Weak references - how useful are they?

Ask Question

So I've been mulling over some automatic memory management ideas lately - specifically, I've been looking at implementing a memory manager based on reference counting. Of course, everyone knows that circular references kill naive reference counting. The solution: weak references. Personally, I hate using weak references in this way (there are other more intuitive ways of dealing with this, via cycle detection), but it got me thinking: where else could a weak reference be useful?

I figure that there must be some reason they exist, especially in languages with tracing garbage collection, which do not suffer from the cyclic reference pitfall (C# and Java are the ones I'm familiar with, and Java even has three kinds of weak references!). When I've tried to find some solid use-cases for them, though, I pretty much just got ideas like "Use them to implement caches" (I've seen that a few times on SO). I don't like that either, since they rely on the fact that a tracing GC will likely not collect an object immediately after it's no longer strongly referenced, except in low-memory situations. These kinds of cases are absolutely invalid with reference counting GC since an object is destroyed *immediately* after it is no longer referenced (except possibly in the case of cycles).

But that really leaves me wondering: How can a weak reference possibly be useful? If you can't count on it referencing an object, and its not needed for things like breaking cycles, then why use one?

weak-references

edited Dec 20 '17 at 9:34
ParkerHalo
3,492 7 20 41

asked Aug 21 '11 at 5:38
Ken Wayne VanderLinde
13.1k 1 28 59

3 Of course, everyone knows that a reference is a reference at the application level. Either you're using the object in question, or you don't care whether it's garbage collected or not. Weak references are ONLY of interest if you're working at the GC level. Here's a good article, if you're not already familiar with it: weblogs.java.net/blog/2006/05/04/understanding-weak-references – paulsm4 Aug 21 '11 at 5:47

"with reference counting GC since an object is destroyed immediately after it is no longer referenced". In practice, reference counts are decremented when variables fall out of scope which can be a long time after the object they refer to was last referenced. The idea that reference counting destroys objects as early as possible is a common misconception. – Jon Harrop Jan 10 '13 at 20:30

4 Answers

Event handlers are a good use case for weak references. The object that fires events needs a reference to the objects to invoke event handlers on, but you typically don't want the event producer's reference holding to prevent the event consumers from being GC'd. Rather, you'd want the event producer to have a weak reference, and it would then be responsible for checking whether the referenced object was still present.

answered Aug 21 '11 at 5:44
Jacob
53.5k 18 114 187

They'd be a better use case for weak references if there were a delegate type which would not keep the target alive, but would invalidate itself if the target were collected, *and* if Delegate.Combine knew enough about this delegate type to skip invalidated instances when combining delegates. Otherwise the .net event pattern has no good remedy for a scenario where many short-lived objects are created, subscribed to events from long-lived objects, and abandoned. – supercat Dec 11 '11 at 17:40

1 @supercat: Well, this isn't a question about .Net, or about any other platform or language - it's just about places where weak-references could be useful. Ironically, the project I was working on was in C++ (no event model), and it involved both reference counting and event handlers! – Ken Wayne VanderLinde Jan 16 '12 at 19:26

Object referenced by WeakReference will be accessible before the gc process.

So if we wanna have the information of the object as long as it exists, we can use WeakReference. For only when the memory is not enough. So, SoftReference will be used to build global cache usually.

answered Aug 21 '11 at 6:02
user898756

Actually, the last part about SoftReference will not work with reference counting GC as objects are collected regardless of the memory situation. But I do like that you clarified that this works reliably for languages with tracing GC. – Ken Wayne VanderLinde Aug 21 '11 at 6:10

> But that really leaves me wondering: How can a weak reference possibly be useful? If you can't count on it referencing an object, and its not needed for things like breaking cycles, then why use one?

