# Everybody thinks about CLR objects the wrong way (well not everybody)

devblogs.microsoft.com/oldnewthing/20100810-01

August 10, 2010

Raymond Chen

Many people responded to Everybody thinks about garbage collection the wrong way by proposing variations on auto-disposal based on scope:

- "Any local variable that is IDisposable should dispose itself when it goes out of scope."
- "You should be able to attach an attribute to a class that says the destructor should be called immediately after leaving scope."
- "It should have promised to call finalizers on scope exit."

What these people fail to recognize is that they are dealing with object *references*, not objects. (I'm restricting the discussion to reference types, naturally.) In C++, you can put an object in a local variable. In the CLR, you can only put an object *reference* in a local variable.

For those who think in terms of C++, imagine if it were impossible to declare instances of C++ classes as local variables on the stack. Instead, you had to declare a local variable that was a pointer to your C++ class, and put the object in the pointer.

| C# | C++ |
|---|---|
| | `void Function(OtherClass o)`<br>`{`<br>`  // No longer possible to declare objects`<br>`  // with automatic storage duration`<br>`  ~~Color c(0,0,0);~~`<br>`  ~~Brush b(c);~~`<br>`  ~~o.SetBackground(b);~~`<br>`}` |

```
void
Function(OtherClass o)
{
 Color c = new
Color(0,0,0);
 Brush b = new
Brush(c);
 o.SetBackground(b);
}
```

```
void Function(OtherClass* o)
{
 Color* c = new Color(0,0,0);
 Brush* b = new Brush(c);
 o->SetBackground(b);
}
```

This world where you can only use pointers to refer to objects is the world of the CLR.

In the CLR, objects never go out of scope because objects don't have scope.[1] Object *references* have scope. Objects are alive from the point of construction to the point that the last *reference* goes out of scope or is otherwise destroyed.

If objects were auto-disposed when references went out of scope, you'd have all sorts of problems. I will use C++ notation instead of CLR notation to emphasize that we are working with references, not objects. (I can't use actual C++ references since you cannot change the referent of a C++ reference, something that is permitted by the CLR.)

**C#**

```
void
Function(OtherClass o)
{
 Color c = new
Color(0,0,0);
 Brush b = new
Brush(c);
 Brush b2 = b;
 o.SetBackground(b2);
}
```

**C++**

```
void Function(OtherClass* o)
{
 Color* c = new Color(0,0,0);
 Brush* b = new Brush(c);
 Brush* b2 = b;
 o->SetBackground(b2);
 // automatic disposal when variables go out of
scope
 dispose b2;
 dispose b;
 dispose c;
 dispose o;
}
```

Oops, we just double-disposed the `Brush` object and probably prematurely disposed the `OtherClass` object. Fortunately, disposal is idempotent, so the double-disposal is harmless (assuming you actually meant disposal and not destruction). The introduction of `b2` was artificial in this example, but you can imagine `b2` being, say, the leftover value in a variable at the end of a loop, in which case we just accidentally disposed the last object in an array.

Let's say there's some attribute you can put on a local variable or parameter to say that you don't want it auto-disposed on scope exit.

**C#**

```
void Function([NoAutoDispose]
OtherClass o)
{
 Color c = new Color(0,0,0);
 Brush b = new Brush(c);
 [NoAutoDispose] Brush b2 = b;
 o.SetBackground(b2);
}
```

**C++**

```
void Function([NoAutoDispose] OtherClass* o)
{
 Color* c = new Color(0,0,0);
 Brush* b = new Brush(c);
 [NoAutoDispose] Brush* b2 = b;
 o->SetBackground(b2);
 // automatic disposal when variables go out
of scope
 dispose b;
 dispose c;
}
```

Okay, that looks good. We disposed the `Brush` object exactly once and didn't prematurely dispose the `OtherClass` object that we received as a parameter. (Maybe we could make `[NoAutoDispose]` the default for parameters to save people a lot of typing.) We're good, right?

Let's do some trivial code cleanup, like inlining the `Color` parameter.

**C#**

```
void Function([NoAutoDispose]
OtherClass o)
{
 Brush b = new Brush(new
Color(0,0,0));
 [NoAutoDispose] Brush b2 = b;
 o.SetBackground(b2);
}
```

**C++**

```
void Function([NoAutoDispose] OtherClass* o)
{
 Brush* b = new Brush(new Color(0,0,0));
 [NoAutoDispose] Brush* b2 = b;
 o->SetBackground(b2);
 // automatic disposal when variables go out
of scope
 dispose b;
}
```

Whoa, we just introduced a semantic change by what seemed like a harmless transformation: The `Color` object is no longer auto-disposed. This is even more insidious than the scope of a variable affecting its treatment by anonymous closures, for introduction of temporary variables to break up a complex expression (or removal of one-time temporary variables) are common transformations that people expect to be harmless, especially since many language transformations are expressed in terms of temporary variables. Now you have to remember to tag all of your temporary variables with `[NoAutoDospose]`.

Wait, we're not done yet. What does `SetBackground` do?

**C#**

**C++**

```
void
OtherClass.SetBackground([NoAutoDispose
] Brush b)
{
 this.background = b;
}
```

```
void
OtherClass::SetBackground([NoAutoDispose]
Brush* b)
{
 this->background = b;
}
```

Oops, there is still a reference to that `Brush` in the `o.background` member. We disposed an object while there were still outstanding references to it. Now when the `OtherClass` object tries to use the reference, it will find itself operating on a disposed object.

Working backward, this means that we should have put a `[NoAutoDispose]` attribute on the `b` variable. At this point, it's six of one, a half dozen of the other. Either you put `using` around all the things that you want auto-disposed or you put `[NoAutoDispose]` on all the things that you don't.[2]

The C++ solution to this problem is to use something like `shared_ptr` and reference-counted objects, with the assistance of `weak_ptr` to avoid reference cycles, and being very selective about which objects are allocated with automatic storage duration. Sure, you could try to bring this model of programming to the CLR, but now you're just trying to pick all the cheese off your cheeseburger and intentionally going against the automatic memory management design principles of the CLR.

I was sort of assuming that since you're here for CLR Week, you're one of those people who actively chose to use the CLR and want to use it in the manner in which it was intended, rather than somebody who wants it to work like C++. If you want C++, you know where to find it.

**Footnote**

[1] Or at least don't have scope in the sense we're discussing here.

[2] As for an attribute for specific classes to have auto-dispose behavior, that works only if all references to auto-dispose objects are in the context of a create/dispose pattern. References to auto-dispose objects outside of the create/dispose pattern would need to be tagged with the `[NoAutoDispose]` attribute.

```
[AutoDispose] class Stream { ... };
Stream MyClass.GetSaveStream()
{
 [NoAutoDispose] Stream stm;
 if (saveToFile) {
  stm = ...;
 } else {
  stm = ...;
 }
 return stm;
}
void MyClass Save()
{
 // NB! do not combine into one line
 Stream stm = GetSaveStream();
 SaveToStream(stm);
}
```

Raymond Chen

## Follow