6.12 — Using declarations and using directives

ALEX OCTOBER 4, 2021

You've probably seen this program in a lot of textbooks and tutorials:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!";

    return 0;
}
```

Some older compilers will also start new projects with a similar program.

If you see this, run. Your textbook, tutorial, or compiler are probably out of date. In this lesson, we'll explore why.

A short history lesson

Back before C++ had support for namespaces, all of the names that are now in the `std` namespace were in the global namespace. This caused naming collisions between program identifiers and standard library identifiers. Programs that worked under one version of C++ might have a naming conflict with a newer version of C++.

In 1995, namespaces were standardized, and all of the functionality from the standard library was moved out of the global namespace and into namespace `std`. This change broke older code that was still using names without `std::`.

As anyone who has worked on a large codebase knows, any change to a codebase (no matter how trivial) risks breaking the program. Updating every name that was now moved into the `std` namespace to use the `std::` prefix was a massive risk. A solution was requested.

Fast forward to today -- if you're using the standard library a lot, typing `std::` before everything you use from the standard library can

become repetitive, and in some cases, can make your code harder to read.

C++ provides some solutions to both of these problems, in the form of `using statements`.

But first, let's define two terms.

Qualified and unqualified names

A name can be either qualified or unqualified.

A **qualified name** is a name that includes an associated scope. Most often, names are qualified with a namespace using the scope resolution operator (::). For example:

```
1 std::cout // identifier cout is qualified by namespace std
  ::foo // identifier foo is qualified by the global namespace
```

For advanced readers

A name can also be qualified by a class name using the scope resolution operator (::), or by a class object using the member selection operators (. or ->). For example:

```
1 class C; // some class
2
3 C::s_member; // s_member is qualified by class C
  obj.x; // x is qualified by class object obj
  ptr->y; // y is qualified by pointer to class object ptr
```

An **unqualified name** is a name that does not include a scoping qualifier. For example, `cout` and `x` are unqualified names, as they do not include an associated scope.

Using declarations

One way to reduce the repetition of typing `std::` over and over is to utilize a `using declaration` statement. A **using declaration** allows us to use an unqualified name (with no scope) as an alias for a qualified name.

Here's our basic Hello world program, using a `using declaration` on line 5:

```
1 #include <iostream>
2
3 int main()
4 {
5    using std::cout; // this using declaration tells the compiler that cout should resolve to std::cout
6    cout << "Hello world!"; // so no std:: prefix is needed here!
7
8    return 0;
9 } // the using declaration expires here
```

The `using declaration` `using std::cout;` tells the compiler that we're going to be using the object `cout` from the `std namespace`. So whenever it sees `cout`, it will assume that we mean `std::cout`. If there's a naming conflict between `std::cout` and some other use of `cout`, `std::cout` will be preferred. Therefore on line 6, we can type `cout` instead of `std::cout`.

This doesn't save much effort in this trivial example, but if you are using `cout` many times inside of a function, a `using declaration` can make your code more readable. Note that you will need a separate `using declaration` for each name (e.g. one for `std::cout`, one for `std::cin`, etc...).

Although this method is less explicit than using the `std::` prefix, it's generally considered safe and acceptable (when used inside a function).

Using directives

Another way to simplify things is to use a `using directive`. A **using directive** imports all of the identifiers from a namespace into the scope of the `using directive`.

Here's our Hello world program again, with a `using directive` on line 5:

```
1  #include <iostream>
2
3  int main()
4  {
5     using namespace std; // this using directive tells the compiler to import all names from namespace
   std into the current namespace without qualification
6     cout << "Hello world!"; // so no std:: prefix is needed here
7     return 0;
8  }
```

The `using directive` `using namespace std;` tells the compiler to import *all* of the names from the `std namespace` into the current scope (in this case, of function `main()`). When we then use unqualified identifier `cout`, it will resolve to the imported `std::cout`.

`Using directives` are the solution that was provided for old pre-namespace codebases that used unqualified names for standard library functionality. Rather than having to manually update every unqualified name to a qualified name (which was risky), a single `using directive` (of `using namespace std;`) could be placed at the top of the each file, and all of the names that had been moved to the `std` namespace could still be used unqualified.

Problems with using directives (a.k.a. why you should avoid "using namespace std;")

In modern C++, `using directives` generally offer little benefit (saving some typing) compared to the risk. Because using directives import *all* of the names from a namespace (potentially including lots of names you'll never use), the possibility for naming collisions to occur increases significantly (especially if you import the `std` namespace).

For illustrative purposes, let's take a look at an example where `using directives` causes ambiguity:

```
#include <iostream>

namespace a
{
 int x{ 10 };
}

namespace b
{
 int x{ 20 };
}

int main()
{
 using namespace a;
 using namespace b;

 std::cout << x <<
'\n';

 return 0;
}

```

In the above example, the compiler is unable to determine whether the `x` in `main` refers to `a::x` or `b::x`. In this case, it will fail to compile with an "ambiguous symbol" error. We could resolve this by removing one of the `using` statements, employing a `using declaration` instead, or qualifying `x` with an explicit scope qualifier (`a::` or `b::`).

Here's another more subtle example:

