C# and C++ type aliases and their consequences

devblogs.microsoft.com/oldnewthing/20220117-00

January 17, 2022

Raymond Chen

The C# and C++ language provide ways to introduce shorter names for things. These shortcuts do not have their own identities; they merely let one name be used as a shorthand for the other thing.

```
// C#
using Console = System.Console;

// C++
using Project = Contoso::Project;
```

The C# and C++ programming languages call these *aliases*. You are allowing an existing type to go by a different name. It does not create a new type, and the new name is interchangeable with the old one.

```
// C++
extern void UpdateProject(Contoso::Project& project);

void example()
{
    Project project;
    UpdateProject(project); // this works
}
```

Similarly, when you import a namespace with a `using` directive, the names from the other namespace are visible in your namespace, but they still belong to that other namespace.[1]

```
// C++
namespace Other
{
    struct OtherStruct;
}

namespace Mine
{
    using namespace Other;
}

void Welcome(Mine::OtherStruct s);
```

The signature of the `Welcome` function is `void Welcome(Other::OtherStruct)`, not `void Welcome(Mine::OtherStruct)`.

This trick also gives you a way to switch easily between two options:

```
#ifdef USE_CONTOSO_WIDGET
using Widget = Contoso::Widget;
#else
using Widget = LitWare::Widget;
#endif

// code that uses Widget without caring whose widget it is
```

The fact that these aliases do not introduce new types means that when you go looking in the debugger, you will see the symbols decorated with their *original* names. Which can be both a good thing and a bad thing.

It's a good thing if you want the original name to be the one seen by the outside world. For example, you might create aliases for commonly-used types in your component, but you want people outside your component to use the original names.

```
// component.h

namespace Component
{
    struct ReversibleWidget;

    void CheckPolarity(ReversibleWidget const&);
}

// component.cpp (implementation)
#include<component.h>

using FlipWidget = Component::ReversibleWidget;

void Component::CheckPolarity(FlipWidget const& widget)
{
    ... do stuff ...
}
```

Inside your component, you'd rather just call it a `FlipWidget`, because that was the internal code name when the product was being developed, and then later, management decided that its public name should be `ReversibleWidget`. You can create an alias that lets you continue using your internal code name, so you don't have to perform a massive search-and-replace across the entire code base (and deal with all the merge conflicts that will inevitably arise).

That the symbols are decorated with the original names can be a bad thing if the original name is an unwieldy mess, which is unfortunately the case with many classes in the C++ standard library.

In the C++ standard library, `string` is an alias for `basic_string<char, std::char_traits<char>, std::allocator<char> >`,[2] so a function like

```
void FillLookupTable(std::map<std::string, std::string>& table);
```

formally has the signature (deep breath)

```
FillLookupTable(std::map<std::basic_string<char, std::char_traits<char>,
std::allocator<char> >, std::basic_string<char, std::char_traits<char>,
std::allocator<char> >, std::less<std::basic_string<char, std::char_traits<char>,
std::allocator<char> > >, std::allocator<std::pair<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > const, std::basic_string<char,
std::char_traits<char>, std::allocator<char> > > > >&):
```

Good luck typing that into a debugger.

[1] The fact that they remain in the original namespace has consequences for argument-dependent lookup:

```
namespace X
{
    struct S {};
    void fiddle(S const&);
}

namespace Y
{
    using namespace X;
    void fiddle(S const&);
}

void test()
{
    Y::S s;
    fiddle(s); // X::fiddle, not Y::fiddle
}
```

[2] What you're seeing is a combination of the type alias *and* the template default parameters.

Raymond Chen

Follow