Why is “using namespace std” considered bad practice?

I've been told by others on numerous occasions that my teacher's advice of exercising `using namespace std` in code was wrong. Hence, we should use `std::cout` and `std::cin`.

Why is `using namespace std` considered bad practice? Is it really that inefficient or risk declaring ambiguous variables (variables that share the same name as a function in `std` namespace)? Or, does this impact performance?

c++ namespaces std c++-faq

edited Sep 8 at 13:24
Rakete1111
12.3k 5 27 60

asked Sep 21 '09 at 3:08
akbiggs
6,873 3 14 33

14 The Google C++ Style Guide doesn't really answer your question but has a lot of general "why"s: google-styleguide.googlecode.com/svn/trunk/… – a paid nerd Sep 21 '09 at 3:17

251 Don't forget you can do: "using std::cout;" which means you don't have to type std::cout, but don't bring in the entire std namespace at the same time. – Bill Sep 21 '09 at 15:29

61 @Bill How would that work? `<< "Hello, World!";` ? :P – Mateen Ulhaq Jul 25 '11 at 23:29

27 @muntoo Heh. You were joking, but just in case, he meant you could type cout instead of typing std::cout. – akbiggs Jul 26 '11 at 1:06

24 Sounds more like a Valgrind bug, as "using namespace" has no effect of that kind... – MFH Jul 16 '12 at 6:43

31 Answers

1 2 next

This is not related to performance at all. But consider this: you are using two libraries called Foo and Bar:

```
using namespace foo;
using namespace bar;
```

Everything works fine, you can call `Blah()` from Foo and `Quux()` from Bar without problems. But one day you upgrade to a new version of Foo 2.0, which now offers a function called `Quux()`. Now you've got a conflict: Both Foo 2.0 and Bar import `Quux()` into your global namespace. This is going to take some effort to fix, especially if the function parameters happen to match.

If you had used `foo::Blah()` and `bar::Quux()`, then the introduction of `foo::Quux()` would have been a non-event.

edited Jun 14 at 16:37
Stephane Bersier
101 7

answered Sep 21 '09 at 3:13
Greg Hewgill
524k 109 893 1054

186 I've always liked Python's "import big_honkin_name as bhn" so you can then just use "bhn.something" rather than "big_honkin_name.something"- really cuts down on the typing. Does C++ have something like that? – paxdiablo Sep 21 '09 at 3:18

435 @Pax namespace io = boost::filesystem; – AraK Sep 21 '09 at 3:19

81 I think it's overstating things to say it's "some effort to fix". You'll have no instances of the new foo::Quux so just disambiguate all your current uses with bar::Quux. – MattyT Sep 21 '09 at 13:44

152 Would any sensible person create a library with types whose unqualified name collide with the std types? – erikkallen Sep 21 '09 at 19:14

53 @TomA: The problem with `#define` is that it doesn't restrict itself to namespaces, but tramples over the whole code base. A namespace alias is what you want. – sbi Sep 25 '09 at 8:28

I agree with everything Greg wrote, but I'd like to add: ***It can even get worse than Greg said!***

Library Foo 2.0 could ***introduce a function***, `Quux()`, that is ***an unambiguously better match*** for some of your calls to `Quux()` than the `bar::Quux()` your code called for years. Then your ***code still compiles***, but ***it silently calls the wrong function*** and does god-knows-what. That's about as bad as things can get.

Keep in mind that the `std` namespace has tons of identifiers, many of which are *very* common ones (think `list`, `sort`, `string`, `iterator`, etc.) which are very likely to appear in other code, too.

If you consider this unlikely: ***There was a question asked*** here on Stack Overflow where pretty much exactly this happened (wrong function called due to omitted `std::` prefix) about half a year after I gave this answer. ***Here is another***, more recent ***example*** of such a question. So ***this is a real problem***.

Here's one more data point: Many, many years ago, I also used to find it annoying having to prefix everything from the standard library with `std::`. Then I worked in a project where it was decided at the start that both `using` directives and declarations are banned except for function scopes. Guess what? It took most of us very few weeks to get to used to write the prefix and after a few more weeks most of us even agreed that it actually made the code *more readable*. (There's a reason for that: ***Whether you like shorter or longer prose is subjective, but the prefixes objectively add clarity to the code.*** Not only the compiler, but you, too, find it easier to see which identifier is referred to.)

In a decade, that project grew to have several million lines of code. Since these discussions come up again and again, I once was curious how often the (allowed) function-scope `using` actually was used in the project. I grep'd the sources for it and only found one or two dozen places where it was used. To me this indicates that, ***once tried, developers didn't find `std::` painful enough*** to employ using directives even once every 100 kLoC *even where it was allowed to be used*.

