Musing Mortoray

PHILOSOPHY

Why Garbage Collection is not Necessary and actually Harmful

ON 2011-03-30 • (44 COMMENTS)

In the world of new languages it seems like garbage collection is standard feature. A way for the runtime to locate unused bits of memory and release them. It has become so common that some people can't even imagine using a language without it. Programmers, particularly ones new to the trade, tend to be exposed only to the virtues whereas the negatives are quietly left in the dark. Perhaps this was a decent reaction to the memory management of C, but as a whole, in general purpose languages, garbage collection is more of a detriment than a benefit.

What is It?

When we speak of garbage collection most people think of the Java style mark-and-sweep, tracing, copying or other scanning like algorithms. Something continually, or regularly, searches the memory of the program looking for items that can be deleted. That is, something is looking for *garbage* to *collect* and discard.

Though often falling under the same umbrella term, we don't simply mean systems where memory is automatically freed when it is no longer used. It is invalid to refer to properly tracked and managed memory as *garbage* in the same sense. We are speaking strictly of the search and destroy type algorithms. Things like reference counting and process cleanup are simply not the same. They aren't looking for things to delete, they know exactly what has to be removed.

Solves only one problem

Garbage collection is a method to free the programmer from worrying about basic memory management. It isn't the only option; simple reference counting also covers the vast majority of memory management needs. Sweeping collection has a distinct feature of being able to resolve cyclic dependencies. That is when A points to B and B points to A. Reference counting cannot manage this in the general case.

Obviously if you have such a feature you will make use of it. I don't see however that cyclic dependencies are a significant obstacle in most programs. Such object relationships can normally be resolved, or significantly mitigated, by one of a number of other techniques.

Does not solve that problem

Even with a garbage collector cyclic references still pose somewhat of a problem. Often objects need to execute some code when they are to be destroyed, they can't just silently disappear like a block of memory. Within a cyclic reference it is, in the general sense, impossible to determine which destructor has to be called first. So instead garbage collectors introduce the concept of finalization, which is a rather marginalized form of a destructor.

Cyclic loops with special cleanup dependencies need to have a programmer clarify what should be done. A finalizer system could possibly get it wrong, thus forcing a programmer to manage the cleanup prior to garbage collection. In situations where the collector can assuredly do the right thing, there is likely an easy alternative for the programmer to avoid the cycle all together.

Requires a simplified memory model

Scanning memory for unused objects requires a very clear idea of what memory actually is. The collector has to to understand all the places where a pointer might be stored. In many high-level languages this is possible solely because the memory model has been simplified to just a stack and a heap. Computers however have all sorts of other places where pointers could be stored. They have shared memory, specialized vector processors, registers, graphics texture memory, in fact almost every chip in the system has some manner in which to store and retrieve data.

It's simply not feasible, or even possible, for a garbage collector to properly scan all this memory. Programs that need these resources have to be aware of its implications on the garbage collector. Perhaps the programmer won't use such memory, but those same implications also apply to a compiler trying to optimize the code. Furthermore, the simplified model requires hiding the true nature of memory making it more difficult to implement certain inter-process and concurrent programming algorithms.

It holds on to resources too long

Object based languages tend to use a concept known as RAII (resource acquisition is initialization). The simple idea is that, by example, creating a **File** object will open the file for you and close it once you stop using the object. With scoped based variables, or reference counting, this is great since the moment the last user of the file is done, it will be closed. With a garbage collection you need to wait for the scanner to find and finalize the object before the file is closed.

This results in a late deallocation. For many resources this causes an exhaustion issue. An OS has a limit on the number of open files. A database can only have so many open cursors. Client sockets are limited, and likely the server will throttle as well. A network protocol may send a packet to end the connection. The file object will flush the data to the disk. It's impossible to say in a generic manner what might need to happen at this point.

So in Java, and other such languages, you'll often see programmers explicitly calling **close** functions on their objects. This harks a lot back to the idea of manual memory management, the one thing that garbage collection was supposed to avoid. Prompt deallocation of unused resources, via the RAII pattern, is critical. Garbage collection can't offer it.

It doesn't prevent memory leaks

Most memory leaks I encounter are not usually the result of a missing **free** or delete statement. Usually they result because somebody is storing objects in a cache. Nothing ever clears this cache so over time it grows. It hoards every object added to it until the program exits or runs out of memory. Garbage collection

can’t do anything here. Those objects are technically in use and thus cannot be freed.

Garbage collection can prevent certain types of memory leaks. But so can reference counting. More so these types of memory leaks are very easy to recognize by readily accessible leak analysis programs. The type of program you’ll want to run to catch the above problems as well, even if you have a garbage collector.

It is slow and doesn’t scale

Compared to manual memory management, or reference counting, or virtually any other non-scanning scheme, garbage collection is slow. This is easy to forget since you can find any number of benchmarks or studies showing that it is just as fast as *something else*. Look closely, or even a cursory glance in most cases, and you’ll see what is actually being measured is nonsensical, oversimplified, or just plain wrong.

Actually, you don’t even need to look closely, you can just view it theoretically. Garbage collection has to scan memory, thus its complexity must be a function of N, where N may be the number of bytes or the number of objects. It must be bound by a linear function of N, since ultimately it needs to scan all of these objects. Even a highly sophisticated, incremental, distributed collector must go through all the objects. This means that every additional block of memory used by a program will ultimately make it slower.

