

Re: How come Ada isn't more popular?

Re: How come Ada isn't more popular?

Source: <http://coding.derkeiler.com/Archive/Ada/comp.lang.ada/2007-02/msg00211.html>

- *From:* Markus E Leypold
<development-2006-8ecbb5cc8aREMOVETHIS@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Mon, 05 Feb 2007 14:43:17 +0100
-

Maciej Sobczak <no.spam@xxxxxxxxxxxxx> writes:

Markus E Leypold wrote:

[I agree with what you say on historical perspective on language transitions and the probabilistic effects that languages have on newbies, so this part was cut.]

If you just do `f(s.c_str())` and `f_is_` properly behaved, that is, only reads from the pointer or does a `strdup()`, everything is fine, but, I note, not thread safe. I wouldn't exclude the possibility that the resulting race condition is hidden in a nice number of C++ programs out there.

If you have a race condition because of some thread is modifying a string object *while* some other thread is using it, then you have a heavy design problem. This is absolutely not related to interfacing with C.

Yes, I realize that. Still, providing a pointer to a the inner state of an object, that only stays valid until I touch the object for the next time is not only not thread safe (which I realized is not the problem) but it's not type safe: A error in programming leads to erroneous execution, i.e. reading and writing invalid memory. That is worse.

I think there was talk once about a thread safe string library, but at the moment I fail to see how that relates to the problem in question.

Your solution is thread safe, if the strings package is (which it wasn't in the past).

Re: How come Ada isn't more popular?

Strings package cannot make it any better, because the granularity of thread-safety results from the program logic, not from the package interface. String is too low-level (it's a general utility) to be thread-safe in any useful sense. That's why: a) it should not be thread-safe on its own, b) you still have a design problem.

Yes. I realize that. Don't know what made me write that :-).

Interestingly, Ada doesn't make it any better. Neither does Java. You always need to coordinate threads/tasks/whatever on some higher conceptual level than primitive string operations.

So forgive me. Let's ditch the thread safety aspect and instead: Giving pointers to internal state of objects violates (a) encapsulation (it fixes a specific implementation) and (b) is not type safe. I'm sure we can hang `c_str()` on account of this charge alone and can drop the thread-unsafety allegation.

[about closures]

You can have it by refcounting function frames (and preserving some determinism of destructors). GC is not needed for full closures, as far as I perceive it (with all my misconceptions behind ;-)).

Yes, one could do it like that. Ref-counting is rumoured to be inefficient

Which relates to cascading destructors, not to function frames.

My impression was it relates to both. Especially since both are interlocked: In a world with closure objects (from OO) and variables can refer to closures (function frames) and vice versa).

but if you don't have too many closure that might just work.

If you have too many closures, then well, you have too many closures. :-)

Re: How come Ada isn't more popular?

Yes :-). Only in a ref counted implementation even too many might not be enough.

We've been talking not only about performance, but also about readability and maintenance. ;-)

Of this thread? :-)

Furthermore I've been convinced that manual memory management hinders modularity.

Whereas I say that I don't care about manual memory management in my programs. You can have modularity without GC.

Certainly. But you can have more with GC.

In a strictly technical sense of the word, yes. But then there comes a question about possible loses in other areas, like program structure or clarity.

I think the absence of manual memory management code actually furthers clarity.

Being able to just drop things on the floor is a nice feature when considered in isolation, but not necessarily compatible with other objectives that must be met at the same time.

Which?

People who don't have GC often say that they can do anything with manual memory management.

And I say that this is misconception. I don't have/use GC and I don't

Re: How come Ada isn't more popular?

Re: How come Ada isn't more popular?

bother with *manual* memory management neither. That's the point. In Ada this point is spelled [Limited_]Controlled (it's a complete mess, but that's not the fault of the concept) and in C++ it's spelled automatic storage duration.

My impression was that Ada Controlled storage is actually quite a clean concept compared to C++ storage duration.

But both tie allocation to program scope, synchronous with a stack. I insist that is not always desirable: It rules out some architecture, especially those where OO abounds.

The problem with Controlled, BTW, is that it seems to interact with the rest of the language in such a way that GNAT didn't get it right even after ~10 years of development. Perhaps difficult w/o a formal semantics.

Today manual memory management is a low-level thingy that you don't have to care about, unless you *really* want to (and then it's really good that you can get it). And as I've already pointed out, in my regular programming manual memory management is a rarity.

