2.9 — Naming collisions and an introduction to namespaces

ALEX[1] SEPTEMBER 13, 2022

Let's say you are driving to a friend's house for the first time, and the address given to you is 245 Front Street in Mill City. Upon reaching Mill City, you take out your map, only to discover that Mill City actually has two different Front Streets across town from each other! Which one would you go to? Unless there were some additional clue to help you decide (e.g. you remember his house is near the river) you'd have to call your friend and ask for more information. Because this would be confusing and inefficient (particularly for your mailman), in most countries, all street names and house addresses within a city are required to be unique.

Similarly, C++ requires that all identifiers be non-ambiguous. If two identical identifiers are introduced into the same program in a way that the compiler or linker can't tell them apart, the compiler or linker will produce an error. This error is generally referred to as a **naming collision** (or **naming conflict**).

If the colliding identifiers are introduced into the same file, the result will be a compiler error. If the colliding identifiers are introduced into separate files belonging to the same program, the result will be a linker error.

An example of a naming collision

a.cpp:

```
#include <iostream>

void myFcn(int x)
{
    std::cout << x;
}
```

main.cpp:

```cpp
#include <iostream>

void myFcn(int x)
{
    std::cout << 2 * x;
}

int main()
{
    return 0;
}
```

When the compiler compiles this program, it will compile *a.cpp* and *main.cpp* independently, and each file will compile with no problems.

However, when the linker executes, it will link all the definitions in *a.cpp* and *main.cpp* together, and discover conflicting definitions for function *myFcn*. The linker will then abort with an error. Note that this error occurs even though *myFcn* is never called!

Most naming collisions occur in two cases:

1. Two (or more) identically named functions (or global variables) are introduced into separate files belonging to the same program. This will result in a linker error, as shown above.
2. Two (or more) identically named functions (or global variables) are introduced into the same file. This will result in a compiler error.

As programs get larger and use more identifiers, the odds of a naming collision being introduced increases significantly. The good news is that C++ provides plenty of mechanisms for avoiding naming collisions. Local scope, which keeps local variables defined inside functions from conflicting with each other, is one such mechanism. But local scope doesn't work for function names. So how do we keep function names from conflicting with each other?

What is a namespace?

Back to our address analogy for a moment, having two Front Streets was only problematic because those streets existed within the same city. On the other hand, if you had to deliver mail to two addresses, one at 209 Front Street in Mill City, and another address at 417 Front Street in Jonesville, there would be no confusion about where to go. Put another way, cities provide groupings that allow us to disambiguate addresses that might otherwise conflict with each other. Namespaces act like the cities do in this analogy.

A **namespace** is a region that allows you to declare names inside of it for the purpose of disambiguation. The namespace provides a scope region (called **namespace scope**) to the names declared inside of it -- which simply means that any name declared inside the namespace won't be mistaken for identical names in other scopes.

Key insight

A name declared in a namespace won't be mistaken for an identical name declared in another scope.

Within a namespace, all names must be unique, otherwise a naming collision will result.

Namespaces are often used to group related identifiers in a large project to help ensure they don't inadvertently collide with other identifiers. For example, if you put all your math functions in a namespace called *math*, then your math functions won't collide with identically named functions outside the *math* namespace.

We'll talk about how to create your own namespaces in a future lesson.

The global namespace

In C++, any name that is not defined inside a class, function, or a namespace is considered to be part of an implicitly defined namespace called the **global namespace** (sometimes also called **the global scope**).

In the example at the top of the lesson, functions main() and both versions of myFcn() are defined inside the global namespace. The naming collision encountered in the example happens because both versions of myFcn() end up inside the global namespace, which violates the rule that all names in the namespace must be unique.

