

## 2.9 — Naming collisions and an introduction to namespaces

 **ALEX**  **NOVEMBER 17, 2023**

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 your friend's 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 mail carrier), 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:

```
#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 245 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 definitions can appear in a namespace. This means we can define variables in the global namespace, though this should generally be avoided (we cover global variables in lesson [7.4 -- Introduction to global variables](https://www.learncpp.com/cpp-tutorial/introduction-to-global-variables/) (<https://www.learncpp.com/cpp-tutorial/introduction-to-global-variables/>)). This also means that other types of statements (such as expression statements) cannot be placed in the global namespace.

```
#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
```

### Key insight

Only declarations and definitions can appear in a 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:

```
#include <iostream>

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

```
#include <iostream>

using namespace std; // this is a using directive that allows us to access names in the std namespace with no namespace
prefix

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

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:

```
#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 [7.12 -- Using declarations and using directives](#) (<https://www.learncpp.com/cpp-tutorial/using-declarations-and-using-directives/>).



#### Next lesson

**2.10** [Introduction to the preprocessor](#)



#### [Back to table of contents](#)



#### Previous lesson

**2.8** [Programs with multiple code files](#)

Leave a comment...

 Name\*


 Email\* 

Notify me about replies:



POST COMMENT

 Find a mistake? Leave a comment above!

 Avatars from <https://gravatar.com/> are connected to your provided email address.

265 COMMENTS

Newest ▾

We and our partners share information on your use of this website to help improve your experience.

Do not sell my info: ☐

OKAY