On the other hand, most languages with GC get it wrong by relying *only* on GC, everywhere, whereas it is useful (if at all) only for memory.

I've heard that complaint repeatedly, but still do not understand it.

The problem is that few programs rely on only memory and in a typical case there are lots of resources that are not memory oriented and they have to be managed, somehow.

When GC is a shiny center of the language, those other kinds of resources suffer from not having appropriate support. In practical terms, you don't have manual management of memory, but you have *instead* manual management of *everything else* and the result is either code bloat or more bugs (or both, typically).

Now, now. Having GC doesn't preclude you from managing resources unrelated to memory in a manual fashion. Apart from that languages with GC often provide nice tricks to tie external resources to their

Re: How come Ada isn't more popular?

Re: How come Ada isn't more popular?

memory proxy and ditch them when the memory proxy is unreachable (i.e. the program definitely won't use the external resource any longer). Examples: IO channels (only sometimes useful), temporary files, files locks, shared memory allocations. Even if you manage resources manually, GC still limits the impact of leaks. And BTW – in functional languages you can do more against resource leaks, since you can "wrap" functions:

```
(with_file "output" (with_file "out_put" copy_data))
```

It's not always done, but a useful micro pattern.

Languages like Ada or C++ provide more general solution, which is conceptually not related to any kind of resource and can be therefore applied to every one.

Since you're solving a problem here, which I deny that it exists, I can't follow you here. But I notice, that

"Languages like C provide a more general solution (with regard to accessing memory), which is conceptually not related to any kind of fixed type system and can therefore implement any type and data model"

would become a valid argument if I agreed with you. It's the generality we are getting rid of during the evolution of programming languages. Assembler is the "most general" solution, but we are getting structured programming, typesystems and finally garbage collection.

The result is clean, short and uniform code, which is even immune to extensions in the implementation of any class. Think about adding a non-memory resource to a class that was up to now only memory oriented – if it requires any modification on the client side, like adding tons of finally blocks and calls to close/dispose/dismiss/etc. methods *everywhere*, then in such a language the term "encapsulation" is a joke.

Well, you think Ada here. In an FP I write (usually) something like:

```
with_lock "/var/foo/some.lck" (fun () -> do_something1 (); do_something2 param; ...).
```

The fact that Ada and C++ don't have curried functions and cannot construct unnamed functions or procedures is really limiting in this case and probably causal to your misconception that it would be necessary to add tons of exception handling at the client side.

Re: How come Ada isn't more popular?

Re: How come Ada isn't more popular?

And BTW: In Ada I would encapsulate the resource in a Controlled object (a resource proxy or handle) and get the same effect (tying it to a scope). Indeed I have already done so, to make a program which uses quite a number of locks, to remove locks when it terminated or crashes. Works nicely.

An ideal solution seems to be a mix of both (GC and automatic objects), but I think that the industry needs a few generations of failed attempts to get this mix right. We're not yet there.

OO is about encapsulation and polymorphism, these don't need references everywhere.

Yes, but -- you want to keep, say, a list of Shape(s). Those can be Triangle(s), Circle(s) etc, which are all derived from class Shape. How do you store this list? An array of Shape'Class is out of question because of the different allocation requirements for the descendants of Shape(s).

Why should I bother?

Note also that I didn't say that references/pointers should be dropped. I say that you don't need them everywhere. That's a difference.

OK, so you need them almost everywhere :-). I take your point.

I've decided, if I want to deliver any interesting functionality to the end user, my resources (developer time) are limited, I have to leave everything I can to automation (i.e. compilers, garbage collectors, even libraries), to be able to reach my lofty goals.

I also leave everything I can to automation. It's spelled [Limited_]Controlled in Ada and automatic storage duration in C++. I cannot imagine reaching my lofty goals otherwise. ;-)

Good. 'Controlled' buys you a lot in Ada, but there are 2 problems

Re: How come Ada isn't more popular?

(a) AFAIS (that is still my hypothesis, binding storage to scope is not always possible (esp. when doing GUIs and MVC and this like). I cannot prove but from what I experienced I rather convinced of it.

(b) AFAIR there are restrictions on `_where_` I can define controlled types. AFAIR that was a PITA.

The point is to know when to optimise, not to do it always.

I didn't even mention the word "optimization". I'm taling about structure.

OK. But how does a program become less structured by removing the manual memory management? The GC is not magically transforming the program into spaghetti code ...

Regards -- Markus

.