New C++ Features

Rex Jaeschke

Edition: 1.0
Printing: November 25, 1999

The training materials associated with this book are available for license. These materials include a manuscript master, overhead transparencies, quick reference guides, instructor guide, and lab solutions in electronic format. Interested parties should contact the author at the address below.

Please address comments and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023 USA
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
rex@RexJaeschke.com

This book was typeset by the author using the TEX typesetting package, LATEX macros, and Makeindex indexing tool.

Contents

Chapter 1

Name Spaces

In this chapter we'll look at how the same name can be used to mean different things in the same program scope. Along the way we'll introduce the new keywords `namespace` and `using`.

1.1 Introduction

An identifier can designate multiple things in the same scope provided each designation belongs to a different identifier category. In such cases, usage of the name is determined by context. As a result, each identifier category has its own *name space*. Standard C has the following name spaces:

- Tag names: These can only be used in conjunction with structure, union, or enumerated types and must be preceded by the keyword `struct`, `union`, or `enum`, respectively.
- Member names: These exist inside structures and unions and must be preceded by the dot (`.`) or arrow (`->`) member-selection operator, as appropriate.
- Label names: These are always followed by a colon and are only used as the target of a `goto`. (`case` and `default` are keywords, not label names, so they cannot be the target of a `goto`.)
- Ordinary names: These are variable names, function names, type synonyms created using `typedef`, and enumeration constant names.

In the following example, the identifier `x` has four different meanings depending on the context of its usage:

```
struct x {                      /* tag name */
        int x;                  /* member name */
};

void x(double a, int b)         /* ordinary name */
{
x:      ;                       /* label name */
}
```

This example compiles without error using a C++ compiler, however, there is one very important difference regarding name spaces in C++ compared to those in C. In C++, tag names share the same name space as ordinary names. Given the structure definition above, in C we define a variable of that type as follows:

```
struct x s;
```

And while we can do the same in C++, we can omit the keyword `struct`:

```
x s;
```

and the vast majority of C++ programmers do exactly that. The rationale for allowing this is that C++ integrates user-defined types into the language so they can be used just like the built-in types. The same result can be achieved in C by using a `typedef`, as follows:

```
typedef struct x {
        int x;
} x;

x s;
```

Now we have `x` being declared as an ordinary name as well as a tag name.

1.2 Name Space Pollution

Once we start implementing and managing large projects, especially those that use libraries from different sources, we need to be concerned about clashes between names. What are the chances that two subsystems use the same name in the same scope for different purposes at the same time? And will such uses of this name conflict with each other? While we can change names in subsystems whose source we control, we cannot do so for most third-party libraries.

Certain naming conflicts can easily be avoided. All identifiers declared inside a block are local to that block so all we have to do is to make sure we don't have conflicts within that source block. And we can easily avoid having duplicate member names within any given structure or union. The real problem comes with names declared at file scope; that is, outside all functions. Specifically, the problem names are tags, nonlocal variables, functions, type synonyms, and enumeration constants. With each new header we include, we further pollute the 'public' name space, increasing the possibility of name space collision.

1.3 The New Namespace Machinery

To help reduce name space pollution and subsequent name space collision, two new keywords have been invented: `namespace` and `using`.

There are two common scenarios in which names collide: a conflict between names declared inside a header and those declared inside user code and, a conflict between names declared inside different headers, usually because they come from different sources. Since these are related problems, we'll look at the combined case, in which the same set of names is declared in two headers as well as the user's code, yet all three sets are intended to mean different things. Each set has three names: a global variable, a public function, and a class declaration. They are declared in the user code as follows (see ns01.cpp):

```
extern unsigned int Total;
int GetValue(void);
class X { public: int f(void); };
```

Header Lib1.h contains the following:

```
namespace Lib1 {
        extern unsigned int Total;
        int GetValue(void);
        class X { public: int f(void); };
}
```

while header Lib2.h looks like this:

```
namespace Lib2 {
        extern unsigned int Total;
        int GetValue(void);
        class X { public: int f(void); };
}
```

