At the very least, C++ namespaces do what they say. The idea of namespaces is quite simple, but their sensible application is not always obvious. **Kevlin Henney** explains how to avoid getting lost in namespace

What's in a name?

THE MOTIVATION FOR NAMESPACES IN C++ is twofold and easy to sketch out: to reduce the scope, quite literally, for name clashes at the global level and to indicate the origin, ownership and conceptual grouping of a global identifier. However, it turns out that while the motivation and core mechanism are fairly straightforward, their effective use is more of a challenge. As with so many things in programming, having a rough idea and a piece of syntax is rarely enough. Let's go back to first principles.

The clash

Consider a common-or-garden date class. Reasonable names for such a type include `date` and `Date`. Indeed, unless someone is wilfully ignoring received wisdom on writing good C++—such as choosing `DATE`, which uses the macro-naming convention, or `_Date`, which uses a name reserved for use by the compiler and standard library—that pretty much exhausts the available options for the most obvious name.

What happens if someone else has chosen the same name for a type? If you are working on your own and are not linking in third party code, the odds are that the problem of name clashes will not affect you. However, a lot of software is developed by teams of programmers building on existing code, developed both in-house and externally ('out-house', given the quality of some of what is available both freely and commercially).

While you don't expect name clashes to be a daily occurrence, when they happen they are tedious and wasteful. Someone has to make a change. The chances are that it will be you. You have to start thinking laterally for a solution if the problem occurs between code from two libraries over which you have no control. And don't think that the problem manifests itself only in the source at compile time when you include two headers with the clashing names: the problem can manifest itself at link time, or even at run time as a bug. The compiler may detect duplicate symbols, such as two definitions for the default constructor, `date::date()`. If the clash is with templates or inline functions, the problem can be more subtle. Your code may link, but at run time strange things start to happen.

For most compilers, template or inline function code that the compiler chooses not to inline (remember, `inline` is a suggestion, not a command) is generated out of line in each translation unit in which it is used. For you to end up with a single, unique definition in the final executable image, the linker has to discard such duplicates. In this case, the baby and bathwater may well go out together, so that programmer-generated duplicates are thrown out with the compiler-generated ones. This means that in the final image the 'wrong' one may be called, hence bugs.

There are a number of creative solutions for compile- and link-time clashes, including preprocessor surgery or hiding the combatant names from one another by strictly segregating their usage across separate DLLs. This isn't a strategy, it is post hoc firefighting.

What needs fixing?

Before addressing how clashes can be resolved more intentionally, it is perhaps worth pausing for a moment to consider what exactly needs solving. First of all, the problem concerns clashes in the global namespace, i.e. the top-level names in your program. Local variables and class members are not the problem and the plain scope-resolution operator (`::`) has always been able to resolve such issues. Second, clashes related to the use of the preprocessor are of a different kind. The

FACTS AT A GLANCE

- Name collisions between classes or constants are more annoying than they are common.
- Using identifier prefixes or nesting names inside a `struct` is a step in the right direction, but only a small step.
- C++ namespaces introduce named, extensible scopes with all the right collision avoidance properties.
- In choosing a good name for a namespace, pick on something stable. Don't use department names, company names or domain names.

solution to such problems is quite simply to stop abusing the preprocessor. Not exactly a complex solution, but, as with any wasting addiction, the habit may be hard for some C++ programmers to break. Third, clashing global variables, i.e. non-`const` data visible to a whole program, tend to suggest that a body of code has much deeper problems than its authors either recognise or are prepared to admit.

Therefore, reasonably, non-nested type names can clash and constant names can clash (most likely, `enum` constants). What about function names? It turns out that they rarely clash. Overloading tends to take them out of each other's way. It is rare that two functions with the same name and argument count will match exactly because they will typically work on separate types: that is, types that are specific to their library or subsystem.

With a traditional object-oriented style, you will not find that many functions floating around the global namespace, so function name clashes effectively disappear off the radar. Most functions will be clearly associated with classes as `static` or, most commonly, non-`static` members. The most common non-member functions in a C++ OO program will be overloaded operators, and these will be associated with the type they operate on, so they are already uniquely qualified and in no need of clash protection.

What about templates? For class templates clashes are as much a problem as they are for ordinary type names. Function templates, however, fall between the two poles. They may clash with other function templates that are similarly parameterised. For instance, two function templates with the same name taking a single argument of parameterised type will be indistinguishable from one another in the eyes of the compiler and linker, and will therefore clash. However, if they have either different argument counts or more specific argument types, such as a non-templated type from their library of origin, they will pass by one another without interference.

Some traditional approaches

Before namespaces were added to C++, there were basically two proactive approaches that could be taken to avoiding name clashes: You could either use a common prefix ahead of all identifiers, or use `struct`s or classes to provide a mini-namespace.

