When does an object become available for garbage collection?

devblogs.microsoft.com/oldnewthing/20100810-00

August 10, 2010

Raymond Chen

As we saw last time, garbage collection is a method for simulating an infinite amount of memory in a finite amount of memory. This simulation is performed by reclaiming memory once the environment can determine that the program wouldn't notice that the memory was reclaimed. There are a variety of mechanism for determining this. In a basic tracing collector, this determination is made by taking the objects which the program has definite references to, then tracing references from those objects, contining transitively until all accessible objects are found. But what looks like a definite reference in your code may not actually be a definite reference in the virtual machine: Just because a variable is in scope doesn't mean that it is live.

```
class SomeClass {
 ...
 string SomeMethod(string s, bool reformulate)
 {
  OtherClass o = new OtherClass(s);
  string result = Frob(o);
  if (reformulate) Reformulate();
  return result;
 }
}
```

For the purpose of this discussion, assume that the `Frob` method does not retain a reference to the object `o` passed as a parameter. When does the `OtherClass` object `o` become eligible for collection? A naïve answer would be that it becomes eligible for collection at the closing-brace of the `SomeMethod` method, since that's when the last reference (in the variable `o`) goes out of scope.

A less naïve answer would be that it become eligible for collection after the return value from `Frob` is stored to the local variable `result`, because that's the last line of code which uses the variable `o`.

A closer study would show that it becomes eligible for collection even sooner: Once the call to `Frob` returns, the variable `o` is no longer accessed, so the object could be collected even before the result of the call to `Frob` is stored into the local variable `result`. Optimizing compilers have known this for quite some time, and there is a strong likelihood that the variables `o` and `result` will occupy the same memory since their lifetimes do not overlap. Under such conditions, the code generation for the statement could very well be something like this:

```
mov ecx, esi        ; load "this" pointer into ecx register
mov edx, [ebp-8]    ; load parameter ("o") into edx register
call SomeClass.Frob ; call method
mov [ebp-8], eax    ; re-use memory for "o" as "result"
```

But this closer study wasn't close enough. The `OtherClass` object `o` becomes eligible for collection even before the call to `Frob` returns! It is certainly eligible for collection at the point of the `ret` instruction which ends the `Frob` function: At that point, the `Frob` has finished using the object and won't access it again. Although somewhat of a technicality, it does illustrate that

> An object in a block of code can become eligible for collection *during execution of a function it called*.

But let's dig deeper. Suppose that `Frob` looked like this:

```
string Frob(OtherClass o)
{
 string result = FrobColor(o.GetEffectiveColor());
}
```

When does the `OtherClass` object become eligible for collection? We saw above that it is certainly eligible for collection as soon as `FrobColor` returns, because the `Frob` method doesn't use `o` any more after that point. But in fact it is eligible for collection when the call to `GetEffectiveColor` returns—even before the `FrobColor` method is called—because the `Frob` method doesn't use it once it gets the effective color. This illustrates that

> A parameter to a method can become eligible for collection *while the method is still executing*.

But wait, is that the earliest the `OtherClass` object becomes eligible for collection? Suppose that the `OtherClass.GetEffectiveColor` method went like this:

```
Color GetEffectiveColor()
{
 Color color = this.Color;
 for (OtherClass o = this.Parent; o != null; o = o.Parent) {
  color = BlendColors(color, o.Color);
 }
 return color;
}
```

Notice that the method doesn't access any members from its `this` pointer after the assignment `o = this.Parent`. As soon as the method retrieves the object's parent, the object isn't used any more.

```
  push ebp                    ; establish stack frame
  mov ebp, esp
  push esi
  push edi
  mov esi, ecx                ; enregister "this"
  mov edi, [ecx].color        ; color = this.Color // inlined
  jmp looptest
loop:
  mov ecx, edi                ; load first parameter ("color")
  mov edx, [esi].color        ; load second parameter ("o.Color")
  call OtherClass.BlendColors ; BlendColors(color, o.Color)
  mov edi, eax
looptest:
  mov esi, [esi].parent       ; o = this.Parent (or o.Parent) // inlined
  // "this" is now eligible for collection
  test esi, esi               ; if o == null
  jnz loop                    ; then rsetart loop
  mov eax, edi                ; return value
  pop edi
  pop esi
  pop ebp
  ret
```

The last thing we ever do with the `OtherClass` object (presented in the `GetEffective-Color` function by the keyword `this`) is fetch its parent. As soon that's done (indicated at the point of the comment, when the line is reached for the first time), the object becomes eligible for collection. This illustrates the perhaps-surprising result that

> An object can become eligible for collection *during execution of a method on that very object*.

In other words, it is possible for a method to have its `this` object collected out from under it!

A crazy way of thinking of when an object becomes eligible for collection is that it happens once memory corruption in the object would have no effect on the program. (Or, if the object has a finalizer, that memory corruption would affect only the finalizer.) Because if memory corruption would have no effect, then that means you never use the values any more, which means that the memory may as well have been reclaimed out from under you for all you know.

A weird real-world analogy: The garbage collector can collect your diving board as soon as the diver touches it for the last time—even if the diver is still in the air!

A customer encountered the `CallGCKeepAliveWhenUsingNativeResources` FxCop rule and didn't understand how it was possible for the GC to collect an object while one of its methods was still running. "Isn't `this` part of the root set?"

Asking whether any particular value is or is not part of the root set is confusing the definition of garbage collection with its implementation. "Don't think of GC as tracing roots. Think of GC as removing things you aren't using any more."

The customer responded, "Yes, I understand conceptually that it becomes eligible for collection, but how does the garbage collector know that? How does it know that the `this` object is not used any more? Isn't that determined by tracing from the root set?"

Remember, the GC is in cahoots with the JIT. The JIT might decide to "help out" the GC by reusing the stack slot which previously held an object reference, leaving no reference on the stack and therefore no reference in the root set. Even if the reference is left on the stack, the JIT can leave some metadata behind that tells the GC, "If you see the instruction pointer in this range, then ignore the reference in this slot since it's a dead variable," similar to how in unmanaged code on non-x86 platforms, metadata is used to guide structured exception handling. (And besides, the `this` parameter isn't even passed on the stack in the first place.)

The customer replied, "Gotcha. Very cool."

Another customer asked, "Is there a way to get a reference to the instance being called for each frame in the stack? (Static methods excepted, of course.)" A different customer asked roughly the same question, but in a different context: "I want my method to walk up the stack, and if its caller is `OtherClass.Foo`, I want to get the `this` object for `OtherClass.Foo` so I can query additional properties from it." You now know enough to answer these questions yourself.

Bonus: A different customer asked, "The `StackFrame` object lets me get the method that is executing in the stack frame, but how do I get the parameters passed to that method?"