I have an admittedly dogmatic opinion that weak references should actually be the *default* way to persistently store a reference to an object with strong references requiring a more explicit syntax, like so:

```
class Foo
{
    ...
    // Stores a weak reference to bar. 'Foo' does not
    // own bar.
    private Bar bar;

    // Stores a strong reference to 'Baz'. 'Foo' does
    // own Baz.
    private strong Baz baz;
}
```

... meanwhile the reverse for locals inside a function/method:

```
void some_function()
{
    // Stores a strong reference to 'Bar'. It will
    // not be destroyed until it goes out of scope.
    ...

    // Acquire a strong reference to 'Baz'.
    Baz baz = baz_weak;
    if (baz)
    {
        // If 'baz' has not been destroyed,
        // do something with it.
        baz.do_something();
    }
}
```

Horror Story

To understand why I have this firm opinion and why weak references are useful, I'll just share a personal story at my experience in a former company that embraced GC across the board.

This was for a 3D product that dealt with hefty things like meshes and textures, some of which could individually span over a gigabyte in memory. The software revolved around a scene graph and a plugin architecture where any plugin could access the scene graph and elements inside, like textures or meshes or lights or cameras.

Now what happened was that the team and our third party developers were not so familiar with weak references and so we had people storing object references to things in the scene graph left and right. Camera plugins would store a list of strong object references to exclude from camera view. The renderer would store a list of objects to use in rendering like a list of light references. Lights would act similarly to cameras and have exclusion/inclusion lists. Shader plugins would store references to textures they use. The list goes on and on.

I was actually the one that had to do a presentation on the importance of weak references for our team one year into development after we discovered so many leaks, even though I was not the one to push the design decision to use GC (I was actually against this). I also had to implement support for weak references into our proprietary garbage collector after the presentation because our garbage collector (written by someone else) did not even support weak references originally.

Logical Leaks

after clearing the scene, the software could take 3 gigabytes of memory and even more the longer you use it. And this was all because the codebase, including third party developers, failed to use weak references when appropriate.

As a result when the user requested to remove a mesh from a scene, perhaps 9/10 places where references to a given mesh were stored would properly release the reference, setting it to a null reference or removing the reference from a list to allow the garbage collector to collect it. However, there would often be a tenth place that forgot to handle such an event, keeping the mesh in memory until that thing itself was also removed from the scene (and sometimes such things lived outside of the scene and were stored in the application root). And that cascaded sometimes to the point where the software would just consume more and more memory the longer you used it to the point where handler plugins (which stick around even after clearing a scene) would extend the lifetime of the entire scene itself by storing a an injected reference to the scene root for DI, at which point the memory wouldn't be freed even after clearing the entire scene, requiring users to periodically restart the software every hour or two just to get it back to a sane amount of memory usage.

These were not easy bugs to discover. All we could see is that the application was using more and more memory the longer you ran it. It wasn't something we could easily reproduce in short-lived unit or integration tests. And sometimes after hours of exhaustive investigation, we'd discover that it wasn't even our own code that caused these memory leaks. It was inside a third party plugin that users used often where the plugin just ended up storing a reference to something like a mesh or texture that it didn't release in response to a scene removal event.

And that tendency to leak more and more memory tends to be there in software written in garbage-collected languages where the programmers aren't careful to use weak references when appropriate. Weak references should be used ideally in all cases where an object does not own another. There should be far more cases where that makes sense than strong references. It doesn't make sense for every object that references everything to share ownership in everything. For most software, the most sensible design is for one thing in the system to own another, like *"scene graphs own scene objects"*, not *"cameras also own meshes because they refer to them in the camera exclusion list"*.

Scary!

Now GC is very scary in a large-scale, performance-critical software where such logical leaks can could be silently extending the lifetime of an object far longer than appropriate by merely storing an object reference to it and failing to release it in response to the appropriate event(s). Worse, all of this flies under the radar of QA and automated testing.

That's a nightmarish scenario in such a context and the only reasonable way I see to avoid such a scenario is to have a coding standard that relies heavily on weak references if you're using GC or just avoid using GC in the first place.