Bottom line: Explicitly prefixing everything doesn't do any harm, takes very little getting used to, and has objective advantages. In particular, it makes the code easier to interpret by the compiler and by human readers — and that should probably be the main goal when writing code.

edited Oct 11 '15 at 10:26
Peter Mortensen
10.5k 13 72 108

answered Sep 21 '09 at 9:26
sbi
138k 36 182 355

68 It does significantly harm the density of code you can pack in a single line. You end up writing your code in a very long-winded way; which reduces readability. Personally, I think shorter (but not too short) code tends to be more readable (since there is less stuff to read, and less stuff to get distracted about). – Lie Ryan Oct 28 '10 at 15:13

47 Guess you missed out on the old days before C++ had a standard `string` class, and seemingly every library had their own. Tell you what: We'll keep writing our code with `std::`, and you can run our code through `grep -v std:: | vim` when you're browsing it. Or you can teach your editor that `std::` is a keyword which is to be colored the same as the background color. Whatever works. – Mike DeSimone Oct 28 '10 at 17:31

47 I don't think `std::` is harmful at all. It carries very important information (namely "whatever comes after is part of the standard library", and it is still a pretty short and compact prefix. Most of the time, it's no problem at all. Sometimes, you have a few lines of code where you need to refer to specific symbols in the `std` namespace a lot, and then a `using` statement in that particular scope solves the problem nicely. But in the general case, it's not noise, it conveys valuable information *in addition* to removing ambiguities. – jalf Oct 28 '10 at 18:39

81 Whenever I see `std::`, I know it's going to be from `std::` without having to think about it. If I see `string` or `list` or `map` by themselves, I wonder a bit. – Mateen Ulhaq Sep 14 '11 at 6:50

17

@LieRyan Then good luck writing a geometry library without ever naming something `vector`, `transform` or `distance`. And those are just examples of the *many many very common names* used in the standard library. Suggesting not to use them out of fear or a biased opinion of the namespace feature that is an integral part of C++ is rather counter-productive. – Christian Rau Mar 7 '13 at 17:04

I think it's bad to put it in the header files of your classes: because then you would be forcing anyone who wants to use your classes (by including your header files) to also be 'using' (i.e. seeing everything in) those other namespaces.

However, you may feel free to put a using statement in your (private) *.cpp files.

Beware that some people disagree with my saying "feel free" like this -- because although a using statement in a cpp file is *better* than in a header (because it doesn't affect people who include your header file), they think it's still not *good* (because depending on the code it could make the implementation of the class more difficult to maintain). This FAQ topic says,

> The using-directive exists for legacy C++ code and to ease the transition to namespaces, but you probably shouldn't use it on a regular basis, at least not in your new C++ code.

It suggests two alternatives:

- A using declaration:

```
using std::cout; // a using-declaration lets you use cout without qualification
cout << "Values:";
```

- Get over it and just type std::

```
std::cout << "Values:";
```

edited Jun 20 at 21:39

answered Sep 21 '09 at 3:22

ChrisW

42.6k 6 72 160

15 It is less bad to put it in a .cpp file than in a header, but the problem stays the same for maintainability. It is taking the risk that two functions with the same name will clash when the code/the library you use/the C++ standard gets modified. – Étienne May 25 '13 at 13:05

2 Have you tried this at all? Because the using namespace directive will not carry over into another file. (GCC 4.8+) – zackery.fix Jan 16 at 6:53

3 @zackery.fix, "Apple LLVM version 7.0.2 (clang-700.1.81)" propagates `using namespace std;` from header to source file, and I verified that GCC does not. So at least having the "using" directive in header is risky. – Franklin Yu Jan 23 at 6:05

Yea.. I don't use LLVM or clang and this is not a standard approach anyway. – zackery.fix Jan 25 at 15:55

1 You really should not tell people to "feel free" with using namespace std in a .cpp file. @Étienne is right. – einpoklum Feb 24 at 16:39

I recently ran into a complaint about Visual Studio 2010. It turned out that pretty much all the source files had these two lines:

```
using namespace std;
using namespace boost;
```

A lot of Boost features are going into the C++0x standard, and Visual Studio 2010 has a lot of C++0x features, so suddenly these programs were not compiling.