It pauses and interrupts

Just being slow or consuming extra processing time is not a significant issue for many programs. Collectors have this an additional drawback of occasionally locking the memory. Locking the memory in most cases ends up locking several threads, or the whole program. With a solid design it is possible to significantly reduce the frequency and duration of interruptions. None are perfect however and they all end up blocking the program for some noticeable period of time.

What is noticeable depends on the application. Obviously for a real-time, or low-latency application, even a few microseconds pose a significant problem. Most apparent however are user applications. We have all sorts of programs which simply stop

reacting for a while, video which pauses in the middle, audio with snaps and crackles, web pages that sometimes take 10 times as long to load, and the list goes on. This is a very unfortunate backwards step in user experience.

Conclusion

Nobody is denying that the original style of **allocate** and **deallocate** leads to problems. Though not all people are however opposed to that style, I think in general we can agree that some kind of automatic memory management is required. Looking at its set of features and drawbacks, scanning garbage collection just isn't the right approach for a general purpose language. Recall in fact that garbage collection has only one unique feature: the ability to resolve cyclic loops. A problem that it can't actually solve correctly and one that has several other alternate solutions.

Perhaps certain domain-specific-languages could use scanning garbage collection as a reasonable way to implement their features. Some languages might consider a hybrid approach. Maybe an ingenious new technique will solve some of the drawbacks. So far however, garbage collection in generic languages has introduced more problems than it has solved.

44 replies »

“I don’t see however that cyclic dependencies are a significant obstacle in programs. Such object relationships can normally be resolved, or significantly mitigated, by one of a number of other techniques.” Reply

Perl 5 is probably the most-used language that has reference-counting as the memory management solution. Speaking as a Perl expert, it is a serious pain in the neck to deal with breaking cycles. They are a very common and appropriate solution to many data modeling needs and are a common cause for bugs, particularly in persistent programs such as web applications. A proper mark-and-sweep, preferably in some background thread, is the correct and appropriate solution.

What you are railing against, and rightly so, is a pre-emptive GC. Parrot, the new VM for Perl 6, doesn’t use a pre-emptive GC like the JVM does. In large part because pre-emption is considered harmful for the very reasons you’ve stated.

> Cyclic-dependencies are likely very domain specific, so how many one encounters could vary a lot. My reservations are there though, in some cases a cyclic dependency is a major pain to resolve. In certain data models GC’ing is also not a big issue: if an object doesn’t need a non-trivial destructor GC’ing is a simple affair. I just don’t like GC being the default technique in some languages, as I find the problems I outlined far outweigh the value in it. If a GC technique comes along that can solve the problems I wouldn’t be against it. Perhaps I’ll take a look at what Perl 6 does in this regards.

“That is, something is looking for garbage to collect and discard.” Reply

That is not how most modern JVM/.Net GC’s work.

They do not look for garbage, they look for *live objects* and assume the rest is okay to use (if they are moving collectors then this is trivially achieved as a side effect of the admittedly expensive movement”

I’d love to know how you would implement a language with full lexical closures (where you can return said functions from another one, so not C++’s lambdas) without GC…

From my point of view full lexical closures is sufficient benefit to most code I write that this making the runtime support GC is more than worth it.

> I think the semantics of whether they look for garbage or look for live objects an implementation detail. Somehow they have to clean up the garbage, this involves looking or tracking. I hope they use the most efficient method suitable to the language, but they cannot avoid that the more memory you use the more time the GC will take. Are there any collectors that have a complexity related to the depth of a dependency chain rather than the total object count?

> I don't think lexical closures themselves are a problem, the moment you allow them to refer to the encapsulate object is where a cyclic reference becomes an issue. In most cases I think reference counting would still work. I'll admit it is possible that from casual use of such structures/syntax you might get a cyclic link. But I'd prefer to see an example of how common this might be and how hard it would be to break the cycle.
>
> Obviously one's viewpoint is an issue. From my experience I've seen that a lack of RAII causing more problems than a lack of memory deallocation. Besides, as I pointed out, GC'ing doesn't solve a lot of memory problems. Perhaps it makes using a few features easier, but the cost is still very high compared to reference counting.

Reply

While I do prefer RAII (or similar patterns) where possible, I find refer counting outdated, not only because it's unable to work with cycles, it's also potentially very slow.

There is a good discussion at the following blog. I'd recommend taking a look at that.
http://flyingfrogblog.blogspot.com/2011/01/boosts-sharedptr-up-to-10-slower-than.html (http://flyingfrogblog.blogspot.com/2011/01/boosts-sharedptr-up-to-10-slower-than.html)

> This is one type of those suspicious benchmarks I noted in the article. The description gives no clue as to what they were actually doing. You have no context as to what type of application this is and whether the situation they are testing is realistic. It is really no effort to come up with specialised benchmarks which show one system behaving better than the other. Thus the results of this benchmark should not be considered.
>
> In any system that simply allocates and deallocates memory GC'ing should be faster. This is because any time you enter the GC section you are going to find and free a lot of memory. However, in a long-lived system, the vast majority of time you enter the GC you won't find anything at all to free, or only a little bit. That is, the GC spends a lot of time scanning and rescanning the same set of objects and finding very little to free up. So while the mechanics of individual reference counting might be slower, they never waste time.
>
> These types of benchmarks also never address the stop-the-world problem. One of the biggest problems with GC speed is that at times your program just freezes while it does something. This is a critical aspect. In many domains this is simply unacceptable — and unfortunately GC is often used in these domains. If there is a GC technology that doesn't do this then it'd go a long way to improving the applicability of GC.