GC Leaks

Now I have never had the most positive opinion about garbage collection admittedly, and it's because in my field at least, it isn't necessarily more desirable to have a logical resource leak that flies under the radar of testing over, say, a dangling pointer crash which can easily detected and reproduced and most likely corrected by the developer before he even commits his code if there's a sound testing and CI procedure.

In my particular case, the most desirable kind of bugs if we're choosing among evils are the ones most easy to discover and reproduce, and GC-type resource leaks are not easy to discover and not easy to reproduce in any sense that helps you discover the source of that leak.

However, my opinion about GC turns a lot more favorable among teams and codebases that make heavy use of weak references and only use strong references where it makes actual sense from a high-level design standpoint to extend the lifetime of an object.

GC isn't a practical defense against memory leaks, quite the opposite. If it was, the least leaky applications in the world would be written in languages that support GC like Flash, Java, JavaScript, C#, and the leakiest software imaginable would be written in languages with the most manual memory management like C, at which point the Linux kernel should be one hell of a leaky operating system which would require restarting every hour or two to reduce memory usage. But that's not the case. It's often quite the opposite with the leakiest applications being written against GC, and that's because GC actually tends to make it harder to avoid logical leaks. Where it does help is avoid physical leaks (but physical leaks are easy enough to detect and avoid in the first place no matter what language you use) and where it does help is to prevent dangling pointer crashes in mission-critical software where it's more desirable to leak memory than to crash because a person's life is at stake or because a crash could translate into a server being unavailable for hours on end. I don't work in mission-critical

```
class Foo
{
     // This makes 'Foo' instances cause 'bar' to leak, preventing
     // it from being destroyed until the 'Foo' instances are also
     // destroyed unless the 'Foo' instances set this to a null
     // reference at the right time (ex: when the user requests
     // to remove whatever Bar is from the software).
     private Bar bar;
}
```

... but weak references don't risk this problem. When you're looking at millions of LOC like the above on one hand and epic memory leaks on the other, it's quite a nightmare scenario when you have to then investigate which analogical `Foo` failed to set the analogical `Bar` to a null reference at the appropriate time because this is the part that's so scary: the code works just fine as long as you ignore gigabytes of memory leaking. Nothing triggers any kind of error/exception, assertion failure, etc. Nothing crashes. All the unit and integration pass without complaint. It all just works except that it's leaking gigabytes of memory causing user complaints left and right while the whole team is scratching their heads about what parts of the codebase are leaky and what aren't while QA tries to do damage control by pragmatically suggesting that users save their work and restart the software every half hour as though that's supposed to be some kind of solution.

Weak References Help a Lot

So please make use of weak references whenever appropriate, and by appropriate, I mean when it doesn't make sense for an object to share ownership in another.

They're useful because you can still detect when an object has been destroyed without extending its lifetime. Strong references are useful when you genuinely need to extend the lifetime of an object, such as inside a short-lived thread so that the object isn't destroyed before the thread is finished processing it, or inside an object that *genuinely* makes sense to own another.

Using my scene graph example, a camera exclusion list does not need to own scene objects already owned by the scene graph. Logically that makes no sense if it did. If we're at the drawing board, no one should think, "yes, cameras should also *own* scene objects in addition to the scene graph itself."

It only needs those references to be able to easily refer back to those elements. When it does, it can

If the camera wants to use a convenient lazy kind of implementation that doesn't have to bother with scene removal events, then weak references at least allow it to do that without leaking epic amounts of memory all over the place. The weak references still allow it to discover, in hindsight, when objects have been removed from the scene and maybe remove the destroyed weak references from the list then without bothering with scene removal events. The ideal solution to me is to use both weak references and also handle scene removal events, but at the very least the camera exclusion list should be using weak references, not strong references.

