Put Up or Shut Up

John R. Ellis
Xerox Palo Alto Research Center
ellis@parc.xerox.com
September 9, 1993

It's time for researchers to put up or shut up.

For over twenty years, we've been touting the supposed benefits of garbage collection: It slices, it dices, it even cuts garden hose! But garbage collection still isn't available to the working programmer. If we really believe that GC improves programmer productivity and product quality, academic and industrial researchers need to get it into the hands of the average programmer. If we can't get commercial programmers using collection in the next several years, we should acknowledge failure and stop most GC research—collection may be a useful prototyping tool, but it just isn't viable for products. Why should the funding agencies and corporations subsidize another twenty years of garbage-collection research?

We could bicker endlessly over why garbage collection hasn't been accepted up till now: The languages have been inappropriate, we focused on hardware-based solutions, the algorithms are too inefficient. Or instead, we can focus on what we can do, starting now, to integrate GC into the real world.

In the past seven months, Dave Detlefs (DEC SRC) and I talked to C++ compiler and tool implementors at Microsoft, Sun, DEC, Purify, and Xerox. It's clear that they view GC as still too speculative, too inefficient, too unportable, too complicated for the average programmer. (Borland wouldn't even return my calls.) The existing language vendors aren't going to add GC to widely used programming environments unless we researchers work directly with them and address their immediate concerns. We must *show* them that GC can be practical for commercial programming.

Now we have an opportunity that won't last long. Rightly or wrongly, the commercial world is adopting “object-oriented programming” with a vengeance: Microsoft, Borland, Sun, Apple, Next and dozens of smaller vendors are pushing object-oriented paradigms in their products. In support of this trend, C++ is rapidly becoming the primary implementation language. This long-delayed embrace of abstraction, encapsulation, and reusable components is making storage management much harder—even “small” desktop applications consists of hundreds of thousands of lines of source and have dozens of man-years invested in them.

It's still early days. There will be several more years of fumbling and false starts, and nothing is set in concrete (yet). Despite the best efforts of the ANSI C++ committee, C++ is still evolving. We're being handed a perfect opportunity to persuade working programmers that GC should be an essential element in their standard C++ and C toolbox. Let's not blow it.

We don't need any more research in base technology—it's already adequate for many commercial uses. What we do need is a lot of effort invested in hard research-engineering issues of how to integrate GC into the current programming environments used by working programmers. The language vendors aren't going to do this work, and even if they wanted to, they shouldn't. They don't have the knowledge or the experience. It's the rightful responsibility of researchers to help the vendors make the hard design decisions.

But doesn't the work of Boehm and Bartlett on partially conservative collectors already provide complete prototypes for vendors to emulate? No. They address only some of the issues, leaving many others unanswered.

The issues fall into three major categories: the language interface, suitable algorithms, and code-generator safety. Dave Detlefs and I have written a paper addressing these in great detail[1], so I'll only summarize the questions:

---

[1] John R. Ellis and David L. Detlefs. "Safe, Efficient Garbage Collection for C++". Systems Research Center Report 102, Digital Equipment Corporation, 1993. Also available by anonymous ftp from `parcftp.xerox.com:/pub/ellis/gc`.

Language interface. How do programmers access garbage collection from within C++ and C? The syntax and semantics of the language interface *must* be standardized soon. Otherwise, application programmers can't write portable programs, and a product manager would be a fool to let his programmers use a non-portable, vendor-specific extension without a very strong reason.

Are all the objects in a program garbage collected? That sounds bad, since we probably can't integrate garbage collection seamlessly into existing libraries without source modification, and since commercial programmers will still want to optimize resource-critical parts of their programs. Allowing both collected and non-collected objects will let GC be introduced incrementally into large systems, allaying the fears of managers.

How do programmers specify which objects should be garbage collected? With type declarations, C++ placement syntax, or standardized class declarations? The solution for C++ will necessarily be different than for C. Type declarations require language changes; placement syntax violates the principle of encapsulation by forcing clients to decide how to allocate objects; standardized class declarations don't allow non-class types to be garbage collected.