The keyword `namespace` defines a declarative region and all names—called *members*—declared within that region are local to that namespace, allowing the same names to be declared local to other namespaces. Typically, a namespace has a name (spelled according to the rules of an identifier). In this example, we have defined two namespaces, `Lib1` and `Lib2`, each containing three members. The three names declared in the user code belong to

what is called the global namespace (the only one we have for external names in C). **Note that unlike most other declaration constructs in C++, the closing brace of a namespace body is not followed by a semicolon.**

It should come as no surprise that to access a name within some namespace, we qualify it by adding its parent namespace name. For example, to access `Total`, `GetValue`, and `X` in namespace `Lib1`, we use `Lib1::Total`, `Lib1::GetValue`, and `Lib1::X`, respectively. Similarly, `Lib2::Total`, `Lib2::GetValue`, and `Lib2::X` access those names in namespace `Lib2`. To access those names in the global namespace we simply use their names—`Total`, `GetValue`, and `X`; we can also use the explicit global namespace qualifier `::`, as in `::Total`, `::GetValue`, and `::X`.

Clearly, the compiler needs to keep all three versions of each name separate so the linker can correctly resolve references to these names. It does this by adding the parent namespace name to the mangled name.[1]

Clearly, if the declarations of these names show them to be local to some namespace, their definitions had better indicate the same thing otherwise we'll get linker errors—let's look at their definitions:

```
unsigned int Total = 100;

int GetValue(void)
{
        return 101;
}

int X::f(void)
{
        return 102;
}
```

There are no surprises here; names in the global namespace are defined as before. However, we cannot precede their names with `::` in their definition although we can optionally do so when we use those names in an expression.

```
namespace Lib1 {
        unsigned int Total = 200;
        int GetValue(void)
        {
                return 201;
        }
        int X::f(void)
        {
                return 202;
        }
}
```

All we have done here is to enclose these definitions inside the namespace `Lib1` so their names get mangled the same as those in the corresponding declarations in Lib1.h.

```
namespace Lib2 {
        unsigned int Total = 300;
}

int Lib2::GetValue(void)
{
        return 301;
}

int Lib2::X::f(void)
{
        return 302;
}
```

[1] Since different compilers may use different name-mangling schemes, we might not be able to link object modules produced by different compilers running on the same system.

In the case of namespace `Lib2`, we have defined only `Total` inside a namespace. In the case of `GetValue` and `X::f`, we have included their parent namespace name explicitly in their names, as shown. Either approach works fine; it's simply a matter of style. However, at a glance, `Lib2::X::f` looks like `f` is a member of class `X` which is a member of class `Lib2` and, of course, that is not true. But a class really is a name space in its own right, so there really ought not to be much confusion in understanding this.

Consider the following example:

```
namespace A {
        #define MIN 10
        const int MAX = 100;
}

namespace B {
        const int MAX = 90;
}
```

Although `MIN` is an identifier, it is not a member of namespace `A`. The preprocessor has its own set of rules, and macro names are not subject to the name space and scoping rules of the C or C++ languages. On the other hand, `MAX` is a member of both namespaces. Since the proliferation of names from existing headers is largely created by macros, we should use `const`-qualified identifiers wherever possible instead, since those are subject to namespace rules.

By the way, we need to keep in mind that the name of a file scope namespace cannot be the same as any other name at file scope. For example, we cannot have a global variable or function with the same name as a file scope namespace. That is, there is not a separate name space for namespace names!

1.4 Nested Namespaces

Namespaces can be nested. For example (see ns02.cpp):

```
// declarations

namespace Outer {
        void f(int i);
        namespace Inner {
                void f(int i);
        }
}

void test()
{
        Outer::f(10);
        Outer::Inner::f(20);
}

// definitions

namespace Outer {
        void f(int i);
        namespace Inner {
                void f(int);
        }
}

void Outer::f(int i) {}
void Outer::Inner::f(int i) {}
```

For names such as `f`, all parent namespace names are somehow encoded in the name produced by the compiler. A namespace can be declared only at file scope or within another namespace.

1.5 Long Names and Namespace Aliases

The simplest way to use namespaces is to qualify each name explicitly with its parent namespace(s). The problem then becomes one of choosing unique names for the namespaces themselves. (Since there is no universal registry of namespace names, we can still get naming conflicts. The best we can do is to have a company-wide registry and to use some company- or project-specific prefix.)

