The case of creating new instances when you wanted to use the same one

devblogs.microsoft.com/oldnewthing/20250529-00/?p=111228

May 29, 2025

A colleague of mine was trying to debug some code that they wrote. They wanted to create an object on demand, but their function to return the on-demand object kept creating new instances rather than reusing the previously-created one.

Here's a simplified version.

```
// For expository simplicity, assume single-threaded objects

struct Widget
{
    Widget(bool debugMode = false);

    int m_count = 0;
    void Increment() { ++m_count; }
};

struct Doodad
{
    std::unique_ptr<Widget> m_widget; // created on demand

    Widget GetWidget()
    {
        if (!m_widget) {
            m_widget = std::make_unique<Widget>();
        }
        return m_widget.get();
    }
};

void Sample()
{
    Doodad doodad;

    // Demand-create the widget and increment its counter.
    doodad.GetWidget().Increment();

    // This assertion fires?
    ASSERT(doodad.GetWidget().m_count > 0);
}
```

It's as if the `GetWidget()` function is creating brand new empty widgets instead of creating one on its first call and reusing it for subsequent calls.

Maybe you can spot the problem.

What struck me was that the return type of `GetWidget()` was wrong. The `m_widget.get()` returns a `Widget*`, but the function returns a `Widget` object. But how did this code even compile? The return type is wrong!

The answer is a backward compatibility feature of C++ combined with a poor choice of default in the language design.

The backward compatibility feature is that pointers can implicitly convert to `bool`: The built-in boolean conversion is a comparison against `nullptr`. This is a backward compatibility feature carried over from C.

The poor choice of default in the language is that any constructor that can be called with a single parameter (though possibly with the help of some defaulted parameters) is by default usable as an implicit conversion. To opt out, you must use the `explicit` keyword.

```
struct Widget
{
    // ↓ usable as implicit conversion from bool
    Widget(bool debugMode = false);

    int m_count = 0;
    void Increment() { ++m_count; }
};
```

If we write out the implicit conversion, the `GetWidget` becomes

```
    Widget GetWidget()
    {
        if (!m_widget) {
            m_widget = std::make_unique<Widget>();
        }
        return Widget(m_widget.get() != nullptr);
    }
```

Now we see more clearly what's going on.

The pointer produced by `m_widget.get()` is converted to a `bool` by checking whether it is null. (And since the pointer inside `m_widget` is always non-null by the time we get to the `return` statement, the result is always `true`.) And then we use the `Widget` constructor that takes a single `bool` parameter and use it as a conversion.

The result is that the `GetWidget()` method always returns a freshly-created `Widget` in debug mode.

I was initially baffled as to how the original code compiled, seeing as the return type was wrong. I figured out that it was the implicit conversion by looking at the code generation. The code for the `return` statement looked like this:

```
        cmp     QWORD PTR [rbx], 0    ; Q: m_widget.get() == nullptr?
        setne   dl                    ; dl = 1 if non-null (constructor parameter)
        mov     rcx, rdi              ; return value slot
        call    Widget::Widget(bool)  ; create a widget
        mov     rax, rdi              ; and return it
```

One possible fix is to return a pointer to the Widget:

```
struct Doodad
{
    〖 ... 〗

    Widget* GetWidget()
    {
        if (!m_widget) {
            m_widget = std::make_unique<Widget>();
        }
        return m_widget.get();
    }
};

void Sample()
{
    Doodad doodad;

    // Demand-create the widget and increment its counter.
    doodad.GetWidget()->Increment();

    // This assertion no longer fires
    ASSERT(doodad.GetWidget()->m_count > 0);
}
```

Another option is to return a reference to the Widget.

```
struct Doodad
{
    〖 ... 〗

    Widget& GetWidget()
    {
        if (!m_widget) {
            m_widget = std::make_unique<Widget>();
        }
        return *m_widget;
    }
};

void Sample()
{
    Doodad doodad;

    // Demand-create the widget and increment its counter.
    doodad.GetWidget().Increment(); // no change to callers

    // This assertion no longer fires
    ASSERT(doodad.GetWidget().m_count > 0);
}
```

While we’re at it, let’s make that constructor explicit to remove the implicit conversion.

```
struct Widget
{
    explicit Widget(bool debugMode = false);

    int m_count = 0;
    void Increment() { ++m_count; }
};
```