Only declarations and definition statements can appear in the global namespace. This means we can define variables in the global namespace, though this should generally be avoided (we cover global variables in lesson 6.4 -- Introduction to global variables (https://www.learncpp.com/cpp-tutorial/introduction-to-global-variables/)[2]). This also means that other types of statements (such as expression statements) cannot be placed in the global namespace (initializers for global variables being an exception):

```
#include <iostream> // handled by preprocessor

// All of the following statements are part of the global namespace
void foo();    // okay: function forward declaration in the global namespace
int x;         // compiles but strongly discouraged: uninitialized variable definition in the global namespace
int y { 5 };   // compiles but discouraged: variable definition with initializer in the global namespace
x = 5;         // compile error: executable statements are not allowed in the global namespace

int main()     // okay: function definition in the global namespace
{
    return 0;
}

void goo();    // okay: another function forward declaration in the global namespace
```

The std namespace

When C++ was originally designed, all of the identifiers in the C++ standard library (including std::cin and std::cout) were available to be used without the *std::* prefix (they were part of the global namespace). However, this meant that any identifier in the standard library could potentially conflict with any name you picked for your own identifiers (also defined in the global namespace). Code that was working might suddenly have a naming conflict when you #included a new file from the standard library. Or worse, programs that would compile under one version of C++ might not compile under a future version of C++, as new identifiers introduced into the standard library could have a naming conflict with already written code. So C++ moved all of the functionality in the standard library into a namespace named "std" (short for standard).

It turns out that *std::cout*'s name isn't really *std::cout*. It's actually just *cout*, and *std* is the name of the namespace that identifier *cout* is part of. Because *cout* is defined in the *std* namespace, the name *cout* won't conflict with any objects or functions named *cout* that we create in the global namespace.

Similarly, when accessing an identifier that is defined in a namespace (e.g. *std::cout*) , you need to tell the compiler that we're looking for an identifier defined inside the namespace (*std*).

Key insight

When you use an identifier that is defined inside a namespace (such as the *std* namespace), you have to tell the compiler that the identifier lives inside the namespace.

There are a few different ways to do this.

Explicit namespace qualifier std::

The most straightforward way to tell the compiler that we want to use *cout* from the *std* namespace is by explicitly using the *std::* prefix. For example:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello world!"; // when we say cout, we mean the cout defined in
6 the std namespace
7     return 0;
  }
```

The :: symbol is an operator called the **scope resolution operator**. The identifier to the left of the :: symbol identifies the namespace that the name to the right of the :: symbol is contained within. If no identifier to the left of the :: symbol is provided, the global namespace is assumed.

So when we say *std::cout*, we're saying "the *cout* that lives in namespace *std*".

This is the safest way to use *cout*, because there's no ambiguity about which *cout* we're referencing (the one in the *std* namespace).

> **Best practice**
>
> Use explicit namespace prefixes to access identifiers defined in a namespace.

When an identifier includes a namespace prefix, the identifier is called a **qualified name**.

Using namespace std (and why to avoid it)

Another way to access identifiers inside a namespace is to use a *using directive* statement. Here's our original "Hello world" program with a *using directive*:

```
1  #include <iostream>
2
3  using namespace std; // this is a using directive that allows us to access names
   in the std namespace with no namespace prefix
4
5  int main()
6  {
7      cout << "Hello world!";
8      return 0;
9  }
```

A **using directive** allows us to access the names in a namespace without using a namespace prefix. So in the above example, when the compiler goes to determine what identifier *cout* is, it will match with *std::cout*, which, because of the using directive, is accessible as just *cout*.

Many texts, tutorials, and even some IDEs recommend or use a using-directive at the top of the program. However, used in this way, this is a bad practice, and highly discouraged.

Consider the following program:

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

using namespace std; // makes std::cout accessible as "cout"

int cout() // defines our own "cout" function in the global namespace
{
    return 5;
}

int main()
{
    cout << "Hello, world!"; // Compile error!  Which cout do we want here?  The one in the std namespace or the one we defined above?

    return 0;
}
```

The above program doesn't compile, because the compiler now can't tell whether we want the *cout* function that we defined, or the *cout* that is defined inside the *std* namespace.