The Usefulness of Weak References in a Team Environment

And that gets to the heart of the usefulness of weak references to me. They are never absolutely required if every developer on your team thoroughly removes/nulls out object references at appropriate times in response to the appropriate events. But in large teams at least, the bugs that can occur which are not prevented outright by the engineering standards will often end up occurring, and sometimes at staggering rates. And there weak references are a fantastic defense against the tendency for applications revolving around GC to have logical leaks the longer you run them. They're a defensive mechanism in my eyes to help translate bugs that would manifest themselves in the form of hard-to-detect memory leaks into easy-to-detect uses of an invalid reference to an object already destroyed.

Safety

They might not seem so useful in the same sense that an assembly programmer might not find much use for type safety. After all, he can do everything he needs with just raw bits and bytes and the appropriate assembly instructions. However, type safety helps to detect human errors more easily by making the human developers more explicitly express what they want to do and constraining what they're allowed to do with a particular type. I see weak references in a similar sense. They help detect human errors that would have otherwise lead to resource leaks if weak references weren't used. It is deliberately imposing constraints on yourself like, *"Okay, this is weak reference to an object so it cannot possibly extend its lifetime and cause a logical leak"* which is inconvenient, but so is type safety to an assembly programmer. It can still help prevent some very nasty bugs.

They're a language safety feature if you ask me and like any safety feature, it's not absolutely required and you generally won't appreciate it until you encounter a team tripping over the same things over and over because such a safety feature was lacking or not adequately used. For solo developers, safety is often one of the easiest things to ignore since if you're competent and careful, you genuinely might not lays down safe engineering practices with an iron fist that, within just a month, you could have accumulated over a hundred thousands lines of extremely buggy code with obscure, hard-to-detect bugs like the GC logical leaks mentioned above. The amount of broken code that can accumulate in just a month absent a standard to prevent the common bugs is quite staggering.

Anyway, I'm admittedly a bit dogmatic about this subject but the opinion was formed over a boatload of epic memory leaks, for which the only answer I saw short of just saying to developers, "Be more careful! You guys are leaking memory like crazy!" was to get them to use weak references more often, at which point any carelessness would not translate into epic amounts of memory leaked. It actually got to the point where we discovered so many leaky places in hindsight that flew under the radar of testing that I deliberately broke backwards source compatibility (though not binary compatibility) in our SDK. We used to have a convention like this:

```
typedef Strong<Mesh> MeshRef;
typedef Weak<Mesh> MeshWeakRef;
```

... this was a proprietary GC implemented in C++ running in a separate thread. I changed it to this:

```
typedef Weak<Mesh> MeshRef;
typedef Strong<Mesh> MeshStrongRef;
```

... and that simple change in syntax and naming convention helped enormously to prevent more leaks except we did it a couple years too late, making it damage control more than anything else.

edited Jan 2 at 11:57

answered Jan 1 at 9:47

Team Upvote
8,717 2 23 57

Thank you for the very interesting read. – R Tsch May 17 at 16:24

I often use `WeakReference` in conjunction with `ThreadLocal` or `InheritableThreadLocal` . If we want a value to be accessible to a number of threads while it is meaningful, but then later remove the value from those threads, we can't actually free the memory ourselves because there is no way to tamper value ceases to be meaningful, you can ask the worker thread to remove the hard reference, which causes the values in all other threads to be immediately enqueued for garbage collection (although they may not be garbage-collected immediately so it's worthwhile to have some other way to prevent the value from being accessed).

answered Jan 30 '17 at 20:22
Kidburla
1,627 1 26 47

Re "..when the value is created..", Why do you need to ensure that its at the creation stage? Couldn't you pass the object reference to the other threads after creation? – Pacerier Sep 19 '17 at 16:46

@Pacerier yes, if you had access to the code running in those other threads to inform those threads to expect a value and add it to their `ThreadLocal` . However, often we don't have access to the code of those threads, because they were created by some third-party library (for example logging library, non-blocking IO library etc). We can't pass any message to those threads or tell them what to do after they have started. However we can tell them what to do when they start, via the `InheritableThreadLocal.childValue` method (or indeed just the default behaviour of `InheritableThreadLocal`). – Kidburla Sep 20 '17 at 18:39