Therefore, avoiding `using namespace X;` is a form of future-proofing, a way of making sure a change to the libraries and/or header files in use is not going to break a program.

edited Oct 11 '15 at 10:33

Peter Mortensen

10.5k 13 72 108

answered Oct 28 '10 at 17:37

David Thornley

45.6k 8 73 132

This. Boost and std have a *lot* of overlap - especially since C++11. – einpoklum Feb 24 at 16:40

One shouldn't use using directive at global scope, especially in headers. However there are situations where it is appropriate even in a header file:

```
template <typename FloatType> inline
FloatType compute_something(FloatType x)
{
    using namespace std; //no problem since scope is limited
    return exp(x) * (sin(x) - cos(x * 2) + sin(x * 3) - cos(x * 4));
}
```

This is better than explicit qualification (`std::sin` , `std::cos` ...) because it is shorter and has the ability to work with user defined floating point types (via Argument Dependent Lookup).

edited Dec 5 '10 at 16:32

Jonathan Sterling
13.8k 11 56 74

answered Sep 21 '09 at 15:47

robson3.14
1,715 10 16

5 I'm sorry, but I strongly disagree with this. – Billy ONeal Dec 5 '10 at 17:15

1 @Billy: There is no other way to support calling userlib::cos(userlib::superint). Every feature has a use. – Zan Lynx Jun 24 '11 at 23:42

9 @Zan: Of course there is. `using std::cos;` , `using std::sin` , etc. The issue though is that any well designed `userlib` is going to have their `sin` and `cos` inside their own namespace as well, so this really doesn't help you. (Unless there's a `using namespace userlib` before this template and that's just as bad as `using namespace std` -- and the scope there is not limited.) Furthermore, the only function like this I ever see this happen to is `swap` , and in such cases I would recommend just creating a template specialization of `std::swap` and avoiding the whole problem. – Billy ONeal Jun 24 '11 at 23:48

8 @BillyONeal: `template<typename T> void swap(MyContainer<T>&, MyContainer<T>&)` (There's no function template partial specialization (FTPS), so sometimes you need to resort to overloading instead. – sbi May 30 '12 at 14:56

12 @BillyONeal: Your (7-times-upvoted!) comment is wrong -- the situation you describe is *exactly* what ADL was designed to cover. Briefly, if `x` has one or more "associated namespaces" (e.g. if it was defined in `namespace userlib`) then any function call that looks like `cos(x)` will *additionally* look in those namespaces -- *without* any `using namespace userlib;` beforehand being necessary. Zan Lynx is right (and C++ name lookup is byzantine...) – j_random_hacker Sep 4 '15 at 15:31

Short version: don't use global using declarations or directives in header files. Feel free to use them in implementation files. Here's what Herb Sutter and Andrei Alexandrescu have to say about this issue in C++ Coding Standards (bolding for emphasis is mine):

> ## Summary
>
> Namespace usings are for your convenience, not for you to inflict on others: Never write a using declaration or a using directive before an #include directive.
>
> Corollary: In header files, don't write namespace-level using directives or using declarations; instead, explicitly namespace-qualify all names. (The second rule follows from the first, because headers can never know what other header #includes might appear after them.)
>
> ## Discussion
>
> In short: You can and should use namespace using declarations and directives liberally in your implementation files after #include directives and feel good about it. **Despite repeated assertions to the contrary, namespace using declarations and directives are not evil and they do not defeat the purpose of namespaces. Rather, they are what make namespaces usable**.

answered Nov 3 '14 at 20:00

mattnewport
9,017 1 20 33

18 Sad that this sane guidance is so buried under misguided answers. – nyholku Jun 11 '15 at 12:44

1 Just one more programmer's opinion here, but while I agree 100% with the statement that the word `using` should never appear in a header, I'm not as convinced about the free license to place `using namespace xyz;` anywhere in your code, especially if `xyz` is `std` . I use the `using std::vector;` form, since that only pulls a single element from the namespace into pseudo-global scope, therefore leading to far less risk of a collision. – dgnuff Feb 23 at 23:51

@Lightness Races in Orbit you are of course entitled to your opinion. Would have been more helpful if there had been some attempt at explanation why you do not agree with advice given in this answer. Especially would be interesting to understand what is the point of namespaces if 'using' them is bad? Why not just name things std_cout instead of std::cout ... the creators of C++/namespace must have had some idea when they bothered to create them. – nyholku Jun 20 at 13:36

@nyholku: No need - the majority of the other answers give the same reasons I would. Also please do no hesitate to note the ":)" I appended to my comment! And that I didn't say namespaces are bad. – Lightness Races in Orbit Jun 20 at 14:03

Yeah, I noticed that :) but IMO the majority of answer (that go against this sage advice) are misguided (not that I made any statistics what is the majority now). If you agree that namespace are 'not bad' then you might say where you think they are appropriate if you disagree with this answer? – nyholku Jun 20 at 16:54

If you import the right header files you suddenly have names like `hex`, `left`, `plus` or `count` in your global scope. This might be surprising if you are not aware that `std::` contains these names. If you also try to use these names locally it can lead to quite some confusion.

