib1.h"
#include "lib2.h"
```

But what if the files have identically-named "entities" (say classes, see *e.g.* below) appearing at *global* scope?

```
// lib1.h
class Gadget { ... };
class Time { ... };
...
```

```
// lib2.h
class Gadget { ... };
class Date { ... };
...
```

Our use of the libraries will lead to compilation error because **`Gadget`** is defined twice at the *global* scope. In other words, each vendor has included a separate **`Gadget`** class in the library, leading to a compile-time name clash. Even if we could somehow use a workaround to get around this *compile-time* problem, there is still a potential problem of *link-time* clashes due to the libraries containing some identically-named symbols.

The problem would not arise if the vendors are good enough to use the `namespace` feature as shown below:

```
// lib1.h
namespace Vendor1
{
   class Gadget { ... };
   class Time { ... };
   ...
}
```

```
// lib2.h
namespace Vendor2
{
   class Gadget { ... };
   class Date { ... };
   ...
}
```

The two classes can now be distinguished with "mangled" names `Vendor1::Gadget` and `Vendor2::Gadget`.

- **`namespace`** definition (syntax) generally and a more inclusive example:

```
namespace identifier
{
   list of 0 or more declarations
}
```

```
namespace MyNamespace
{
   const double PI = 3.1416;
   typedef double value_type;
   enum MyColor { RED = 1, GREEN, BLUE };
   class MyClass { ... };
   class AnotherClass { ... };
   int MaxOf2Ints(int i1, int i2)
   { return (i1 > i2) ? i1 : i2; }
   void AnotherFunction();
}
```

  - **`namespace`** syntax is very similar to **`class`** syntax but notice the following:
    - There's *no access specification* (`public`, *etc.* allowed only if they are put *within* classes in the namespace).
    - There's *no semicolon after the closing curly brace.*
  - **`namespace`** can only appear at the *global* scope or within another **`namespace`**.
    - (For our purpose, in all likelihood, we don't have to worry about the latter.)

- If the `identifier` does not already exist in the current scope, a new namespace with that name is defined; if the `identifier` already exists, the definition for the existing namespace is *continued*. This way, a namespace can be *extended* to enable a single large namespace to be defined over *multiple files*:

```
namespace N // new namespace N defined
{
   int f(int); // f's prototype
   typedef double value_type;
}
```

```
namespace N // add more to namespace N
{
   int f(int x) { ... } // f's definition
   enum MyColor { RED = 1, GREEN, BLUE };
}
```

- But how would we actually use (refer to) names that have been grouped using namespace?
  - Explicit qualification (*i.e.*, use the "mangled" names of the "entities"):

```
#include "lib1.h"
#include "lib2.h"
void someFunc()
{
   Vendor1::Gadget g1;
   Vendor2::Date date_stamp;
   ...
}
```

```
#include <iostream>
int main()
{
   std::cout << "Hello World!"
             << std::endl;
   return 0;
}
```

  - **`using`** declaration:

```
#include "lib1.h"
#include "lib2.h"
void someFunc()
{
   using Vendor1::Gadget, Vendor2::Date;
   Gadget g1;
   Date date_stamp;
   ...
}
```

```
#include <iostream>
int main()
{
   using std::cout, std::endl;
   cout << "Hello World!" << endl;
   return 0;
}
```

  - **`using`** directive:

```
#include "lib1.h"
#include "lib2.h"
void someFunc()
{
   using namespace Vendor2;
   using Vendor1::Gadget;
   Gadget g1;        // Vendor1's Gadget
   Date date_stamp; // Vendor2's Date
   ...
}
```

```
#include <iostream>
int main()
{
   using namespace std;
   cout << "Hello World!" << endl;
   return 0;
}
```

- To effectively use the **`namespace`** feature to avoid name clashes, we still have to:
  - Make names of *different namespaces* distinct (among our own and from those defined in libraries we use).
  - Ensure names (of "entities") *within each namespace* don't clash.

  Thus a library vendor may use a lengthy name for a namespace to minimize potential clashes with other vendors:

```
namespace BestInTheWorldObjectsLibrary { ... }
```

  But using such lengthy namespace name to refer to names in the namespace (in "explicit qualification" fashion) will be unwieldy. A *namespace-aliasing* definition enables us to use a shorthand name to avoid the unwieldiness:

```
namespace BITWOL = BestInTheWorldObjectsLibrary;
...
```

  (**`BITWOL`** is declared as a namespace that stands for the longer name *in the scope of the alias definition*.)