1.8a — Naming conflicts and the std namespace

BY ALEX ON NOVEMBER 6TH, 2016 | LAST MODIFIED BY ALEX ON MARCH 20TH, 2017

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 a particular donut shop) 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 (variable and/or function names) be non-ambiguous. If two identifiers are introduced into the same program in a way that the compiler 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 doSomething(int x)
{
    std::cout << x;
}
```

b.cpp:

```
#include <iostream>

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

main.cpp:

```
void doSomething(int x); // forward declaration for doSomething

int main()
{
    doSomething(5);

    return 0;
}
```

Files a.cpp, b.cpp, and main.cpp will all compile just fine, since individually there's no problem. However, when a.cpp and b.cpp are put in the same project together, a naming conflict will occur, since the function doSomething() is defined in both. This will cause a linker error.

Most naming collisions occur in two cases:
1) Two files are added into the same project that have a function (or global variable) with the same name (linker error).
2) A code file includes a header file that contains an identifier that conflicts with something else (compile error). We'll discuss header files in the next lesson.

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 (such as local scope, which keeps variables inside functions from conflicting with each other, and namespaces, which we'll introduce shortly), so most of the time you won't need to worry about this.

The std namespace

When C++ was originally designed, all of the identifiers in the C++ standard library (such as cin and cout) were available to be used directly. However, this meant that any identifier in the standard library could potentially conflict with a name you picked for your own

identifiers. 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 functionality introduced into the standard library could conflict. So C++ moved all of the functionality in the standard library into a special area called a namespace.

Much like a city guarantees that all roads within the city have unique names, a namespace guarantees that identifiers within the namespace are unique. This prevents the identifiers in a namespace from conflicting with other identifiers.

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 it lives inside. All of the functionality in the C++ standard library is defined inside a namespace named *std* (short for standard). In this way, we don't have to worry about the functionality of the standard library having a naming conflict with our own identifiers.

We'll talk more about namespaces in a future lesson and also teach you how to create your own. For now, the only thing you really need to know about namespaces is that whenever we use an identifier (like std::cout) that is part of the standard library, we need to tell the compiler that that identifier lives inside the std namespace.

Rule: When you use an identifier in a namespace, you always have to identify the namespace along with the identifier

Explicit namespace qualifier std::

The most straightforward way to tell the compiler that cout lives in the std namespace is by using the "std::" prefix. For example:

```
1    std::cout << "Hello world!";
```

This is the safest way to use cout, because there's no ambiguity about where cout lives (it's clearly in the std namespace).

C++ provides other shortcuts for indicating what namespace an identifier is part of (via using statements). We cover those in lesson **4.3c -- Using statements**.


1.9 -- Header files


Index


1.8 -- Programs with multiple files

Share this:

C++ TUTORIAL | PRINT THIS POST

55 comments to 1.8a — Naming conflicts and the std namespace

sri
July 1, 2018 at 8:31 pm · Reply

Does overriding a function contradict the ODR because you are redefining a unction with the same name and number of parameters?

Alex
July 9, 2018 at 9:38 am · Reply