If all the standard stuff is in its own namespace you don't have to worry about name collisions with your code or other libraries.

edited Aug 31 '13 at 22:17 user283145

answered Sep 21 '09 at 3:23 sth 130k 34 207 313

5 +1 not to mention `distance`. still i prefer non-qualified names whereever practically possibility, since that increases readability for me. plus, i think the fact that we usually don't qualify things in oral speech, and are willing to spend time resolving possible ambiguities, means that it has value to be able to understand what one is talking about without qualifications, and applied to source code that means it's structured in such a way that it's clear what it's all about even without qualifications. – Cheers and hth. - Alf Dec 15 '12 at 5:49

To be fair, though, you don't have most of those if you don't include `<iomanip>`. Still, good point. – einpoklum Feb 24 at 16:42

Do not use it globally

It is considered "bad" only when **used globally**. Because

- you clutter the namespace you are programming in.
- readers will have difficulty seeing where a particular identifier comes from, when you use many `using namespace xyz`.
- whatever is true for *other* readers of your source code is even more true for the most frequent reader of it: yourself. Come back in a year or two and take a look...
- if you only talk about `using namespace std` you might not be aware of all the stuff you grab -- and when you add another `#include` or move to a new C++-revision you might get name conflicts you were not aware of

You may use it locally

Go ahead and use it locally (almost) freely. This, of course, prevents you from repetition of `std::` -- and repetition is also bad.

An idiom for using it locally

In C++03 there was an idiom -- boilerplate code -- for implementing a `swap` function for you classes. It was suggested that you actually use a local `using namespace std` -- or at least `using std::swap`:

```
class Thing {
    int    value_;
    Child  child_;
public:
    // ...
    friend void swap(Thing &a, Thing &b);
};
void swap(Thing &a, Thing &b) {
    using namespace std;      // make `std::swap` available
    // swap all members
    swap(a.value_, b.value_); // `std::stwap(int, int)`
    swap(a.child_, b.child_); // `swap(Child&,Child&)` or `std::swap(...)`
}
```

This does the following magic

- the compiler will choose the `std::swap` for `value_`, i.e. `void std::swap(int, int)`
- If you have an overload `void swap(Child&, Child&)` implemented the compiler will choose it
- If you do *not* have that overload the compiler will use `void std::swap(Child&,Child&)` and try its best swapping these.

With C++11 there is no reason to use this pattern anymore. The implementation of `std::swap` was changed to find a potential overload and choose it.

answered Jan 18 '13 at 9:34

towi
7,815 10 57 128

2 *"The implementation of std::swap was changed to find a potential overload and choose it."* - What? Are you sure about that? Though it is true that providing a custom `swap` in the first place isn't that much important in C++11 anymore, since the `std::swap` itself is more flexible (uses move semantics). But `std::swap` automatically chosing your own custom swap, that is absolutely new to me (and I don't really believe it). – Christian Rau Mar 7 '13 at 17:09

@ChristianRau I think so, yes. I read this on SO somewhere. We can always ask Howard, he should know. I am digging and digging now... – towi Mar 8 '13 at 8:44

@ChristianRau Cant find a definitive link right now (need to go back to work), but I got to n3490 and n1252 – towi Mar 8 '13 at 8:52

8 Even in the swap case, the clearer (and thankfully more common) idiom is to write `using std::swap;` rather than `using namespace std;` . The more specific idiom has fewer side effects and therefore makes the code more maintainable. – Adrian McCarthy Feb 27 '14 at 17:24

2 The final sentence is wrong. In C++11 the Std Swap Two Step was officially blessed as the *right* way to call `swap` , and various other places in the standard were changed to say they call `swap` like that (N.B. as stated above, `using std::swap` is the right way, not `using namespace std`). But `std::swap` itself was emphatically **not** changed to find some other `swap` and use it. If `std::swap` gets called, then `std::swap` gets used. – Jonathan Wakely Jul 16 '15 at 10:59

I agree that it should not be used globally, but it's not so evil to to use locally, like in a `namespace` . Here's an example from *"The C++ Programming Language"* :

```
namespace My_lib {

    using namespace His_lib; // everything from His_lib
    using namespace Her_lib; // everything from Her_lib

    using His_lib::String; // resolve potential clash in favor of His_lib
    using Her_lib::Vector; // resolve potential clash in favor of Her_lib

}
```

In this example, we resolved potential name clashes and ambiguities arising from their composition.

Names explicitly declared there (including names declared by using-declarations like `His_lib::String`) take priority over names made accessible in another scope by a using-directive (`using namespace Her_lib`).

answered Aug 29 '13 at 9:44

Oleksiy
8,628 11 42 89

Experienced programmers use whatever solves their problems and avoid whatever creates new problems. Thus they avoid header-file-level using-directives for obvious reason.