I think your intuition of the cost of a modern concurrent ephemeral garbage collector isn't accurate. Gen-0 collections happen often, aren't noticeable, and recollect a lot. Collections in older generations don't happen as often, and when they do, they still let threads run most of the time.

I wrote a game using .Net and XNA and ran it on both the PC and the Xbox 360, which made clear the difference in performance between a naive garbage collector (as available on the Xbox), and a generational one (as available on the PC). On the PC, the impact of GC wasn't any more noticeable than the impact of multi-tasking. On the Xbox, the impact was such that it deeply impacted the design and the looks of the code.

> I don't doubt that there are very good garbage collectors, I expect that. But can you point to one that gives me a guarantee that it won't interrupt my processing? There are lot of domains of software where even a momentary pause (sub-millisecond) is essentially a defect. There are also several domains where it just doesn't matter, thus this aspect of GC is acceptable in those domains.
>
> However, even with a guarantee of non-interruption I won't have all of my points addressed: just one of them will be answered.

"Are there any collectors that have a complexity related to the depth of Reply
dependency chain rather than the total object count?"

Loads, that's pretty much the whole point of 'look for live objects not dead'. You are taking your way of thinking "delete does something in my world so freeing up memory must require some action per thing" when it doesn't require it in other models. If you have a moving GC, and you only ever move reachable objects then, once you have moved all the reachable objects into a compact region (trivial – the pointer rewriting and write barriers required to make that trivial are not of course), then guess what 'freeing' all those dead objects involves?

Changing a single pointer for the top of the heap. Job done.

Thus a very high rate of allocation can still be faster with GC (so long as the live set is small) than other techniques because allocation is simply a pointer increment each time, deallocation is the live set traversal (again, perhaps very fast) and a single pointer change.

There are of course ways you can write things to make GC very expensive (pointer heavy code isn't great for example) but many such examples have crappy behavior on modern machines due to their lack of spatial locality anyway.

As to closures if you have to wrap every function type in some smart pointer I don't regard this as overly pleasant, but is in theory doable (being able to overload () at least makes it a little nicer). But yes it becomes incredibly easy to get a cycle.
Composition becomes tricky though. The compiler would have to become aware that it was capturing an instance that required reference count delete cascades and supply the relevant glue. Are you Talking of some theoretical C++ compiler that would do this? If none exist it does suggest that it is not so feasible as you think, though I'd love to see an example against.

“But can you point to one that gives me a guarantee that it won’t interr Reply
processing?”

You mean hard realtime, soft realtime or probablistic non pausing of other threads?

hard is problematic, because basically you don’t want to be dynamically allocating in those sorts of scenarios anyway but there has been academic work on it.

Soft then there’s *products* out there for it
http://www-01.ibm.com/software/webservers/realtime/# (http://www-01.ibm.com/software/webservers/realtime/#)

If you completely change much of your paradigm you can do some very nice stuff:
http://prog21.dadgum.com/16.html (http://prog21.dadgum.com/16.html)
To say that erlang is used in places with serious latency requirements is an understatement.

Obviously if you want determinism then the more control you take over the behavior the more deterministic you can be. I guarantee that for the vast majority of coders (myself included) that means the less clever the code that is running is and overall you will get worse throughput, but you do indeed make it predictable and will likely reduce worst case latency (with some potential cost to average latency).

If those things matter by all means don’t use GC (or use a platform where you gain greater control over GC).

I think many of your responders are commenting because you are crit Reply
something you do not understand. I would not critique many aspects of (say) C++ that annoy me because I do not understand C++ well enough. I might *question* why they were there, why they were allowed to persist, but I would not start a post with “Foo considered Harmful” unless I felt I had a very serious handle on exactly what I was talking about (to the extent that I could implement the feature, and several of its alternates).

I’d still expect to be wrong, at least in part.

> My experience with garbage collection is primarily limited to Java (I suppose some Flash as well). I do look at other languages, but I don’t have infinite exposure obviously.
>
> I’m always open for debate, that is the point of the article. I see value in GC, but I’m playing strong devil’s advocate here. And of all the comments I’ve got, none have addressed all of my concerns presented in article. Only the speed of GC and the interruption time have been addressed. The other critiques still stand there without criticism/reply.

Reply

You have asserted things like: “Often objects need to execute some code when they are to be destroyed, they can’t just silently disappear like a block of memory. ”

This is very bogus in the vast majority of applications (the number of types that require it is actually tiny), the number of instances of them is almost certainly tiny in comparison to the addressable memory space of the application (try creating file handles without deleting them, you’ll run out waaay faster than you run out of heap) and if you need a deterministic order of ‘non memory cleanup” (I will hereafter refer to this as disposal and it has nothing to do with the memory being made available again though obviously that must happen after) then the program needs to be written with that in mind anyway to be correct. At that point it is far better to wrap those things up into one thing which is responsiple for this invariant (DRY) and use that one thing. At this point you dispose of one thing and have it dispose of the others. whatever well defined structure that would allow reference counting to work can be applied to the disposal procedure (if it’s stack bound then helpfully you don’t need either and just do the moral equivalent of c#’s using or RAII if it’s supported and you’re done).