The problem with the first approach is that, in practice, most choices for prefixes are one or two characters in length. Much more than this is normally considered a chore, and it tends to hide the real part of the name unless suitable spacing or separation is used, which in practice means an underscore character. However, there is not a lot of useful informational bandwidth in a couple of characters. The chances that the abbreviations are both clearly meaningful and suitably collision-proof are typically low. As a general solution, there seem to be genuine practical and cognitive problems with identifier prefixes of any length.

Another problem with prefixes is that many programmers already use them for other purposes. It is true that in most such cases the practice is not a particularly effective one (e.g. restating the blindingly obvious by using `C` to prefix class names or the more general `T` to prefix type names), but programmers nevertheless persist in their use. You could end up with prefix soup if an uncritical programmer fell for all the possible name mangling fashions.

Name mangling is really the responsibility of the language and compiler. Compilers already mangle function names to include class membership and argument type information. A common pre-namespace idiom was to use empty classes or `struct`s to wrap `typedef`s or `enum` types. For expressing regular procedural functions, some programmers also enclosed otherwise ordinary functions in a scope, making them `static` members rather than ordinary member functions.

As far as it goes, this technique is OK, but it doesn't actually go that far. Yes, it does reduce the chance of name collision, especially between constants, such as enumerators. And yes, the use of the scope-resolution operator is a visibly distinct separator between the name of the scope and the name that is being enclosed. However, to use any enclosed name always requires the fully qualified name. There is no discounted concession or reasonable short cut for frequent users. You wanted a distinct name, you got it and you're stuck with it. Perhaps more significantly, the enclosing scope is in practice limited to a single header file. A class scope is a closed construct: the scope cannot be extended to embrace new content and it cannot span multiple header files. This makes the technique useless for anything except simple modular wrapping.

Namespace, the final frontier

The C++ `namespace` construct was introduced to address all of the issues outlined so far, and potentially more. If you are already familiar with namespaces, the article has so far acted as a revision and reminder of the motivation. It has possibly also pointed out a couple of things that are not normally mentioned, such as the low priority that should be attached to anticipating function name collisions.

A namespace is in some ways like a class and in many ways not:

- A namespace acts to introduce a scope that contains features such as classes and functions. That scope can be opened and closed in any file.
- A class can have a private section, but the contents of a namespace are always public.
- Namespaces can nest only inside one another. They cannot be nested inside classes or functions.
- A namespace cannot have the same name as another type defined at the same level.
- Unlike a class, a namespace does not have a terminating semi-colon.
- A namespace that has no name is both formally and obviously an unnamed namespace. It allows you to keep names private to a translation unit, displacing (and deprecating) a similar role of `static`. Where `static` applies only to functions and data, an unnamed namespace applies also to types.
- The global namespace is the scope outside of any named or unnamed namespace. It is effectively the root and default namespace.
- Entities with linkage that are enclosed in a namespace will have their linkage name affected by the namespace. This means that you cannot simply take some headers for a precompiled library and wrap those headers in an alternative namespace: the result won't link because the original code was compiled with the identifiers in the global namespace.

Directory listings revisited

To demonstrate a couple of these features in practice, let's revisit the directory listing classes described in two previous columns[1, 2]. Two classes, `dir_stream` and `dir_iterator`, were defined. The common prefix seems to be crying out for some better factoring, and a namespace offers a suitable approach. In the header defining the stream we would have:

```
namespace dir
{
class stream
{
public:
      explicit stream(const std::string &);
      ~stream();
      operator const void *() const;
      stream &operator>>(std::string &);
      ...
};
}
```

And in the header for the iterator:

```
namespace dir
{
class stream;
class iterator : public std::iterator<
      std::input_iterator_tag, std::string>
{
public:
      iterator();
      explicit iterator(stream &);
      ~iterator();
      const std::string &operator*() const;
      const std::string &operator->() const;
      iterator &operator++();
      iterator operator++(int);
      bool operator==(const iterator &) const;
      bool operator!=(const iterator &) const;
      ...
};
}
```

The member function definitions appear in the same namespace. For instance, the iterator's postincrement operator defined in terms of the preincrement operator:

```
namespace dir
{
...
iterator iterator::operator++(int)
{
      iterator old = *this;
      ++*this;
      return old;
}
...
}
```

Notice that names declared in the same namespace can be referred to directly, without any scope resolution, when accessed elsewhere in the same namespace.

Using declarations to your advantage

There are a number of different approaches to using a name from a named namespace. The simplest is just using the fully qualified name. Alternatively, there is a `using` declaration that imports a specific name as a more local declaration, and then there is the more sweeping `using` directive that imports all the names from namespace for unqualified use:

```
using namespace std;

void list_dir(const string &name)
{
typedef ostream_iterator<string> out;
using dir::stream;

stream directory(name);
dir::iterator begin(directory), end;
copy(begin, end, out(cout, "\n"));
}
```

`using` declarations and directives can appear at either global or local scope. In the code example, a wholesale usage of the `std` namespace is made at the global level, allowing unqualified use of `std::string`, `std::ostream_iterator`, `std::copy` and `std::cout`. A `using` declaration imports `dir::stream` for local use, whereas `dir::iterator` is fully named for its use.

