# 2.9 — Naming collisions and an introduction to namespaces

BY ALEX ON NOVEMBER 6TH, 2016 | LAST MODIFIED BY NASCARDRIVER ON DECEMBER 28TH, 2019

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 pull up 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**).

## An example of a naming collision

a.cpp:

```
#include <iostream>

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

main.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) definitions for a function (or global variable) are introduced into separate files that are compiled into the same program. This will result in a linker error, as shown above.
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.

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 functions 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 and 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 (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.

## 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:

```
#include <iostream>

int main()
{
    std::cout << "Hello world!"; // when we say cout, we mean the cout defined in the std names
    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.

## 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*:

```
#include <iostream>

using namespace std; // this is a using directive telling the compiler to check the std namespa

int main()
{
    cout << "Hello world!"; // cout has no prefix, so the compiler will check to see if cout is
    return 0;
}
```

A **using directive** tells the compiler to check a specified namespace when trying to resolve an identifier that has no namespace prefix. So in the above example, when the compiler goes to determine what identifier *cout* is, it will check both locally (where it is undefined) and in the *std* namespace (where it will match to *std::cout*).

Many texts, tutorials, and even some compilers 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:

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

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

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

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

    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. They violate the reason why namespaces were added in the first place.

We'll talk more about using statements (and how to use them responsibly) in lesson **6.12 -- Using statements**.

**2.10 -- Introduction to the preprocessor**

**Index**

**2.8 -- Programs with multiple code files**

C++ TUTORIAL | PRINT THIS POST

## 87 comments to 2.9 — Naming collisions and an introduction to namespaces

« Older Comments 1 2

salah
February 6, 2020 at 3:12 pm · Reply

Amazing tutorial do you have another tutorial like : data structure?

slate
January 30, 2020 at 5:36 am · Reply

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!

but in other lessons you say that compiler dont look what is int the other files so why its make error

nascardriver
January 30, 2020 at 7:44 am · Reply

The compiler doesn't care about other source files, but the linker does. The compiler compiles each source file individually, the linker combines the compiled files into one binary.

**slate**
January 30, 2020 at 8:48 am · Reply

thank you, but for example as defined by a variable with the same name in different files, there is no error why

nascardriver
January 30, 2020 at 8:50 am · Reply

If you define two variables in different files with the same name you get an error. We walk about this in the lesson on global variables.

cas
January 27, 2020 at 8:58 am · Reply

This tutorial is amazing

Omran
January 26, 2020 at 8:29 am · Reply

bro this is an amazing tutorial , i'm actually enjoying it and also clicking on every ad to support it thanks a lot guys :)

Sam
December 25, 2019 at 2:22 pm · Reply

"A namespace is a region that allows you to declared names inside of it for the purpose of disambiguation"

Grammatical error. Change "declared" to "declare"!

Merry Christmas!

nascardriver
December 28, 2019 at 4:18 am · Reply

Fixed, thanks!

koe
December 9, 2019 at 1:41 pm · Reply

"There are a few different ways to do this."

Only two are mentioned here, better to say:
"There are a couple ways to do this."
or
"There are a few different ways to do this. We will highlight two here, the other methods (?) are bad (?)."

Nirbhay
November 14, 2019 at 4:42 am · Reply

Hello Alex!

I would be thankful if you can find a way to show the readers which part of the tutorial is updated or which new content is added. I went through this tutorial quite some time ago and now I do not know what to look for in this update. :)

Thanks

Alex
November 15, 2019 at 2:56 pm · Reply

Unfortunately I'm not aware of any easy way to do this. :(

**sharfan**
November 14, 2019 at 2:04 am · Reply

very good notes

Vai
October 16, 2019 at 4:04 am · Reply

"Avoid using directives (such as using namespace std;) at the top of your program. They violate the reason why namespaces were added in the first place." then why "using namespace" was introduced? When will it be useful?

**nascardriver**
October 17, 2019 at 12:51 am · Reply

The rule isn't "never use", it's "avoid". There are cases in which `using namespace` is useful. If you know that your libraries are compatible with each other (`std` isn't even compatible with other parts of C++, so don't use `using namespace std;` unless it's in a small scope), you can use `using namespace`.

Alex
October 18, 2019 at 2:46 pm · Reply

It can still be useful when used inside a smaller scope, such as a function.

Nirbhay
November 15, 2019 at 12:54 am · Reply

@Alex please explain it with an example..why can it be used inside a smaller scope, such as a function?

nascardriver
November 15, 2019 at 3:30 am · Reply

Using `using namespace` in a function limits the scope that's affected by the `using` directive. Anyone reading the function should be able to see the directive and write code accordingly. If the directive was at file-scope, it's easy to miss.

I'll use `chrono` for an example of a `using` directive. It doesn't make sense for `std`.

```
1  #include <iostream>
2  #include <chrono>
3
4  void printTime()
5  {
6    using namespace std::literals::chrono_literals;
7
8    std::cout << "one minute and ten seconds is " << (1min + 10s).count() << "
```

```
}

void printTimeLong()
{
  std::cout << "one minute and ten seconds is " << (std::chrono::minutes{ 1 }
}

int main()
{
  printTime();
  printTimeLong();

  return 0;
}
```

As you can see, it makes sense `using namespace` in this case, as the code without it is a lot longer and not as easy to read.
If we used the namespace at file-scope, we wouldn't be able to user other user-defined literals anymore.

« Older Comments 1 2