If you make obviously wrong statements why would you expect people to provide detailed rebuttals?

your RAII only works if *all* pointers to the object are smart, or it is stack allocated. In the stack allocated world you do indeed get a nice simple idiom (RAII) where as languages without explicit stack allocation (with associated destructor calls done by the compiler on your behalf) require the explict insertion of something to do it for you (in java this boiler plate is annoying, if find ‘using’ in c# pleasant since it highlights the unmanaged resource – I can appreciate some people may not) If you use shared pointers you can have something have an indeterminate lifetime but with guaranteed release of the resources when the last reference goes away, at the cost of using the right pointer wrapper everywhere. this is nice I admit. I would argue that it is relatively uncommon to require such arbitrary lifetime objects, Since it implies shared ownership of something that should not have indeterminate lifespan anyway. It is far nicer to link the lifetime of such a resource to something more concrete (a session, a thread, an async call) and where this is not great in performance terms wrap the objects with pooling mechanisms which extend the actual lifespan but retain the logical view to the rest of the world (there are issues with this of course but it is quite commonly applicable).

What objects do people s=usually use that own non memory resources? file handles, socket handles, database handles (in many cases a special case of the sockets).

Ultimately there’s nothing stopping you doing the equivalent of auto pointer wrappers in something like c#/java (nicer in c# thanks to reified generics) and making Dispose do the pointer decrement and real dispose on 0. Obviously this adds the slowness to it inherent in such measures, and is far less pleasant in terms of the intrusiveness of the wrapper to your code) and is a runtime guard not a compile time guard (though the C++ solution does not compile time

protect you from the accidental use of no wrapping pointer, nor using the wrong one so neither is perfect).

Perhaps an ideal would be to allow ref counting to be injected into the type system to support automatic deterministic disposal coupled with finalization within the GC for occasions when we mess it up with cycles.

That would be an interesting experiment, you would pay the cost of ref counting only for those types with deterministic requirements, with a slight additional cost for those that were stack bound, but then again such things are likely to be expensive by their very nature so I doubt this would even feature much…

> I find the RAII pattern to be cleaner than the dispose pattern in many cases. This is from my Java experience where I have seen a great deal of code that "forgets" to call dispose. With stack-based objects and RAII you can not forget to call the destructor. I find this a very unfortunate loss of GC based languages. And since there are not stack based objects you can't actually write an auto_ptr like solution — you will always need to explicity call the "dispose" function. I even did a quick search and the first response for "dispose pattern" is at Microsoft and it's example (http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx) fails to cleanup correctly in the case of an exception!
>
> I don't avoid the dispose pattern completely though. I agree that the moment you enter the realm of shared pointers in C++ you should start to become leery of the RAII pattern. Since no path of code really has any idea of the state of the object, a delayed destructor is just as inevitable as a delayed finalization.
>
> Your ideal is also my ideal. I want a compiler intelligent enough to figure out what type of memory management is needed. Then I want keywords that will indicate which type of resources may not belong to a cycle: ones that require deterministic cleanup.

"fails to cleanup correctly in the case of an exception!"

Yes the example should have used either a using or a finally block, but in fairness the *process* is exiting at that point so it doesn't matter in that context (still poor but then again so is plenty of other documentation).

You do not incidentally always need to call Dispose explicitly, the using construct in c# deals with the syntactic sugar of the finally block and additional scope of the variable (something that did indeed annoy me in java) and because Dispose is backed into the BCL other sugars like foreach guarantee to invoke it if present.

Also there is nothing specific to GC that forces this. It is entirely possible to have deterministic disposal with GC memory management, see C++/CLI (http://msdn.microsoft.com/en-us/library/ms177197.aspx (http://msdn.microsoft.com/en-us/library/ms177197.aspx)).

> Johann is quite right, conflating GC with deterministic release of other resources is the problem. No such restriction exists in theory (or reality as demonstrated in C++/CLI) but it is certainly true to say that java still makes it painful (and c# is far from perfect).

> Note that Java 7 introduced a try-with-resources (http://docs.oracle.com/javase/7/docs/technotes/guides/language/try-with-resources.html) statement which provides the functionality of C#'s 'using', which is even in some ways superior. Yet, it is still inferior to C++'s destructor.

I wrote a more complete answer earlier, but it seems it got "collected" [illegible] Reply
reaching the server…

"Solves only one problem": GC solves memory management. That's fine with me.

"Does not solve that problem": GC does not solve other resource management, such as e.g. file descriptors. That should not be held against it.

"Requires a simplified memory model": Fine with me, that's the model I'm exposed to (and I guess 90% of all programmers who I hope will never be allowed to write C/C++ code that I might have to maintain).

"It holds on to resources too long": That's misleading. GC does not "hold" resources other than memory. What you meant is that using using GC for resource management is a bad idea, and everybody agrees on that. That's what the dispose pattern and RAII are supposed to be used for. GC is only for *memory* management. The fact that RAII and memory management are so tightly coupled in C++ is confusing many, I think. It does not have to be that way.

"It doesn't prevent memory leaks": Technically speaking it does. What it doesn't do is automatically fix a design that artificially holds on to memory longer than intended. But until computers can guess what we intend, not much can be done about the problem.

"It is slow and doesn't scale": True. Some applications simply can't afford the costs of a VM and a GC, but they are becoming fewer and fewer, relatively to the growing number of applications.

"It pauses and interrupts": So does my multi-tasking OS, and it's never bothered me too much.

> A GC prevents the RAII pattern. In particular, the lack of scope based automatic destruction interferes with the type of resource management I have referenced. I find this is a valid argument to hold against GC.
>
> In my opinion, and I doubt I'm alone here, a memory leak is any time an application retains memory for significantly beyond the time it is needed. Holding unneeded objects in a cache fully qualifies in my mind as a leak. Perhaps our semantics differ, but surely you agree this **is** a memory issue.