And they try to avoid full qualification of names inside their source files. A minor point is that it's not elegant to write more code when less code suffice *without good reason*. A major point is turning off ADL.

What are these *good reasons*? Sometimes you explicitly want turning off ADL. Sometimes you want to disambiguate.

So the following are OK:

1. Function-level using-directives and using-declarations inside functions' implementations
2. Source-file-level using-declarations inside source files
3. (Sometimes) source-file-level using-directives

edited Oct 11 '15 at 10:35

Peter Mortensen
10.5k 13 72 108

answered Mar 29 '11 at 8:10

Alexander Poluektov
5,057 15 21

2 Define "elegance". – sbi Mar 26 '15 at 3:22

I also consider it a bad practice. Why? Just one day I thought that function of a namespace is to divide stuff so I shouldn't spoil it with throwing everything into one global bag. However, if I often

use 'cout' and 'cin', I write: `using std::cout; using std::cin;` in cpp file (never in header file as it propagates with `#include`). I think that noone sane will ever name a stream `cout` or `cin`.;)

answered Sep 21 '09 at 9:34

Yelonek
278 2 9

4 That's a local using *declaration*, a very different thing from a using *directive*. – sbi May 26 '12 at 6:41

Another reason is surprise.

If I see `cout << blah`, instead of `std::cout << blah`

I think what is this 'cout'? Is it the normal cout? Is it something special?

edited Nov 18 '14 at 13:48

answered Sep 21 '09 at 3:13

Martin Beckett
70.8k 15 145 222

---

1. you need to be able to read code written by people who have different style and best practices opinions than you.
2. If you're only using cout, nobody gets confused. But when you have lots of namespaces flying around and you see this class and you aren't exactly sure what it does, having the namespace explicit acts as a comment of sorts. You can see at first glance, 'oh, this is a filesystem operation' or 'thats doing network stuff'.

edited Sep 21 '09 at 4:16

answered Sep 21 '09 at 4:04

Dustin Getz
11.3k 9 59 114

Consider

```
// myHeader.h
#include <sstream>
using namespace std;


// someoneElses.cpp/h
#include "myHeader.h"

class stringstream {  // uh oh
};
```

Note that this is a simple example, if you have files with 20 includes and other imports you'll have a ton of dependencies to go through to figure out the problem. The worse thing about it is that you can get unrelated errors in other modules depending on the definitions that conflict.

It's not horrible but you'll save yourself headaches by not using it in header files or the global namespace. It's probably alright to do it in very limited scopes but I've never had a problem typing the extra 5 characters to clarify where my functions are coming from.

edited Jan 16 '13 at 9:53

ulidtko
5,921 2 29 60

answered Sep 21 '09 at 3:19

Ron Warholic
8,656 20 43

It's all about managing complexity. Using the namespace will pull things in that you don't want, and thus possibly make it harder to debug (I say possibly). Using std:: all over the place is harder to read (more text and all that).

Horses for courses - manage your complexity how you best can and feel able.

edited Feb 25 '14 at 19:40

answered Sep 21 '09 at 3:14

Preet Sangha
47.9k 14 94 156

It's nice to see code and know what it does. If I see "std::cout" I know: That's the cout stream of the std library. If I see "cout" then I don't know. It *could* be the cout stream of the std library. Or there could be an "int cout = 0;" ten lines higher in the same function. Or a static variable named cout in that file. It could be anything.

Now take a million line code base, which isn't particularly big, and you're searching for a bug, which means you know there is one line in this one million lines that doesn't do what it is supposed to do. "cout << 1;" could read a static int named cout, shift it to the left by one bit, and throw away the result. Looking for a bug, I'd have to check that. Can you see how I really really prefer to see "std::cout"?

It's one of these things that seem a really good idea if you are a teacher and never had to write and maintain any code for a living. I love seeing code where (1) I know what it does and (2) I'm confident that the person writing it knew what it does.

edited Mar 26 '15 at 3:25 — sbi — **138k** 36 182 355

answered Mar 13 '14 at 17:22 — gnasher729 — **34.2k** 3 32 59

1 How do you know "std::cout << 1" isn't reading a static int named cout in std namespace shifting it by one and throwing away result? Also how do you know what "<<" does ;) ??? ... seems like this answer is not good data point to avoid 'using'. – nyholku Jun 11 '15 at 12:43

If someone has redefined std::cout to be an integer, then your problem isn't technical, but social -- someone has it in for you. (and you should probably also check all of the headers for things like #define true false, etc) – Jeremy Friesner Jul 7 at 4:33

Using many namespaces at the same time is obviously a recipe for disaster, but using JUST namespace `std` and only namespace `std` is not that big of a deal in my opinion because redefinition can only occur by your own code...

