6.12 — Using statements

BY ALEX ON NOVEMBER 9TH, 2016 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2020

If you're using the standard library a lot, typing `std::` before everything you use from the standard library can become repetitive. C++ provides some alternatives to simplify things, called `using statements`.

Using declarations

One way to simplify things is to utilize a **using declaration** statement.

Here's our Hello world program, with 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 st
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 you use (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).

The using directive

Another way to simplify things is to use a **using directive** statement. 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 that we're using everything
6     cout << "Hello world!"; // so no std:: prefix is needed here!
7     return 0;
8  }
```

The `using directive` `using namespace std;` tells the compiler that we want to use *everything* in the `std namespace`, so if the compiler finds a name it doesn't recognize, it will check the `std namespace`. Consequently, when the compiler encounters `cout` (which it won't recognize), it'll look in the `std namespace` and find it there. If there's a naming conflict between `std::cout` and some other use of `cout`, the compiler will flag it as an error (rather than preferring one instance over the other).

For illustrative purposes, let's take a look at an example where a `using directive` 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`, or using an explicit `a::` or `b::` prefix with variable `x`.

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 namesp

    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.

Limiting the scope of using declarations and directives

If a `using declaration` or `using directive` is used within a block, the `using statement` applies only within that block (it follows normal block scoping rules). This is a good thing, as it reduces the chances for naming collisions to occur just within that block. However, many new programmers put `using directives` into the global scope. This pulls all of the names from the namespace directly into the global scope, greatly increasing the chance for naming collisions to occur (see lesson **2.9 -- Naming collisions and an introduction to namespaces** for more information on this).

Best practice

Although many textbooks and tutorials use them liberally, avoid `using directives` altogether. `Using declarations` are okay to use inside blocks, where their impact is limited, but not in the global 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.

6.13 -- Typedefs and type aliases

Index

6.11 -- Scope, duration, and linkage summary

C++ TUTORIAL | PRINT THIS POST

104 comments to 6.12 — Using statements

« Older Comments 1 2

Josh
December 18, 2019 at 5:58 am · Reply

"Rule: Avoid “using” statements outside of a function (in the global scope)."

and

"Suggestion: We recommend you avoid “using directives” entirely."

aren't in those nice colorful windows.

nascardriver
December 18, 2019 at 6:58 am · Reply

Lesson colored, thanks!

Charan
December 14, 2019 at 8:08 am · Reply

Hey,
So, the conflict in the more subtle example occurs because the global namespace is declared by default by our program(or compiler?who does that by the way?) ,isn't it?As if the compiler has to look into both name spaces. I don't know if it's the case but isn't it more 'tactical' if the compiler looks into the std namespace first and then into the global namespace?

Arionne Coleman
June 28, 2019 at 4:55 pm · Reply

I thought you needed parenthesis to call a function, why would cout confuse the compiler instead of recognizing that it is using the std namespace and not the user-defined function?

nascardriver
June 30, 2019 at 9:07 am · Reply

You need parenthesis to call a function, but you can use functions without calling them.
The example can never work if `cout` is a function, but the compiler doesn't detect it.

Red Lightning
April 21, 2019 at 6:05 am · Reply

The stupidest thing I can think of is a programmer making a namespace, and then putting a "using directive" in the global scope of his program, which results in the same chances for naming collisions as if he didn't even make the namespace.

```
#include <iostream>
namspace abc
{
    constexpr int someVariable{ 1234567890 };
}

using namespace abc;

int main ()
{
    char abc{ 34 };
    std::cout << abc;//I think it will work fine
}
```

Compiling...and...error.

Jeroen P. Broks
February 17, 2019 at 11:38 am · Reply

Now the syntax of using "using" in a scope block is a bit confusing to me.
Maybe also because the closest Pascal variant I ever saw is more logical:

```
x.h1:=1;
x.h2:=2;
with x do
Begin
    h1:=1;
    h2:=2
end;
```

Thanks to 'width' the h1 and h2 are part of x.... Now the "using" variant C++ offers wants it INSIDE the block....

```
{
   using namespace blahblah;
   // code
}
```

In stead of:

```
using namespace blahblah {
   // code
```

```
}
```

I guess enabling "using" in the global scope could be the most logical explanation for this (which "if" or "while" can't do), but still it baffles me... This syntax could haunt me, hahaha ;)

Now I wonder....