> And GC, as you agree, cannot solve this. Thus my point stands, GC is not a complete solution for memory management.
>
> But let's look at the main point here. You mention a lot about certain types of applications, and the previous comment also uses the phrase "vast majority of applications". Let's put aside disagreement about what most applications are, and for a second just assume that GC works the majority of the time. We still have a lot of programming where GC can not be used. If the trend is that all new languages use GC that means those systems will be relegated to using C and C++ indefinitely. Do you want these two languages to stick around forever? I don't, but GC is one of the things that prevents migrating a lot of systems to a new language.

> "A GC prevents the RAII pattern".
No it does not. Take C#, and make the two following changes:

1. IDisposable is taken away and Object has a method Dispose() instead.

2. All variable declarations are now done through "using() {}".

How is that not RAII? The designers of C# chose to have it another way, but I don't think GC was the reason.

> "a memory leak is any time an application retains memory for significantly beyond the time it is needed".
What does "needed" mean? Let me suggest two definitions:

1. a resource R in a program P is not needed at time T if the state of P at T has no references to R.

2. a resource R in a program P is not needed at time T if there exists a program Q which is observably identical to P where Q at time T holds no references to R.

According to definition 1, a dialog that has been closed by the user but is kept in memory by an event handler is not a leak. It is however according to definition 2. The problem is that solving leaks as defined by 2 is a very difficult problem to solve.

A leak according to definition 2 is a design defect, and as such, should be caught by a design verification tool. It may involve a mix of static analysis and run-time checks, which are probably too costly to be affordable in normal use. The good news are, there are such tools, see for instance RedGate's and JetBrains' memory profilers.

> "Do you want these two languages to stick around forever? I don't, but GC is one of the things that prevents migrating a lot of systems to a new language."
I actually agree with you on that point. What we need is a platform that gives programmers full control over GC, and possibly mix different schemes within an application. This would probably require to have clear separation between memory regions, preventing references across regions.
Applications would be built around processes that communicate by message

passing.
In a sense, we already have that. .Net or Java for the region using GC, interfacing through interop with native code.

> The “using” directive is a nice syntactic thing to avoid needing to do “finalize” block with calls to “dispose”. I don’t agree that it is a good solution to the problem of resource management. The key difference between this, and C++ RAII is **who** decides how the object should be handled. In the C# case the caller must still remember they should free up the object right away, in the C++ case the class definition gets to enforce the pattern in normal use.
>
> Please keep in mind I’m not trying to defend C++ here, I’m trying to defend the pattern. Even more so I’m trying to indicate that delayed resource deallocation is a bad thing. “Dispose” and “RAII” both help, but aren’t the ultimate solution. I will say however the C#’s Dispose system is far superior to Java (which I’m more familiar with). The docs clearly show how the “Dispose” method will be called. One of my criticisms of Java’s finalize is that you can’t do anything in the method, since your held objects are in an undefined state, it appears C# clears this up and says your objects still exist during Dispose.

“A GC prevents the RAII pattern.” Reply

No it doesn’t. (As evidenced above). What you are disliking is the inability in many managed languages (the most popular imperative ones being c# and java) to allocate on the stack and have the destructor call inserted for you.

Yes there is value in being able to do this in some cases but it means any time you aren’t putting it on the stack you must remember to use the (right) smart pointer and avoid cycles or it’s either freed early or never. Conversely in, say c#, you must in the stack bound case remember to put a using block in, forgetting will lead to a lack of freeing until later, in the complex cases it likewise will not free till later.
This isn’t great (especially when cargo cult programmers start inserting calls to GC.Collect() at is “fixes” the resource leaks) but for reasonable programmers is just fine (it’s also trivial to diagnose, and easy to fix).

“In my opinion, and I doubt I’m alone here, a memory leak is any time an application retains memory for significantly beyond the time it is needed. Holding unneeded objects in a cache fully qualifies in my mind as a leak.” Reply

A leak is when memory is unreclaimable. A ‘managed leak’ is when you hold onto memory *needlessly* thus keeping it alive.

The two are vastly different (one requires tools like valgrind to find, the other is trivial to spot with a basic debugger).

I do *not* agere that GC doesn’t solve this. It’s not going to be right in all situations but implying that memory needs freeing up the moment you

personally are finished with it is *worse* in most cases for performance. Heck why bother at all in short running applications, the process is going away anyway!

Perhaps you are annoyed by many jvm's refusal to return allocated memory to the OS if memory pressure reduces? If so you would again be conflating a concept with _implementation_ Some JVMs do return the memory back (or can be configured to) the .Net CLR returns it.

You have gone from an article saying "GC considered harmful" to saying "GC is not a complete solution for memory management." i.e. some apps (or indeed whole families of them) are not appropriate for managed languages. Well *duh* of course they aren't. No one ever said they were (to imply that we think this is a straw man argument)

> I disagree on your (and Johann's) definition of leak. Who cares what the implementation is actually doing. From a user's perspective is anything that will cause the program to continue to consume more and more memory over time without increasing the program's workload. If Firefox continues to use more memory as I browse that is a memory leak — I don't care what is the cause of that leak. Thus object's stuck in the cache which will never be used again, are leaked objects.
>
> I don't care that the VM doesn't return memory to the OS. The typical malloc implementation on Linux won't do this either for a C++ program (with the exception of large blocks of memory).
>
> Your last statement about my change in position is misleading. Not all my concerns about the GC are related to managing memory at all.