Consider the case of Acme Corporation which sells a graphics library. To ensure that the entry points to their library do not conflict with other third-party library function names, the company declares all these names in their own namespace. But what to call this namespace? The obvious choice might be as follows:

```
namespace Acme_Corporation_Graphics {
        void DrawLine( /* ... */ );
        // ...
}
```

And while that certainly reduces the likelihood of a conflict with another vendor's header, it isn't the most convenient name to read and write. However, the use of such names is necessary if one vendor's namespaces are to be kept separate from another's. However, any user of such libraries knows exactly which third-party products they are currently using so they can use an abbreviated name without fear of name clashes. For example (see ns03a.cpp):

```
#include <ACGraphics.h>

namespace ACG = Acme_Corporation_Graphics;

void test()
{
        Acme_Corporation_Graphics::DrawLine();
        ACG::DrawLine();
}
```

The long- and short-form names in the function calls are equivalent. The short name simply acts like a macro except that the substitution is done by the compiler rather than the preprocessor.

We can use this name association capability along with conditional compilation to select different versions of some library. For example (see ns03b.cpp):

```
namespace Acme_Corporation_Graphics_V1 {
        void DrawLine( /* ... */ );
        // ...
}

namespace Acme_Corporation_Graphics_V2 {
        void DrawLine( /* ... */ );
        // ...
}

#ifdef ACG_V2
namespace ACG = Acme_Corporation_Graphics_V2;
#elif ACG_V1
namespace ACG = Acme_Corporation_Graphics_V1;
#else
#error "No ACG library was selected."
#endif

void test()
{
        ACG::DrawLine();
}
```

We can also create an alias for a series of nested namespaces. For example (see ns03c.cpp):

```
namespace A {
        namespace B {
                namespace C {
                        void f(int i);
                }
        }
}

namespace XX = A::B::C;

void test()
{
        XX::f(10);
}
```

Like a namespace name, a namespace alias cannot be a name already declared for any reason at the same scope.

1.6 Adding to a Namespace

It is possible to declare a given namespace multiple times in the same translation unit (i.e., source file plus included headers) with each such declaration having different members. The effect of doing so is cumulative; that is, we can add more members to an existing namespace. For example (see ns04.cpp):

```
namespace A {
        void f(int i);
        class X { /* ... */ };
}

void test1()
{
        A::f(10);
        A::f(10.5);     // calls f(int)
}

namespace A {
        int b(void);
        void f(double d);
        class Y { /* ... */ };
}

void test2()
{
        A::f(10);
        A::f(10.5);     // calls f(double)
}
```

This is a useful capability since it allows us to build a single namespace using contributions from different headers. We can even add overloaded versions of existing functions as is done with function `f` above. As noted in the source comments, the calls `A::f(10.5)` actually result in calls to different functions; however, that is not surprising nor should it be a real problem. Assuming the namespace contributions result from an `#include` directive, it is bad style to define a function (such as `test1`) between two such directives.

What if multiple contributions to a namespace redeclare the same name? The result is the same as if we had redeclared the same name in the global namespace.

1.7 The Standard Headers and `using` Directives

If we are starting a new project and creating new libraries, clearly we can and should start using namespaces. However, how can we integrate with existing libraries, including those described by the headers defined in the C++ standard?

Let's use the standard C header ctype.h as an example for our discussion. This header contains the following:

```
int isalpha(int);
int isupper(int);
// ...
```

In reality, there would be conditional compilation directives allowing these prototypes to be used with both C and C++ compilers, but those are irrelevant to our discussion.

The C++ standard declares all file scope names in the standard library to belong to the namespace `std`. Therefore, ctype.h will look something like this (see ns05.cpp):

```
namespace std {
        int isalpha(int);
        int isupper(int);
        // ...
}
```

However, that would require the prefix `std::` in front of each use of these names. Therefore, the header would also contain the following *using directive*:

```
using namespace std;
```

Now consider the following code fragment:

```
#include <ctype.h>

void test()
{
        int i = isupper('a');           // uses std:isupper
}
```

The `using` directive makes all the member names of namespace `std` visible. If the compiler can't resolve a name locally it looks in any namespaces that have been made visible via a `using` directive. Therefore, calls to `isupper` resolve to `std::isupper`.