Can non-collected objects reference collected objects and vice versa? Prohibiting such cross-heap references will make the incremental introduction of collection much harder. Since the prohibition will be hard to enforce by convention and could lead to nasty, subtle bugs, it would probably need to be enforced automatically within the type system, necessitating non-trivial language changes. On the other hand, cross-heap references puts bigger demands on collector implementations.

Can pointers to collected objects be passed to libraries written in other languages (C, Ada, Fortran, ...)? If not, that makes the incremental introduction of GC into large systems much harder, since there are existing reusable components written in those languages that commercial programmers aren't about to give up just for the privilege of using GC. But passing collected pointers to code written in other languages requires all these same language and implementation issues to be addressed for those languages as well. Doing so formally, with proper language specifications and working through the standards process, is a practical impossibility, though perhaps not a practical necessity.

Can programmers create pointers to the interiors of objects? That's a common, though perhaps not necessary idiom in C++; it's essential in C (for example, to implement by-reference parameter passing). Prohibiting interior pointers is almost certainly unacceptable to large numbers of programmers, regardless of their objective usefulness. On the other hand, handling arbitrary interior pointers imposes significant costs on collectors, and these costs haven't received much attention from researchers. (All current C++ implementations use interior pointers internally not only for evaluating addressing expressions into stack and register temporaries, but also for representing source pointers to multiply inherited objects; such pointers can be stored in the heap.)

What are the safety rules that programmers must follow to ensure correct operation of the collector with their program? It's clear that at least some restrictions must be placed on the *de facto* definitions of C++ and C as used by most programmers, and probably even the “formal” ANSI standard specifications need some restrictions. For example, can pointers be cast to integers and back again? Can pointers be stored in unions?

Do programmers have the option of asking the language implementation to automatically enforce the safety rules in selected sections of their code, as they do with the safe subsets of Cedar, Modula-2+, and Modula-3? When programmers violate the safety rules, garbage collectors can make a total hash of the heap, and the resulting mess can be much harder to debug than if they were using explicit storage management. It's clear that systems programmers occasionally need to use language features whose use safe use can't be automatically enforced, but a safe subset allows programmers to identify which portions of their program can be automatically checked and which can't.

How do programmers specify finalization (object clean-up)? The natural mechanism in C++ is to use destructors, but that raises religious issues that leaders in the C++ language community seems unwilling or unable to discuss substantively.

What are the semantics of finalization? In collected environments such as Lisp, Cedar, and Modula, clean-ups get applied asynchronously with respect to other threads of execution, but that's problematic for most C++ and C implementations that have only one thread of control. What guarantees of clean-up ordering are provided to the programmer? Are clean-ups guaranteed to be invoked at all?

Should there be multiple heaps, each with a different semantic or performance policy? Should the definitions of some or all of these multiple heaps be standardized? Research experience with multiple heaps has been very limited to date. Trying to introduce not-well-understood concepts into an industrial design is dangerous. Just getting plain-old-vanilla GC into the hands of programmers might be a big enough task with sufficient pay-off over many years.

Suitable collection algorithms. Besides the traditional performance problems of garbage collection (latency and space and time overhead), practical C++/C garbage collection introduces more headaches. Depending on the answers to the questions above, commercial viable collectors face additional requirements:

> *Interior pointers* addressing the middle of objects.
>
> *Cross-heap pointers* from collected to non-collected objects.
>
> *Unions* containing collected pointers (C++ and C unions are untagged, a requirement imposed by their de facto language definitions, if not their formal definitions).
>
> *Multi-lingual compatibility*, allowing collected pointers to be passed to code written in other languages.
>
> *Minimal changes to existing compilers and runtime environments*. Vendors have limited budgets and overwhelming skepticism, and the fewer changes required by GC, the better.