So just consider them functions as reserved names like "int" or "class" and that is it.

People should stop being so anal about it. Your teacher was right all along. Just use ONE namespace; that is the whole point of using namespaces the first place. You are not supposed to use more than one at the same time. Unless it is your own. So again, redefinition will not happen.

edited Oct 11 '15 at 10:44 — Peter Mortensen — **10.5k** 13 72 108

answered Nov 9 '13 at 15:09 — user2645752 — **87** 1 2

Creating collisions isn't that hard - short strings like `min` , `end` and `less` appear in the `std::` namespace. But more, now that `std::` has thousands of symbols in it, it's useful for the reader to know where a new symbol they might not know comes from. – Tom Swirly May 24 at 23:32

A namespace is a named scope. Namespaces are used to group related declarations and to keep separate items separate. For example, two separately developed libraries may use the same name to refer to different items, but a user can still use both:

```
namespace Mylib{
    template<class T> class Stack{ /* ... */ };
    / / ...
}
namespace Yourlib{
    class Stack{ /* ... */ };
    / / ...
}
void f(int max) {
    Mylib: :Stack<int> s1(max) ; / / use my stack
    Yourlib: :Stack s2(max) ; / / use your stack
    / / ...
}
```

Repeating a namespace name can be a distraction for both readers and writers. Consequently, it is possible to state that names from a particular namespace are available without explicit qualification. For example:

```
void f(int max) {
    using namespace Mylib; / / make names from Mylib accessible
    Stack<int> s1(max) ; / / use my stack
    Yourlib: :Stack s2(max) ; / / use your stack
    / / ...
}
```

Namespaces provide a powerful tool for the management of different libraries and of different versions of code. In particular, they offer the programmer alternatives of how explicit to make a

reference to a nonlocal name.

Source : An Overview of the C++ Programming Language by Bjarne Stroustrup

edited May 9 '15 at 22:26
Czipperz
767 1 5 17

answered Apr 5 '15 at 12:56
Rohan Singh
81 1 1

3 Very interesting that a this answer that is based on guidance from no other that Bjarne Stroustrup has earned -2... boy Bjarne must have been a poor and inexperienced programmer when he introduced this feature into C++ – nyholku Jun 11 '15 at 12:48

@nyholku: See this. – sbi Aug 4 '15 at 16:01

specifically what in *this*? – nyholku Aug 4 '15 at 21:48

I do not think it is necessarily bad practice under all conditions, but you need to be careful when you use it. If you're writing a library, you probably should use the scope resolution operators with the namespace to keep your library from butting heads with other libraries. For application level code, I don't see anything wrong with it.

answered Sep 21 '09 at 3:34
Dr. Watson
2,375 3 21 38

"Why is 'using namespace std;' considered a bad practice in C++?"

I put it the other way around: Why is typing 5 extra chars is considered cumbersome by some?

Consider e.g. writing a piece of numerical software, why would I even consider polluting my global namespace by cutting general "std::vector" down to "vector" when "vector" is one of the problem domain's most important concepts?

answered May 13 '13 at 15:18
Solkar
938 8 19

16 It's not just 5 extra chars; its 5 extra chars every time you reference any object type in the standard library. Which, if you're using the standard library very much, will be often. So it's more realistically thousands of extra chars in a decent sized program. Presumably the 'using' directive was added to the language so that it could be used... – Jeremy Friesner Sep 3 '13 at 3:42

2 Its not 5 extra chars every time, it is 5 chars and probably a couple mouse clicks to pull down a menu and do a Find and Replace in the editor of your choice. – DaveWalley Mar 6 '14 at 1:54

1 Readability. `cout << hex << setw(4) << i << endl;` is easier to read than `std::cout << std::hex << std::setw(4) << i << std::endl;` – oz1cz Oct 24 '14 at 9:47

11 And even worse: `std::map<std::string,std::pair<std::string,std::string>>` is horrible compared to `map<string,pair<string,string>>` . – oz1cz Oct 24 '14 at 10:41

It's a good practice is to typedef your STL containers anyway so std:: there really doesn't matter. And C++11 brought us the auto keyword which makes things even easier when e.g. using iterators. – juzzlin Jan 19 at 12:04

I agree with the others here, but would like to address the concerns regarding readability - you can avoid all of that by simply using typedefs at the top of your file, function or class declaration.

