IB9N7 C++ for Quantitative Finance

Lecture 7n: Namespaces

J.F.Reizenstein@warwick.ac.uk

23 November 2015
(Week 8)

Outline

In this (part of the) lecture, we'll cover the following topic:

- Namespaces.

Namespaces

So far, every function that we have defined has been part of the global namespace. But we have also *used* many entities from the `std` namespace.

Namespaces

Namespaces

Namespaces provide a method of grouping related variables, functions and classes (**entities**).

Allows division of the main scope of a project into sub-scopes, and hence (crucially) helps prevent name collisions.

Namespaces are generally useful for large projects, situations where there are many programmers, and external libraries.

The standard library is in the `std` namespace, so in any event, need to understand them.

Use of namespaces

Syntax for namespaces

Give it a name, and define entities within it!

```
namespace ‹my_namespace›
{
    ‹entities›
}
```

(Namespaces can also have no name; these are special and are called anonymous namespaces.)

(Not part of a `namespace` block $\implies$ global namespace.)

Note that namespaces can be nested! Can also later resume (add to) an earlier `namespace` block.

Accessing the entities within a namespace

- Within a ‹`my_namespace`› namespace block, identifiers also from ‹`my_namespace`› can be used exactly as they are declared.
- Outside of such a block, the *scope resolution operator* `::` must be used to access elements in the namespace; ‹`my_namespace`›`::` must be prefixed to identifiers, e.g. `std::cout`.
- Entities in the global namespace can be accessed using `::identifier`.
- Functions with the same name can co-exist in different namespaces. Typically, in a large project, there might be a namespace for each team or type of functionality.

Namespace aliases

Namespace aliases can also be declared. The syntax for this is

```
namespace ‹new_namespace_name› =
    ‹old_namespace_name›;
```


Namespaces: an example

Example

```
#include <iostream>
#include <cstdlib>

namespace first { double bar() { return 1.0; } }
namespace second { int bar() { return 2; } }
namespace third { int bar() { return 5; } }
namespace first { double foo() { return 8.0; } } // resumes

int main()
{
    std::cout << first ::bar() << std::endl;
    std::cout << second::bar() << std::endl;
    std::cout << third ::bar() << std::endl;
    return EXIT_SUCCESS;
}
```

The `using` keyword

Repeatedly using the scope resolution operator can become tedious when accessing elements from frequently used namespaces.

- The **`using`** keyword can be used to give **declarations** and **directives** to the compiler, to import entities into the current scope or to add to the lookup tree, and may save you some typing.
- **`using`** can be used outside of any functions, or within functions etc.
- However, they can cause confusion (the directive form especially)!

Namespace declarations

Namespace declarations

`using` ‹**`namespace_name`**›**`::`**‹**`identifier`**›**`;`**

Imports the named entity into the current scope.

Only imports the entities you need, not every entity in the namespace.

(There are many entities in the **`std`** namespace, so importing all of it may cause particular trouble.)

Less confusing, but can still cause namespace pollution if used excessively.

Example: global namespace declaration

Example in global scope

```
1  #include <iostream>
2  #include <cstdlib>
3  using std::cout;
4  using std::endl;
5
6  int main()
7  {
8      cout << "Hello world" << endl;
9      return EXIT_SUCCCESS;
10 }
```

Better to use only at local scope if at all (see next slide).

Example: local namespace declaration

Example in local scope

```
1  #include <iostream>
2  #include <cstdlib>
3
4  int main()
5  {
6      using std::cout;
7      using std::endl;
8      cout << "Hello world" << endl;
9      return EXIT_SUCCCESS;
10 }
```

Namespace directives

Namespace directives

```
using namespace ‹namespace_name›;
```

Does not import entities within the namespace into the current scope, but adds the namespace to the lookup search tree.

Warning! Namespace directives at global scope are often considered **evil** in large programs. Avoid!

Namespace directives are especially evil in header files at global scope because it pollutes every client that does the `#include`.

Example namespace directive

Example in global scope (bad in large programs)

```
1 #include <iostream>
2 #include <cstdlib>
3 using namespace std;
4
5 int main()
6 {
7     cout << "Hello␣world" << endl;
8     return EXIT_SUCCESS;
9 }
```


Example namespace directive

Example in local scope

```
1 #include <iostream>
2 #include <cstdlib>
3
4 int main()
5 {
6     using namespace std;
7     cout << "Hello␣world" << endl;
8     return EXIT_SUCCESS;
9 }
```

Less evil than at global scope.

Ambiguity (1/3) ☆

If there is a conflicting entity in the current scope, cannot import.

Example

```
1 float bar() { return 7.0f; }
2 using first::bar; // error (because signatures are the same)
```

Ambiguity (2/3) ☆

If there is an imported entity in the current scope, cannot define conflicting entity.

Example

```
using first::bar;
float bar() { return 7.0f; } // error (because signatures are the same)
```


Ambiguity (3/3) ☆

However, it is not an error to import entities in such a way that the imported entities conflict with each other in the current scope.

Example

```
using first::bar;
using first::bar; // fine, just a redeclaration (no conflict)
using second::bar; // fine
using third::bar; // fine
```

In this situation, just as in function overloading, compiler sometimes gives an error if trying to **use** an identifier that is ambiguous.

Then have to qualify such individual cases with scope resolution operator.

Again, sometimes the compiler may choose a match without alerting you to the potential ambiguity.

`main` function

The **`main`** function must be part of the global namespace in order for the linker to identify it as the program entry point.

Importing the **`main`** function into the global scope does not import in such a way that it can be found by the linker.

Unnamed namespaces

Anonymous namespaces also protect against name clashes and external use.

Anonymous namespace

```
namespace
{
    ‹entities›
}
```

The compiler assigns a **unique name** to anonymous namespaces, local to a translation unit.

It also behaves as if you have

```
using namespace ‹that unique name›;
```

in the translation unit, so you can actually access the identifiers in the namespace (this is not evil).

Unnamed namespaces

Anonymous namespaces in a cpp file are good for two things.

- They tell humans that these functions (etc.) are only being used by other functions in the file, and nowhere else in the program. This helps to signal what they are for.
- They tell the compiler that these functions are only used in this translation unit. This often means that they do not need to be made available to the linker as proper functions. Certain optimisations may then happen, for example inlining.