These requirements rule out many types of algorithms that researchers have investigated in the past. Partially conservative algorithms (those that scan stacks and registers conservatively and most of the heap precisely) are the most promising, especially the algorithms of Boehm and Bartlett. Virtual-memory synchronization may be the only practical way of introducing generational and incremental collection without modifying all the different language compilers used to produce a program. These algorithms are certainly feasible on Unix, OS/2, and Windows NT, but it's not clear how hard it is to implement them on traditional platforms such as Windows 3.1 and Macintosh.

In my visits with vendors, they were most worried about the performance of garbage collection, especially partially conservative algorithms. The only thing that will allay their fears is substantive measurements of real commercial programs using garbage collection, something that we as researchers don't yet have. We may be confident that our years of experience with academic environments such as Lisp, Cedar, and Modula will carry over to commercial C++ and C programs, but without real data, the vendors will continue to ignore garbage collection.

Measurements of small academic programs don't cut it. (Small is anything less than 100,00 lines that runs for less than a few minutes.) Microsoft couldn't care less about a 25,000 line VLSI routing program.

Getting commercial-scale benchmarks is very hard, not least because vendors treat their applications' source code as family jewels. When I was at DEC SRC, I initiated a project to measure the performance of some large DEC programs modified to use garbage collection, and I

got Ben Zorn a research grant that would allow him to use those programs for his own university research. We need more such industry-research cooperation.

I suspect the most challenging remaining algorithmic problems are not total execution time, but rather latency and space overhead. A 15% increase in execution time isn't an issue for many commercial applications, but long pauses or a factor-of-two increase in total memory are unacceptable. We don't have a good story to tell about memory overhead with large, long-running programs.

Code-generator safety. Partially conservative algorithms such as the Boehm and Bartlett collectors are designed to work with existing C++ and C compilers. Unfortunately, most good compilers can generate code that will cause incorrect operation of garbage collections. With many compilers, this only happens with optimization turned on, and rarely even then. To be safe, researchers using these collectors routinely turn off optimization or accept the rare possibility of a garbage collection bug (in practice, their programs are likely to have many more non-GC-related bugs). But commercial compilers need a more robust solution.

Implementing code-generator safety is straightforward and well understood, but retrofitting existing commercial optimizing compilers can be very difficult. Of all the changes Detlefs and I proposed in our talks to compiler vendors, code-generator safety was the only one that they thought would take any appreciable effort to implement.

Vendors were also concerned about the runtime costs of code-generator safety, since the straightforward techniques sometimes (but infrequently) generate extra instructions and use an extra register. Our limited research experience to date with code-generator safety indicates that in practice the overhead is insignificant, but we have no substantive measurements, especially on Intel architectures in which register pressure is a severe problem. The issue needs more attention from researchers before vendors will be satisfied.

Conclusion. Of all the ways researchers have studied for improving programmer productivity, getting garbage collection into the hands of working programmers could yield the biggest bang for a small buck. But we can't expect vendors to come up with acceptable language and implementation designs without the closer cooperation of researchers. It's unlikely that any vendor will do any of this design work at all. They won't want to invest their limited resources until researchers 1) gather more believable measurements of commercially realistic benchmarks, and 2) build consensus about the language design and implementation issues and construct prototypes that address the concerns of vendors and their customers.

The problems I've raised here may not seem sexy in comparison with the traditional longer-term problems that many GC researchers like to focus on. But vendors aren't going to the solve the problems, and unless we do, garbage collection will remain an esoteric domain that's rightly perceived to be of little short- or long-term value to industry. Funding agencies and corporate management will feel justified in reducing or even eliminating funding.

In the longer term, commercializing garbage collection will make GC research more interesting and more exciting. Researchers in other fields such as operating systems, file systems, and networking have large-scale commercial systems that they can measure, analyze, improve, and sometimes revolutionize. But most garbage-collection research is stale and boring, because we don't have that real-world connection.