Johann Deneux said:

Reply

> Take C#, and make the two following changes:
>
> 1. IDisposable is taken away and Object has a method Dispose() instead.
>
> 2. All variable declarations are now done through "using() {}".
>
> How is that not RAII? The designers of C# chose to have it another way, but I don't think GC was the reason.

I think this is underestimating the power of RAII as used in C++. The pattern of acquiring resources in constructor and releasing them in destructor is utilized not only in automatic (stack-based) objects. Such RAII objects can be used as base classes, or class members. Or can be returned form factory functions by means of move constructor (http://akrzemi1.wordpress.com/2011/08/11/move-constructor/). In neither of this cases is it possible to find a clear scope enclosed by curly braces. And the C#'s 'using' and Java's try-with-resources just do not work there.

The difference between C++ and ‘managed languages’ is that the former treats memory as just another resource; the latter make the special case for memory resource. There are valid reasons for making this distinction, but there are also good reasons for treating all resources uniformly.

Reply

I did not use GC much, but I believe that GC is not primarily about avoi memory leaks. If a platform provides a mandatory GC it has the luxury of disallowing any other means of manual memory management. If programmers cannot manually manage memory they cannot make any memory-related bugs. Memory leak is the least of such bugs. The real memory bugs in C++ and C applications are that once a month or so, the function that normally runs fine, returns rubbish data or crashes. Or function f() crashes in production, but if you wrap it with logging statements it works fine; also it works fine when you debug. But once you remove the logging statements it starts crashing again magically — because someone in an unrelated place exceeded the array index by one.
The problem in languages like C and C++ is that they are, in some way, wrappers over an assembly language. You have access to raw memory. While types from the file system tell you how to interpret the portions of memory, due to unsafe features, this mechanism can be messed up and the same address in memory is interpreted by **struct Rational** in one place and as **int** in another place at the same time. This makes these languages *memory-unsafe*. In contrast, languages with mandatory GC make the code memory-safe (because they can afford to disallow any manual memory management).

So now you have a trade-off between a memory-safe system and a fast, RAII-enabled system. I guess both choices are valid.

> Safe memory is perhaps a feature of such languages that I tend to overlook. I’m not sure how much this truly relates to the GC however, as a non-GC language could offer safe memory — you’d have to lose some lower level memory constructs. Additionally, a GC based system doesn’t guarantee memory-safety on its own. Consider that GC for C++ is possible, yet you can still trash your memory to your heart’s content.
>
> We also can’t forget that Java still has things like null pointers and containers which can contain invalid types. I’d say this is very similar to corrupting your memory. Though at least with Java you have a very good idea of where the problem is, rather than in C where memory corruption can occur from anywhere at any time.
>
> I agree safe memory is good, but I don’t think GC is required for this, nor does GC completely achieve it. Totally safe memory comes with a performance cost (GC aside). I wonder if its possible to get C-level performance with safely managed memory?

Reply

“Safe memory is perhaps a feature of such languages that I tend to ove
I’m not sure how much this truly relates to the GC however, as a non-GC
language could offer safe memory — you’d have to lose some lower level

memory constructs. Additionally, a GC based system doesn't guarantee memory-safety on its own. Consider that GC for C++ is possible, yet you can still trash your memory to your heart's content "

This is how I see it. GC is a proven tool for enabling memory-safety because if you provide it, you can disallow any other memory-related operations and say "use GC instead". It is an acceptable (not for everyone, though) alternative to unsafe memory-ops. And I agree with you: an "optional GC for C++" would not be able to offer this sort of memory-safety.

"We also can't forget that Java still has things like null pointers and containers which can contain invalid types. I'd say this is very similar to corrupting your memory. Though at least with Java you have a very good idea of where the problem is, rather than in C where memory corruption can occur from anywhere at any time. "

I argue that NullPointerExceptions or dynamic-cast exceptions in Java are a different class of problems.Memory is not corrupt in these cases. The VM keeps track of the memory (e.g. what class type this bit represents) and reports errors upon invalid requests. This might cause a program crash, but a program crash is a blessing (IMHO) as compared to a program that still runs and appears robust, but produces incorrect, yet non-suspicious, output.

"I agree safe memory is good, but I don't think GC is required for this, nor does GC completely achieve it. Totally safe memory comes with a performance cost (GC aside). "

Other ways for ensuring memory-safety are perhaps possible, but an "exclusive GC" has one obvious advantage over them: you have it today, it is proven to be working in many commercial applications.

"I wonder if its possible to get C-level performance with safely managed memory?"

For some simple cases this is already achieved in C++. Compare this Java code:

```
void fun()
{
  Vector<int> vec = new Vector<int>();
  vec.add(1);
  use(vec);
} // all the GC overhead
```

And this C++ code:

```
void fun()
{
  vector<int> vec;
  vec.push_back(1);
  use(vec);
} // automatic memory management!
```

But then, consider the following function:

```
int&amp;mp; access( vector<int> & vec, unsigned i )
{
  return vec[i]; // is i in range?
}
```

How would you check it. It is not possible to do it at compile-time, so you need to add some run-time overhead, which some will find inacceptable.