I usually use it in my class declaration as methods in a class tend to deal with similar data types (the members) and a typedef is an opportunity to assign a name that is meaningful in the context of the class. This actually aids readability in the definitions of the class methods.

```
//header
class File
{
   typedef std::vector<std::string> Lines;
   Lines ReadLines();
}
```

and in the implementation:

```
//cpp
Lines File::ReadLines()
{
    Lines lines;
```

```
    //get them...
    return lines;
}
```

as opposed to:

```
//cpp
vector<string> File::ReadLines()
{
    vector<string> lines;
    //get them...
    return lines;
}
```

or:

```
//cpp
std::vector<std::string> File::ReadLines()
{
    std::vector<std::string> lines;
    //get them...
    return lines;
}
```

answered Feb 12 '15 at 0:40

Carl
21k 6 51 81

Just a minor comment, while typedef is useful I'd consider making a class that represents Lines instead of using typedef. – Eyal Solnik Mar 11 at 9:23

With unqualified imported identifiers you need external search tools like *grep* to find out where identifiers are declared. This makes reasoning about program correctness harder.

answered Apr 11 '13 at 14:22

August Karlstrom
3,681 1 20 33

To answer your question I look at it this way practically: a lot of programmers (not all) invoke namespace std. Therefore one should be in the habit of NOT using things that impinge or use the same names as what is in the namespace std. That is a great deal granted, but not so much compared to the number of possible coherent words and pseudonyms that can be come up with strictly speaking.

I mean really... saying "don't rely on this being present" is just setting you up to rely on it NOT being present. You are constantly going to have issues borrowing code snippets and constantly repairing them. Just keep your user-defined and borrowed stuff in limited scope as they should be and be VERY sparing with globals (honestly globals should almost always be a last resort for purposes of "compile now, sanity later"). Truly I think it is bad advice from your teacher because using std will work for both "cout" and "std::cout" but NOT using std will only work for "std::cout". You will not always be fortunate enough to write all your own code.

NOTE: Don't focus too much on efficiency issues until you actually learn a little about how compilers work. With a little experience coding you don't have to learn that much about them before you realize how much they are able to generalize good code into something something simple. Every bit as simple as if you wrote the whole thing in C. Good code is only as complex as it needs to be.

answered Jun 27 '13 at 20:33

Noneyo Getit
47 1

I agree with others - it is asking for name clashes, ambiguities and then the fact is it is less explicit. While I can see the use of using ... my personal preference is to limit it. I would also strongly consider what some others pointed out:

If you want to find a function name that might be a fairly common name, but you only want to find it in std namespace (or the reverse: you want to change all calls that are NOT in namespace std, namespace X, ...), then how do you propose to do this? You could write a program to do it but wouldn't it be better to spend time working on your project itself rather than writing a program to maintain your project?

Personally I actually don't mind the std:: prefix. I like the look more than not. I don't know if that is because it is explicit and says to me "this isn't my code... I am using the standard library" or if it is something else, but I think it looks nicer. This might be odd given that I only recently got in to C++ (used and still do C and other languages for much longer and C is my favourite language of all time, right above assembly).

There is one other thing although it is somewhat related to the above and what others point out. While this might be bad practise, I sometimes reserve std::name for standard library version and name for program-specific implementation. Yes indeed this could bite you and bite you hard but it all comes down to that I started this project from scratch and I'm the only programmer for it. Example: I overload std::string and call it string. I have helpful additions. I did it in part because of my C and Unix (+ Linux) tendency towards lower-case names.

Besides that, you can have namespace aliases. Here is an example of where it is useful that might not have been referred to (I didn't read all the responses and I'm having to rush off for a while in a moment). I use the C++11 standard and specifically with libstdc++. Well check this. It doesn't have complete std::regex support. Sure it compiles but it throws an exception along the lines of it being an error on the programmer's end. But it is lack of implementation. So here's how I solved it. Install boost's regex, link in boost's regex. Then, I do the following so that when libstdc++ has it implemented entirely, I need only remove this block and the code remains the same:

```
namespace std
{
    using boost::regex;
    using boost::regex_error;
    using boost::regex_replace;
    using boost::regex_search;
    using boost::regex_match;
    using boost::smatch;
    namespace regex_constants = boost::regex_constants;
}
```

I won't argue on whether that is a bad idea or not. I will however argue that it keeps it clean for MY project and at the same time makes it specific: True I have to use boost BUT I'm using it like the libstdc++ will eventually have it. Yes, starting your own project and starting with a standard (...) at the very beginning goes a very long way with helping maintenance, development and everything involved with the project!