When using a using-directive in this manner, *any* identifier we define may conflict with *any* identically named identifier in the *std* namespace. Even worse, while an identifier name may not conflict today, it may conflict with new identifiers added to the std namespace in future language revisions. This was the whole point of moving all of the identifiers in the standard library into the *std* namespace in the first place!

Warning

Avoid using-directives (such as *using namespace std;*) at the top of your program or in header files. They violate the reason why namespaces were added in the first place.

Related content

We talk more about using-declarations and using-directives (and how to use them responsibly) in lesson 6.12 -- Using declarations and using directives (https://www.learncpp.com/cpp-tutorial/using-declarations-and-using-directives/)[3].

Next lesson
2.10 Introduction to the preprocessor

4

Back to table of contents

5

Previous lesson
2.8 Programs with multiple code files

6

7

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Find a mistake? Leave a comment!

Avatars from https://gravatar.com/[9] are connected to your provided email address.

Notify me about replies:

POST COMMENT

193 COMMENTS Newest

naming collision
September 5, 2022 7:45 pm

>>2. Two (or more) definitions for a function (or global variable) are introduced into the same file (often via an #include). This will result in a compiler error.

I was wondering why on the previous lesson when we included "#include "iostream"" twice in main.cpp and input.cpp it didn't throw any naming collision?

1 Reply

#include hater

Reply to naming collision [10] October 21, 2022 4:57 am

I'd recommend not using #include at all and write your own code that does the function for you. This could take a while but would save you from having to use those nasty include statements.

0 Reply

Tom Duhamel

Reply to naming collision [10] September 15, 2022 9:53 pm

<iostream> does not contain any definition at all, only declarations. You will find that decorations can be duplicated as often as necessary. There will not be any conflict, as long as all of the different declarations are of the same signature.

0 Reply

Alex Author

Reply to naming collision [10] September 13, 2022 5:34 pm

Header guards (which every header file should have) will prevent the content of a header file from being #included more than once into the same file.

2 Reply

naming collision

Reply to Alex [11] September 13, 2022 5:50 pm

It means both #include "iostream" has a header guard! That makes sense. Thanks!

Last edited 1 month ago by naming collision

0 Reply

Stan

Reply to naming collision [10] September 6, 2022 8:02 am

The answer is in your comments I guess?
if #include is in the same file.. compile error. But we used it in two different files (main.cpp and input.cpp)
Without #include <iostream> you can not access the input/output functions we are using :)

0 Reply

bpw

Reply to Stan [12] September 7, 2022 2:24 am

Because iostream is an include that contains the function declarations only. Each .cpp file needs those declarations in order to compile. Remember that each .cpp file is compiled as a standalone unit.
The linker finally links everything against the C++ standard library which contains the definitions of the functions declared in the iostream header. Thus, there are no multiple function definitions.

0 Reply

naming collision

Reply to Stan [12] September 6, 2022 10:15 am

I am afraid you're wrong. If we #include the same file twice in a file, no naming collision will occur and that makes me confused why?!

0 Reply

Stan

Reply to naming collision [13] September 6, 2022 10:21 am

I guess I am way too 'begginer'to comprehend your question properly. I just started this course, but still feel like CPP being 'trust the programmer'language is what makes this possible. Since it reads each file independently it just trusts you that you know what you are doing. Sorry if I still can not properly comprehend your question but I hope I will get something out of it too if that is the case :)

Last edited 1 month ago by Stan

0 Reply

Austin

August 14, 2022 6:11 pm

This is entirely in my opinion, I think adding alternate definitions next to keywords would be very beneficial for someone like me, however, if you don't feel like doing this, anyone can just write the definitions down in their notes and look at it when they need to.

0 Reply

Alex Author

Reply to Austin [14] August 15, 2022 11:45 am

What do you mean by "alternate definitions"?