Reply

Jones et al. recently published an excellent book that goes into detail o of the subjects you are interested in. “The Garbage Collection Handbook http://gchandbook.org/ (http://gchandbook.org/)

Note:

1. Reference counting is a form of automatic garbage collection. See “A unified theory of garbage collection”. http://researcher.ibm.com/files/us-bacon/Bacon04Unified.pdf (http://researcher.ibm.com/files/us-bacon/Bacon04Unified.pdf)

2. You assume that scope-based reference counting collects immediately when, in fact, it keeps floating garbage around until the last reference falls out of scope and that can be a long time after it became unreachable.

3. There are cycle collection algorithms (e.g. trial deletion) for reference counting collectors.

4. Finalizers are very rare in real code.

5. RAII is standard in .NET and F# even has language support for it, e.g. do “use stream = System.IO.File.OpenRead filename” and the stream will be closed when that variable falls out of scope.

6. Early versions of both the JVM and CLR used reference counting. They switched to tracing collection precisely because it is much faster. There are experimental versions of the JVM using the latest forms of reference counting and they are still slower than today’s tracing collectors. That’s why we use tracing collectors!

7. Staccato is a parallel, concurrent and real-time garbage collector. http://researcher.ibm.com/files/us-bacon/McCloskey08Staccato.pdf (http://researcher.ibm.com/files/us-bacon/McCloskey08Staccato.pdf)

I collated some of the reasons why modern garbage collectors are fast here. http://flyingfrogblog.blogspot.co.uk/2012/05/how-can-garbage-collection-be-faster.html (http://flyingfrogblog.blogspot.co.uk/2012/05/how-can-garbage-collection-be-faster.html)

You may also be interested in my classification of the major families of GC algorithms. http://flyingfrogblog.blogspot.co.uk/2011/11/kinds-of-garbage-collection.html (http://flyingfrogblog.blogspot.co.uk/2011/11/kinds-of-garbage-collection.html)

> Oh my, I might actually have to buy the book just to see the full context of these points. I can reply briefly, without context however.
>
> 1. I find most people think of scanning garbage collectors when they hear the term. I won't disagree, but it is kind of a misnomer to call RAII a GC since there never is garbage laying around.
>
> 2. If an unreachable object has not been deleted the reference count system is simply broken. I have never seen this fail before. This point either makes me curious about the book, or lets me dismiss it as nonsense. I'm yet undecided.
>
> 3. Good to know, perhaps we can build a system with reference counting that deals with loops.
>
> 4. Because they are useless since there is no guarantee as to when/if they are called.
>
> 5. I consider the natural RAII of C++ to be superior to both "use" declarations and "defer" statements. Though both of those are better than nothing at all. Plus, in a scanning system the "defer" and "use" are actually dangerous since if somebody still has access to those objects and intends on using them, the higher level scope will be breaking them the moment it leaves scope.
>
> 6. Sounds like the same invalid argument again. I'm sure scanning is faster than ref-counting if you use it on every single object in the system and always inc/dec on every copy. Typically in C++ the number of objects using shared pointers pales in comparision to the total number of objects, and the number of times they are copied is much less than there actual use.
>
> 7. Continually scanning memory from another processor is a significant waste of resources. Your CPU is involved in an endless trafficking of data between caches for the sole purpose of doing GC. I also can't see this scaling to a system that has 12, 24, or more cores.

"If an unreachable object has not been deleted the reference count sys Reply simply broken"

That is a common misconception. Consider the program "{ MyObj foo(); f(foo); g() }". If foo uses scope-based reference counting then the count will be decremented to zero and foo will be reclaimed at the end of scope after the call to "g" even if "g" does not require foo. With a tracing collector, the compiler will overwrite the dead local reference to foo with something that is live, the garbage collector will not find anything referencing foo and can reclaim foo during the call to g. With tail call elimination, f's entire stack frame will have disappeared by the time g is running. This is why tracing collectors can and do collect earlier than scope-based reference counting.

“lets me dismiss it as nonsense”

A lot of people have researched garbage collection over the past 50 years. Dismissing their work because you assume it is nonsense when you haven’t even read up on the subject would be phenomenally retarded.

“I also can’t see this scaling to a system that has 12, 24, or more cores.”

Azul have been doing this with almost 1,000 cores for years now. Don’t speculate when you can just check the facts.

> This argument about scope-based reference counting is, as I suspected, not valid. Something has to determine this reference is no longer being used — and this can’t just be the GC. The compiler has to somehow indicate that the stack-based reference is no longer used, otherwise the GC will find it on the stack and assume it is still used. In the exact same fashion a non-GC’d language can determine this and release the object early.
>
> This also seems like a rather trivial gain. Given the rate at which you go through functions in a program, being able to clean up a few objects prior to the scope exiting doesn’t seen like a must-have feature.
>
> I must also note thatn in a non-GC language most objects tend to reside on the stack anyway where the actual memory management is nearly free. A fair comparision has to be able to consider these objects which simply don’t need to reside in the heap at all.
>
> Azul’s own site only claims from 1 to 10’s of cores, not 1000’s. I’m also not claiming it can’t be done at all, just that it will take a performance hit while scaling.

“Something has to determine this reference is no longer being used — can’t just be the GC.” Reply

That “something” is called liveness analysis. Its sole purpose is to remove values before the variable containing them goes out of scope.

“In the exact same fashion a non-GC’d language can determine this and release the object early.”

That would no longer be “scope-based reference counting” but it is possible. The problem is that you are then bumping reference counts every time a register takes a local copy of a reference or loses one (by being overwritten). I already demonstrated scope-based reference counting running 10x slower than a GC. Reference counting registers and stack slots would be even slower.

