

## Re: Destructor: not guaranteed to be called?

---

*Source:*

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.languages.vc/2006-01/msg00613.html>

---

- *From:* "Brandon Bray [MSFT]" <[branbray@xxxxxxxxxxxxxxxxxxxxxxxxxxxx](mailto:branbray@xxxxxxxxxxxxxxxxxxxxxxxxxxxx)>
  - *Date:* Tue, 31 Jan 2006 11:58:32 -0800
- 

It seems like the discussion has come to realize the difference between finalizers and destructors. The first is non-deterministic and loosely coupled, whereas the later is deterministic.

I do think there is a misunderstanding of the differences between destructors in managed code and destructors in native code. While there are differences, the discussion here hasn't highlighted any of them.

Arnaud Debaene wrote:

- > I agree : the point is that there is NO destructors in .NET!!! There are
- > finalizers, which are a different beast. CLI "destructors" have been
- > mapped to finalizers as best as MS could (generating code that implement
- > IDisposable, etc...), but this is by no way a native C++ destructor.

It's unfortunate that C# decided to use the tilda syntax for finalizers, and even more unfortunate that the old Managed C++ syntax did the same thing. However, the CLR makes no mention of destructors... so there's no real mapping to do. Destructors are a language level implementation, not a runtime issue.

- > The whole point of the IDisposable interface is to circumvent this
- > limitation of the GC, although it is still an inferior solution
- > compared to the native, synchronous C++ destructor, IMHO.

I'm curious how IDisposable presents an inferior solution. From my perspective as a language designer, I see IDisposable as the implementation detail for destructors in C++. Really, you don't have to know anything about IDisposable to use destructors in C++/CLI. To me, the biggest limitation imposed on destructors as a result of IDisposable is that all destructors are public and virtual. I actually that's a good thing, and it's a mistake that unmanaged C++ allows destructors to be anything else.

- > Agreed. There is NO destruction in .NET (nor in Java).

The premise of this statement is flawed. Destruction is a language level service, because only the language can determine when it is appropriate to deterministically cleanup objects. Why? Because the programmer needs to be involved – otherwise you deal with the infamous halting problem. The CLR is

## Re: Destructor: not guaranteed to be called?

a collection of services that can be supplied to a running program. As long as we're dealing with Turing Machines, the CLR will never be able to provide deterministic cleanup as a service.

So, that means deterministic cleanup must be moved to the language level. The best way to accomplish that and maintain a sense of cross-language functionality was to create a common API. That was `IDisposable`. From there, it's a matter of how the languages treat destruction semantics. C++/CLI does everything that unmanaged C++ does, including automatic creation of destructors when embedded types have destructors.

> No more memory leaks... The main reason for GC is to avoid raw memory  
> leaks, not to get a better model for logical destruction of objects.

While GC is primarily about memory leaks, I would argue it serves to do much more. C++ is inherently not type safe because it allows for things like use of an object after delete. GC in the context of a language like C++ is the only way to achieve type safety.

Also, if you are truly using Object Oriented Programming, objects will represent resources like files, network connections, UI, etc. This means that memory has a direct correlation to other resources, so GC has the potential to cleanup a lot more than just memory.

Lastly, deterministic cleanup is really bad at cleaning up in certain situations. A frequent example is shared resources that form a dependency cycle. The impact of reference counting is well understood, and all of the practices applied to unmanaged C++ frequently result in fragile programs. In situations like these, garbage collection is the best solution. The problem that usually results is programmers don't adapt to a different environment, and instead try to contort deterministic practices to a non-deterministic environment.

The short story... writing robust code still requires smart people thinking solutions all the way through.

---

Brandon Bray, Visual C++ Compiler <http://blogs.msdn.com/branbray/>  
Bugs? Suggestions? Feedback? <http://msdn.microsoft.com/productfeedback/>

- 
- *Follow-Ups:*
    - ◆ **Re: Destructor: not guaranteed to be called?**  
◇ From: Arnaud Debaene
  - *References:*
    - ◆ **Destructor: not guaranteed to be called?**

Re: Destructor: not gauranteed to be called?

◇ *From:* Peter Oliphant

◆ ***Re: Destructor: not gauranteed to be called?***

◇ *From:* Carl Daniel [VC++ MVP]

◆ ***Re: Destructor: not gauranteed to be called?***

◇ *From:* Peter Oliphant

◆ ***Re: Destructor: not gauranteed to be called?***

◇ *From:* Arnaud Debaene

- Prev by Date: ***Visual C++ bug report***
- Next by Date: ***Re: Visual C++ bug report***
- Previous by thread: ***Re: Destructor: not gauranteed to be called?***
- Next by thread: ***Re: Destructor: not gauranteed to be called?***
- Index(es):
  - ◆ ***Date***
  - ◆ ***Thread***