The usage in the fragment is contrived and more than a little inconsistent. It demonstrates syntax and basic concepts, but not necessarily best practice. There are different opinions as to what constitutes best practice. Some advocates always use fully qualified names and never employ using directives, whereas others see no harm in them. In spite of the variation, there are some valuable and general points that can be made.

Be consistent in how a particular namespace is pulled into code. For example, because the `std` namespace is effectively part of the common platform, some developers like to pull it all in with a `using` directive in their source files. This is fine, so long as it is done consistently. Inconsistently using a directive in one source file and fully qualifying in another serves no useful development purpose. As with other practices in software development, anything that's not useful should not be used.

Never use namespace `using` directives or declarations as a convenience in header files, even for namespace `std`. In the privacy of your own source file you can do what you want without affecting other source files. Placing a `using` directive in a header instantly floods any file that `#includes` it with a host of unwanted and unnecessary names, pre-empting any policy that file may have on namespace usage and roundly defeating the whole point of namespaces. This is a frequent bad habit in a lot of commercial code that I have seen, suggesting that even though the idea of namespaces is quite simple, some programmers can be even simpler. Therefore, always declare features in headers using fully qualified names. If a function definition appears in a header, because it is either inlined or templated, and fully qualified names are considered too cumbersome in the function body, place the `using` at the beginning of the function body, not outside it. Such careful scoping ensures that headers are properly self-contained.

What and how to name

This is all good, but when would you actually define namespaces in your own code? It is one thing to say you have a language mechanism, but quite another to say that you have a method that makes sense of it.

A reasonable enough starting point would be to consider defining a namespace when you might otherwise have been tempted to use a common prefix on identifiers. The `dir` namespace was an example of this.

Another reasonable starting point would be to consider a namespace when a class or `struct` seems only to have public members that are types, constants or `static` member functions. An exception to this is where the type in question defines a trait, a policy or both[3]. In the current language, namespaces cannot be templated or used as template parameters, so a `struct` is really the most suitable expression of such designs.

In search of stability

I will leave you with one final recommendation to consider: make sure that your namespace names are stable and long lived. If a name is not stable, it means that you will have to change the namespace name and therefore any code that depends on it when the name is no longer applicable, even though all the namespace contents are otherwise unaffected. An architecture should be structured so that stable concepts depend on more stable, not less stable, concepts.

What, specifically, have I got in mind? Organisational structure is the main offender. What at first seems like a good idea—enclosing your code in a namespace that corresponds to your company or department—turns out to be quite the opposite. It is a superficially good idea that lacks any sound reasoning or depth when you scratch the surface.

Department or division partitioning is generally far less stable than code for an installed system. Companies reorganise themselves internally whenever they need to be seen to be doing something by their customers or shareholders, or simply when they're bored. The reorganisation may be as simple as renaming a few departments or it may involve relocations, regional restructuring and redundancies. Experience suggests that such events are more frequent than they are effective, but whichever way you look at it, an organisation's internal structure is clearly not a bellwether of stability.

Likewise, anyone who has been in software development for more than a few minutes will know that company names are as unstable as transuranic elements. Companies merge, demerge, spin-off, acquire, rebrand and debrand, add a dot-com suffix or remove a dot-com suffix with a regularity that keeps industry commentators employed, but does little for anyone else. Even worse is to base a namespace on a company's domain name, an even less stable concept. Source code is more likely to outlive any specific incarnation of its sponsor. So, choose cautiously and wisely; namespaces are easier in theory than in practice. ■

References/bibliography:

1. Kevlin Henney, "Promoting polymorphism", *Application Development Advisor*, October 2001
2. Kevlin Henney, "The perfect couple", *Application Development Advisor*, November-December 2001
3. Kevlin Henney, "Flag waiving", *Application Development Advisor*, April 2002

Kevlin Henney is an independent software development consultant and trainer. He can be reached at www.curbralan.com

ADA's free e-mail newsletter for software developers

By now many of you will have had a free on-line newsletter which is a service to readers of ADA and packed with the very latest editorial for software and application developers. However, some of our readers don't get this free service.

If you don't get your free newsletter it's for one of two good reasons.

1. Even though you subscribed to ADA, you haven't supplied us with your e-mail address.
2. You ticked the privacy box at the time you subscribed and we can't now send you our newsletter.

Sign up for this free service at www.appdevadvisor.co.uk/subscribe.htm

www.appdevadvisor.co.uk/subscribe.htm

SIGS Application Development ADVISOR