Edit: Now that I have time, just to clarify something. I don't actually think it is a good idea to use a name of a class/whatever in the STL deliberately and more specifically in place of. string is the exception (ignore the first, above, or second here, pun if you must) for me as I didn't like the idea of 'String'. As it is, I am still very biased towards C and biased against C++. Sparing details, much of what I work on fits C more (but it was a good exercise and a good way to make myself a. learn another language and b. try not be less biased against object/classes/etc which is maybe better stated as: less closed-minded, less arrogant, more accepting.). But what IS useful is what some already suggested: I do indeed use list (it is fairly generic, is it not ?), sort (same thing) to name two that would cause a name clash if I were to do "using namespace std;" and so to that end I prefer being specific, in control and knowing that if I intend it to be the standard use then I will have to specify it. Put simply: no assuming allowed.

And as for making boost's regex part of std. I do that for future integration and - again, I admit fully this is bias - I don't think it is as ugly as boost::regex:: ... Indeed that is another thing for me. There's many things in C++ that I still have yet to come to fully accept in looks and methods (another example: variadic templates versus var args [though I admit variadic templates are very very useful!]). Even those that I do accept it was difficult AND I still have issues with them.

edited Oct 13 '14 at 21:11

answered Oct 13 '14 at 17:30

user4138451

An example where using namespace std throws complilation error because of the ambiguity of count, which is also a function in algorithm library.

```
#include <iostream>

using namespace std;

int count = 1;
int main() {
    cout<<count<<endl;
}
```

answered Dec 31 '14 at 8:00

Nithin

70 6

---

1 `::count` --problem solved. Usually you'll have more stuff from the std namespaced than from elsewhere, ergo keeping the using namespace directive might save you typing. – PSkocik Jul 11 '15 at 19:12

It depends on where it is located. If it is a common header, then you are diminishing the value of the namespace by merging it into the global namespace. Keep in mind, this could be a neat way of making module globals.

edited Oct 11 '15 at 10:18

Peter Mortensen
10.5k 13 72 108

answered Sep 21 '09 at 3:15

MathGladiator
709 1 5 21

A concrete example to clarify the concern. Imagine you have 2 libraries, foo and bar, each with their own namespace:

```
namespace foo {
    void a(float) { /* does something */ }
}

namespace bar {
    ...
}
```

Now let's say you use foo and bar together in your own program as follows:

```
using namespace foo;
using namespace bar;

void main() {
    a(42);
}
```

At this point everything is fine. When you run your program it 'does something'. But later you update bar and let's say it has changed to be like:

```
namespace bar {
    void a(float) { /* does something completely different */ }
}
```

At this point you'll get a compiler error:

```
using namespace foo;
using namespace bar;

void main() {
    a(42);  // error: call to 'a' is ambiguous, should be foo::a(42)
}
```

So you'll need to do some maintenance to clarify which 'a' you meant (i.e. `foo::a`). That's probably undesirable, but fortunately it is pretty easy (just add `foo::` in front of all calls to `a` that the compiler marks as ambiguous).

But imagine an alternative scenario where bar changed instead to look like this instead:

```
namespace bar {
    void a(int) { /* does something completely different */ }
}
```

At this point your call to `a(42)` suddenly binds to `bar::a` instead of `foo::a` and instead of doing 'something' it does 'something completely different'. No compiler warning or anything. Your program just silently starts doing something complete different than before.

When you use a namespace you're risking a scenario like this, which is why people are uncomfortable using namespaces. The more things in a namespace the greater the risk of conflict, so people might be even more uncomfortable using namespace std (due to the number of things in that namespace) than other namespaces.

Ultimately this is a trade-off between writability vs reliability/maintainability. Readability may factor in also, but I could see arguments for that going either way. Normally I would say reliability and maintainability are more important, but in this case you'll constantly pay the writability cost for an fairly rare reliability/maintainability impact. The 'best' trade-off will determine on your project and your priorities.

edited Nov 2 at 14:35

answered Sep 2 at 20:06

Kevin
2,711 3 18 25

I think using **locally** or **globally** should depend on the application.

Because, when we use the library locally, sometimes code going to be a real mess. Readability is going to low.

so, we should use libraries locally when only there is a possibility for conflicts.

I am not more experiences person. So, let me know if I am wrong.

answered Sep 17 '13 at 6:55

meupul
42 5

Yes, the namespace is important. Once in my project, I needed to import one var declaration into my source code, but when compiling it, it conflicted with another third-party library.

At the end, I had to work around around it by some other means and make the code less clear.

edited Oct 11 '15 at 10:42

Peter Mortensen
10.5k 13 72 108

answered Apr 9 '13 at 3:44

harris
549 6 20

Here is an example showing how `using namespace std;` can lead to name clash problems:

Unable to define a global variable in c++

In the example a very generic algorithm name (`std::count`) name clashes with a very reasonable variable name (`count`).

answered Aug 15 at 8:00

Martin
5,114 5 27 50

---

1 2 next

protected by Ben Voigt May 9 '15 at 23:03

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?