```
#include <iostream>
namespace jeroen {

     void helloworld(){
         std::cout << "Hello World\n";
     }

     void hi(){
         helloworld(); // does this work, since it's part of the same namespace? Or must I
     }
}
```

nascardriver
February 18, 2019 at 3:10 am · Reply

Namespace members can access other members of the same namespace without "jeroen::".

Alex
February 18, 2019 at 7:59 pm · Reply

Pascal's "with/do" is designed to allow you to quickly access the fields/members of a given record/struct.

C++'s "using" is designed to provide access to all declarations and definitions within a given namespace without the namespace qualifier.

Totally different uses, hence, different syntaxes.

Jules
February 10, 2019 at 12:36 am · Reply

In this 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;
}

How can you call a function (cout, the user defined one) without its arguments?, in this case empty ().
Woudnt the compiler throw an error at you?

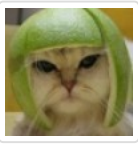
nascardriver
February 10, 2019 at 11:11 am · Reply

Hi Jules!
The compiler will error out. It doesn't check the types of @cout and @std::cout. It notices that there are 2 functions/variables with the same name and stops.
If you were to remove the "using namespace std;" line, there'd still be an error (Because cout is missing "()")

Benur21
January 18, 2019 at 2:43 pm · Reply

Please, can you put all "using" words that you mean the "using" statement between quotes (or italics)? I couldn't understand a few sentences in the first place because of this.

PS: Or if you mean "using" not as a statement, but as a "using declaration", for example, put them also between quotes. (look at the titles)



Alex
January 21, 2019 at 5:28 pm · Reply

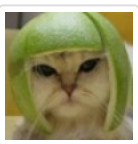
I italicized the various using-terms, so they're easier to read in context. Thanks for the suggestion.

Suprith
July 9, 2018 at 11:37 pm · Reply

#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;
}
My doubt is , in the above example, cout should suppose to resolve to std:cout. Why it'll conflict with int cout().



Alex
July 11, 2018 at 9:06 am · Reply

Nope. using namespace std tells the compiler "std::cout" should be referenceable as "cout". Thus, when we call "cout", the compiler doesn't know whether we mean std::cout or our own cout function.

If you want std::cout to take precedence over our own cout inside this function, you should use a using declaration (e.g. using std::cout). Then when the compiler sees cout, it will know that std::cout should take precedence.

In other words, using declarations (e.g. using std::cout) imply a precedence, using directives (e.g. using namespace std) do not.

Suprith
July 11, 2018 at 12:17 pm · Reply

Thank you so much.

saj
November 17, 2018 at 4:19 am · Reply

If using std::cout imply a precedence, How will I refer to my own cout() function?
With a global scope operator :: ?
And what if my cout() is declared in another .cpp file

Alex
November 18, 2018 at 3:05 pm · Reply

Yes, you can use namespace::cout() where namespace is whatever namespace your cout is in, or ::cout() if your cout is in the global namespace.

It doesn't matter if it's in another file, as long as the current use can see the forward declaration.

why
May 28, 2018 at 10:09 pm · Reply

HII ALEX ,MY doubt is here "cout" means that is in iostream where cout() this is another function .so it wont give compile error.

```
#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;
}
```

nascardriver
May 29, 2018 at 7:05 am · Reply

Hi why!

Just try to compile it yourself, it doesn't work. Don't use 'using namespace' and give your functions unique names and you'll be fine.

Q
March 7, 2018 at 2:50 am · Reply

Hi Alex,

In the most recent version of Code::Blocks (17.12), the statement using namespace std is standard for every new file or project you create.
If it really is such bad practice to use a using directive statement with global scope, then why would a IDE like Code::Blocks implement it in its standard template (with global scope)?
Wouldn't it suffice if I refrain from naming any of my objects after anything in the std library?
Below an example of the standard piece of code you get every time you create a project:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

On a unrelated note, thanks for the tutorial. It really is one of the best ones out there.
Cheers.

nascardriver
March 7, 2018 at 3:08 am · Reply

Hi Q!

> why would a IDE like Code::Blocks implement it in its standard template (with global scope)?
I don't know. A lot of IDEs do this. You can change the new project template in Code::Blocks to your liking.

> Wouldn't it suffice if I refrain from naming any of my objects after anything in the std library?
You should definitely do this. But how do you know none of the other code you're using was written with naming conflicts in mind? eg. min and max are common names that cause conflicts from time to time.

Q
March 7, 2018 at 3:34 am · Reply

Hello nascardriver!

> But how do you know none of the other code you're using was written with naming conflicts in mind? eg. min and max are common names that cause conflicts from time to time.
This is a very good point, did not think of this.
I will just use the scope resolution operator or use a using namespace std in block scope if it makes the code more readable.

Thank you for replying so fast!

Alex
March 7, 2018 at 7:07 am · Reply

A few thoughts here:
1) Yes, it really is bad practice to use "using namespace std" in the global scope (it's more okay at function scope if used judiciously, but personally I avoid it altogether). I'm surprised Code::Blocks does this by default, and I'm not sure what the rotionale for doing so would be.

2) While your code might not have a naming conflict now, you can't guarantee it won't with future extensions of the language! So code that compiles today might not compile once you've upgraded your compiler. Using the explicit qualifier avoids these kinds of issues.

Q
March 9, 2018 at 3:24 am · Reply

Thank you for your reply and clearing this up, no idea why C::B would do this by default either. I'll just be using std:: from now on. Thanks :)

« Older Comments 1 2