“This also seems like a rather trivial gain. Given the rate at which you go through functions in a program, being able to clean up a few objects prior to the scope exiting doesn’t seen like a must-have feature.”

The “g” function in my example could be your main loop. Reference counting just kept “foo” alive for the entire duration of the program when it was eligible for collection after initialization.

“I must also note thatn in a non-GC language most objects tend to reside on the stack anyway”

You can stack allocate with a GC.

“Azul’s own site only claims from 1 to 10's of cores, not 1000's.”

I said “almost 1,000” and you just have to Google to find lots of information that contradicts your statements:

See “The Azul Vega 3 system, featuring up to 864 processor cores and 768GB of memory in a single system…” http://www.azulsystems.com/products/vega/perf (http://www.azulsystems.com/products/vega/perf)

Here is a slideshow presentation about the scalability of their lock-free concurrent hash table implementation running on 768 cores. http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf (http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf)

“I’m also not claiming it can’t be done at all, just that it will take a performance hit while scaling.”

Then your statement is universally applicable to everything (nothing scales perfectly) and, therefore, conveys no information.

FWIW, there is a superb web page with lots of accurate information about memory management and garbage collection here: http://www.memorymanagement.org/ (http://www.memorymanagement.org/)

> I’m glad they have a fast concurrent hash, that’s good, but it isn’t a test of the GC. If somebody would write the same code in a non-GC system as a comparison perhaps we’d have a reference point.
>
> Their Vega appliance benchmarks don’t look impressive. The Trade6 benchmark shows a performance gain of about 40x compared to the DualCore, yet the Vega 7380 has over 800 cores more.
>
> A concurrent garbage collector overturns only one point in my article, the one about pauses and interruptions. By using an additional processor it looks like you can avoid the interruptions a GC would typically cause.
>
> I’m not convinced yet that the Azul solution scales in the way I’m thinking. I need to see a test showing how fast memory is actaully recovered and available for further allocation. The speed of recovery can obviously be compensated by simply having more memory available. Do you have benchmarks showing how GC behaves when an application approaches the memory limit? A comparison with a non-GC program?

“Nothing ever clears this cachce so over time it grows.” Reply

> + another 7 spelling mistakes. 2011 just wasn’t my year in spelling I guess. Thanks again.

Amen, brother. You know, I spend more time worried about object des Reply
in my exclusively GC-based language than I ever did with reference counting. GC has been a real productivity killer for me.

Re memory safety with C-like performance, one of the developers of L Reply
an interesting project. AIUI, it takes the output from clang and runs a couple of extra transformation passes, changing pointers into something like Go's "slice" objects, adding range checks to pointer arithmetic (which can, of course, be optimized away in most circumstances where the code is safe anyway) and disallowing unsafe pointer conversions. The resulting code is then run with a special allocator that uses virtual memory aliasing to ensure that no two allocations ever use virtual addresses that are in the same page (although they can, of course, reside on the same physical page). Then, when an allocation is freed, its page can be marked as not present, so the OS can trap any attempt to use a pointer to it again.

Obviously this restricts the kinds of program that can run, but it can apparently guarantee memory safety with little or no overhead in terms of processor use (although extra memory is required for page tables) for a reasonably large class of C programs.

As far as I know, the only language that supports prompt collection Reply
(deterministic garbage collection) and automatic memory management as well RAII using an algorithm that handles cyclic directed graphs of objects is Qore; if you are interested, there is a wiki page prompt collection approach here: https://github.com/qorelanguage/qore/wiki/Prompt-Collection (https://github.com/qorelanguage/qore/wiki/Prompt-Collection)

While this approach would be difficult to apply to a language like C++ without major changes to how memory management works and the use of pointers, there is no reason that this approach could not be used in other languages with automatic memory management.

(disclaimer: I'm the author of the language and the prompt collection approach)

> I like this approach. It's like the cycle detection of D, but I don't think that approach guarantees prompt deletion. I think your approach is what I would like to do in Leaf. It lets me defer the decision as well: I can just use reference counting for now and add in the cycle aspect later (I presume).
>
> Your example of repeated `finally` clauses and O(n) effort is a signfiicant reason to want the RAII pattern.

> I can just use reference counting for now and add in the cycle aspect later (I presume).

That's what I did with Qore (and also what MS did with VB 6, except "later" never came for VB regarding handling cyclic directed graphs of refcounted objects)

Thanks for your positive comments on the GC approach; let me know if you want any info on the algorithm – I'll have to check out Leaf as well…

Nothing prevents you to use RAII in java or c#. In fact standard librari Reply
encourage you to do so.

> Though you can follow an RAII pattern, the syntax of these languages doesn't not make it as easy as the Ctor/Dtor pairs in C++. The `using` or `finally` structure is a bit more bulky, though still usable and necessary at times. It's specifically the need to keep doing so makes me question some of the value of a scanning GC. If the need for this setup was completely eliminated I'd be far happier with scannign GC.

IMO scope-based resource management with destructors allowing the resource management to be entirely in the API and not depending on proper use by the user of the API is a natural evolution of automatic memory management found in high level languages. In fact, for me RAII only makes sense when the resource management responsibility is entirely in the API and not pushed on the programmer (for example with `using` or `finally`).

The Java interpreter knows exactly what objects and arrays it has allo Reply
can also figure out which local variables refer to which objects and arrays, and which objects and arrays refer to which other objects and arrays. Thanks for sharing.