```
#include <iostream> // imports the declaration of std::cout

int cout() // declares our own "cout" function
{
    return 5;
}

int main()
{
    using namespace std; // makes std::cout accessible as "cout"
    cout << "Hello, world!"; // uh oh!  Which cout do we want here?  The one in the std namespace or the
one we defined above?

    return 0;
}
```

In the above example, the compiler is unable to determine whether our use of `cout` means `std::cout` or the `cout` function we've defined, and again will fail to compile with an "ambiguous symbol" error. Although this example is trivial, if we had explicitly prefixed `std::cout` like this:

```
std::cout << "Hello, world!"; // tell the compiler we mean
std::cout
```

or used a `using declaration` instead of a `using directive`:

```
using std::cout; // tell the compiler that cout means
std::cout
cout << "Hello, world!"; // so this means std::cout
```

then our program wouldn't have any issues in the first place. And while you're probably not likely to write a function named "cout", there are hundreds, if not thousands, of other names in the std namespace just waiting to collide with your names.

Even if a `using directive` does not cause naming collisions today, it makes your code more vulnerable to future collisions. For example, if your code includes a `using directive` for some library that is then updated, all of the new names introduced in the updated library are now candidates for naming collisions with your existing code.

There is a more insidious problem that can occur as well. The updated library may introduce a function that not only has the same name, but is actually a better match for some function call. In such a case, the compiler may decide to prefer the new function instead, and the behavior of your program will change unexpectedly.

Consider the following program:

foolib.h:

```
namespace foo
{
    // pretend there is some useful code that we use
here
}
```

main.cpp:

```
#include <iostream>
#include <foolib.h>

int someFcn(double)
{
    return 1;
}

int main()
{
    using namespace foo; // Because we're lazy and want to access foo:: qualified names without typing
the foo:: prefix
    std::cout << someFcn(0); // The literal 0 should be 0.0, but this is an easy mistake to make

    return 0;
}
```

This program runs and prints `1`.

Now, let's say we update the foolib library, which includes an updated foolib.h. Our program now looks like this:

foolib.h:

```
namespace foo
{
    // newly introduced function
    int someFcn(int)
    {
        return 2;
    }

    // pretend there is some useful code that we use here
}
```

main.cpp:

```
#include <iostream>
#include <foolib.h>

int someFcn(double)
{
    return 1;
}

int main()
{
    using namespace foo; // Because we're lazy and want to access foo:: qualified names without typing the foo:: prefix
    std::cout << someFcn(0); // The literal 0 should be 0.0, but this is an easy mistake to make

    return 0;
}
```

Our `main.cpp` file hasn't changed at all, but this program now runs and prints `2`!

When the compiler encounters a function call, it has to determine what function definition it should match the function call with. In selecting a function from a set of potentially matching functions, it will prefer a function that requires no argument conversions over a function that requires argument conversions. Because the literal `0` is an integer, C++ will prefer to match `someFcn(0)` with the newly introduced `someFcn(int)` (no conversions) over `someFcn(double)` (requires a conversion from int to double). That causes an unexpected change to our program results.

This would not have happened if we'd used a `using declaration` or explicit scope qualifier.

The scope of using declarations and directives

If a `using declaration` or `using directive` is used within a block, the names are applicable to just that block (it follows normal block scoping rules). This is a good thing, as it reduces the chances for naming collisions to occur to just within that block.

If a `using declaration` or `using directive` is used in the global namespace, the names are applicable to the entire rest of the file (they have file scope).

Cancelling or replacing a using statement

Once a `using statement` has been declared, there's no way to cancel or replace it with a different `using statement` within the scope in which it was declared.

```
int main()
{
    using namespace foo;

    // there's no way to cancel the "using namespace foo" here!
    // there's also no way to replace "using namespace foo" with a different using statement

    return 0;
} // using namespace foo ends here
```

The best you can do is intentionally limit the scope of the `using statement` from the outset using the block scoping rules.

```
int main()
{
    {
        using namespace foo;
        // calls to foo:: stuff here
    } // using namespace foo expires

    {
        using namespace Goo;
        // calls to Goo:: stuff here
    } // using namespace Goo expires

    return 0;
}
```

Of course, all of this headache can be avoided by explicitly using the scope resolution operator (::) in the first place.

Best practices for using statements

Avoid `using directives` (particularly `using namespace std;`), except in specific circumstances. `Using declarations` are generally considered safe to use inside blocks. Limit their use in the global namespace of a code file, and never use them in the global namespace of a header file.

Best practice

Prefer explicit namespaces over `using statements`. Avoid `using directives` whenever possible. `Using declarations` are okay to use inside blocks.

Related content

The `using` keyword is also used to define type aliases, which are unrelated to using statements. We cover type aliases in lesson 8.6 -- Typedefs and type aliases.

Next lesson

6.13 Unnamed and inline namespaces

Back to table of contents

Previous lesson

6.11 Scope, duration, and linkage summary

DP N N FOUT

Newest