Now that you get the idea, here's what is really going on in a library that conforms to the C++ standard. A new header called cctype (with no .h suffix) has been defined and it contains the following:

```
// header cctype

namespace std {
        int isalpha(int);
        int isupper(int);
        // ...
}
```

while ctype.h contains

```
// header ctype.h

#include <cctype>
using namespace std;
```

Old code that includes ctype.h directly gets all the `std` member names from that header dumped into the global namespace while code that includes cctype must access them explicitly using a `std::` prefix, unless a using directive is present.

The C standard defines 15 standard headers and amendment 1 (adopted early in 1996) added three more (iso646.h, wctype.h, and wchar.h). All of these exist in C++ implementations. As we saw with ctype.h, the .h is dropped and a leading c is added, so cstdio corresponds to stdio.h, cstring to string.h, and so on.

The inclusion of multiple C headers, as in the following example:

```
#include <string.h>
#include <ctype.h>
#include <stddef.h>
```

causes all member names declared therein to be dumped into the global namespace since each header will contain a `using namespace std` directive. As a result, existing code containing calls to C library functions and references to standard types will be preserved.

Of course, users should not add any member names to namespace `std`; that namespace is reserved by future versions of the C++ standard.

All names declared in user headers should be members of some user-defined namespace.

When used in the global name space, the using directive is seen more as a transition aid as vendors and users add namespace support to existing libraries. However, using directives can be very useful inside a namespace or a function definition.

1.8 The `using` Declaration

While a using directive allows all the names in some given namespace to be used without qualification, a *using declaration* allows member names to be made visible on an individual basis. Consider the following example (see ns06.cpp):

```
namespace A {
        class X { /* ... */ };
        extern int Y;
        void f(int);
        void f(double);
        // ...
}

using A::X;

X x1;
float Y;
void f(void);

void test1()
{
        X x2;
        char Y;
        using A::f;

        f(10);          // A::f(int);
        f(10.5);        // A::f(double);

        Y = 'A';        // local Y
        ::Y = 1.5f;     // global Y
        A::Y = 15;      // A::Y
}

void test2()
{
        f();            // global f(void);
        Y = 1.5f;       // global Y
}
```

The declaration using `A::X;` is at file scope, so the name `X` is now visible through the end of this translation unit and we use it to define objects `x1` and `x2`.

The declaration `using A::f;` is at block scope, so the name `f` is visible through the end of that block. And since `f` is the name of a set of overloaded functions, all versions of that function currently in scope are visible. And so the

two calls to `f` result in the calls as shown in the corresponding comments. Since this using declaration goes out of scope at the end of function `test1`, the `f` called in `test2` is the global `f` declared at file scope.

The name `A::Y` is never made visible via a using directive or declaration; however, we can access it explicitly as shown in `test1`.

A using declaration can also be useful in adjusting member access. For example (see ns07a.cpp):

```
class Base {
        // ...
public:
        void f(int);
};

class Derived : private Base {
        // ...
public:
        Base::f;
};

void test()
{
        Derived d;

        d.f(10);
}
```

Since `Base` is a private base class of `Derived`, its member function `f` cannot ordinarily be called for an object of type `Derived`. However, by adjusting the access of `f` with `Base::f` in the public section of `Derived`, we can make `f` available. However, now that using declarations exist, they can be used to solve this problem instead, as follows (see ns07b.cpp):

```
class Derived : private Base {
        // ...
public:
        using Base::f;
};
```

The using declaration is preferred and the old-style approach has been deprecated.

1.9 File Scope Statics

An object or function declared `static` at file scope is accessible by name only from within that same translation unit. We can achieve the same affect using a unnamed namespace. For example, instead of the following (see ns08a.cpp):

```
static int si;
static void local(int i);
```

we can use (see ns08b.cpp)

```
namespace {
        int si;
        void local(int i);
}
```

instead. All unnamed namespaces in the same scope in the same translation unit have the same unknown name. The names of members of unnamed namespaces are not available to other translation units.

The C++ standard's committee feels so strongly about this approach being superior to that of using `static`, it has declared the `static` approach to be deprecated.