1 Reply

Dave

Reply to Alex [15] August 16, 2022 6:43 pm

I'm just extrapolating, but OP might mean something like a paragraph of an explicit definition next to (maybe a hover-over) those bolded key terms such as **naming collision** or **the global scope**. I could be wrong though. Awesome guide nonetheless!

0 Reply

Krishnakumar

July 20, 2022 12:25 pm

>We talk more about using statements (and how to use them responsibly)

Potentially confusing tautology.

Perhaps reword the parenthesised text as "and how to employ them responsibly"

0 Reply

Cris

Reply to Krishnakumar [16] October 25, 2022 7:06 am

No, this really isn't confusing at all.

0 Reply

T.V.

July 3, 2022 10:56 pm

I've tried many times to get into programming. Started with Python, it was okay but I wasn't motivated to continue, however after using this site to learn C++ I have actually been enjoying the learning curve. I like how in-depth this site is, and how I am learning many things to avoid. The Author of this is an absolute legend.

21 Reply

Krishnakumar

June 18, 2022 1:59 pm

```
1 int cout() // declares our own "cout" function in the global namespace
2 {
3     return 5;
4 }
```

is a definition, not a pure declaration, right? The comment is a bit misleading.

1 Reply

> **Alex** Author
>
> Reply to Krishnakumar [17] June 19, 2022 11:20 am
>
> A definition serves as a declaration, so it's both. But yes, it's more accurate to call it a definition. Updated. Thanks!
>
> 1 Reply

Krishnakumar

June 18, 2022 1:57 pm

"Many texts, tutorials, and even **some compilers** recommend or use a using-directive at the top of the program."

Just curious to know which compilers recommend this?

0 Reply

> **Alex** Author
>
> Reply to Krishnakumar [18] June 19, 2022 11:19 am
>
> Changed compilers to IDEs since it's the IDEs that sometimes do this when they provide a starter "Hello, world!" program when you create a new proejct.

2 Reply

thenotoriouscoder
June 16, 2022 12:39 am

I have started learning C++ and I asked one of my friends to recommend some material for me and he recommended your website. Now, it is easy to learn c++.

7 Reply

Pri
June 6, 2022 3:10 pm

Thank you so much for putting this together! This is incredibly helpful and easy to follow!

1 Reply

Ajit
May 30, 2022 8:40 am

This is interesting

```
#include <iostream>

using std::cout;

int foo{1};

int main()
{
    int foo{2};
    cout<<"Value of foo in Main : "<<foo<<'\n';
    cout<<"Value of foo in Global : "<<::foo<<'\n';
    return 0;
}
```

I like the use of `::` !

```
Value of foo in Main : 2
Value of foo in Global : 1

[Program finished]
```

6 Reply

Tabitha
May 19, 2022 9:00 am

Just started reading your tutorial on c++. This is the most comprehensive site for learning for real. I have used others, but, this stands out. Thanks a lot Alex for sharing the things you know.

7 Reply

> **Alex** Author
>
> Reply to Tabitha [19] May 19, 2022 12:53 pm
>
> You're welcome. Thanks for visiting!
>
> 4 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-global-variables/
3. https://www.learncpp.com/cpp-tutorial/using-declarations-and-using-directives/
4. https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/programs-with-multiple-code-files/
7. https://www.learncpp.com/naming-collisions-and-an-introduction-to-namespaces/
8. https://www.learncpp.com/cpp-tutorial/the-override-and-final-specifiers-and-covariant-return-types/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-572748
11. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-573085
12. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-572758
13. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-572762

14. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-572058
15. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-572114
16. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-570858
17. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-569883
18. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-569882
19. https://www.learncpp.com/cpp-tutorial/naming-collisions-and-an-introduction-to-namespaces/#comment-568787
20. https://www.ezoic.com/what-is-ezoic/