

The Case for Garbage Collection in C++

D. Edelson* I. Pohl†

Baskin Center for Computer and Information Science
University of California
Santa Cruz, CA 95064

1 August 1990

Introduction

C++ represents an imperative based hybrid Object-Oriented Programming Language [ES90]. It integrates low-level and high-level programming language features in an attempt to benefit from both. The low-level features allow the programmer to write run-time efficient code. The high-level features support code-reuse through inheritance, and the development of large, maintainable codes through encapsulation. The base language C is a well designed system implementation language, and as such is easily compilable for most traditional architectures into highly efficient run-time object code. The OOP features are derived from SIMULA67 [B⁺73]. They allow the programmer to design modules that in the Platonic sense capture the characteristics of the abstraction to be programmed about [Poh91, Str90].

Garbage Collection is a technique for automatically identifying and deallocating inaccessible dynamically allocated memory. It is so useful, and perhaps, so difficult, that it has been a field of active research for over three decades. Garbage collection has been called a critical feature of an object-oriented programming environment [Mey88]. It is not provided in C++. A C++ programmer is responsible for explicitly deallocating unneeded blocks. For general, dynamic graph data structures, this is impossible without some form of graph traversal. In those cases the C++ programmer must either

* daniel@cis.ucsc.edu, (408) 479-7983

† pohl@cis.ucsc.edu, (408) 459-2263

implement garbage collection, or restrict the data structure. This makes C++ less convenient for manipulating such data structures than languages with automatic reclamation such as Eiffel and Modula-3 [Mey88, CDG⁺88]. We will argue that GC should be provided—at least as an option—and that it can be done in a way that is consistent with the goals of C++.

Garbage Collection and C++

Let us try to understand why it is not provided. Stroustrup designed C++ to be run-time efficient. He, as did Wirth in Pascal, wish the programmer to decide what features to use and to pay for only those features [Wir71]. A consequent requirement is that programming constructs be compilable into at most a few simple machine instructions. This is a sound design philosophy for imperative languages. A language that mandates GC, such as LISP or Smalltalk, violates this principle.

An archetypical C++ program consists of a set of abstractions encapsulated in `classes` communicating through their public interfaces. We expect a dynamic data structure to be implemented inside one of these modules. It will be part of a much larger program, possibly containing many such modules. The `class` containing the dynamic data structure can benefit from garbage collection. It will be easier to develop, and it may execute more efficiently. The rest of the program must not be penalized for coexisting with garbage collection. Equivalently, the module that uses garbage collection must not be penalized for being part of a large system. It must be encapsulated.

A GC scheme needs three properties to be appropriate for C++. All three follow directly from the design philosophy. A garbage collection scheme that adheres to these criteria is useful and appropriate for constructing large object-oriented systems in C++.

1. The GC scheme must be efficient,
2. Code that does not use collected objects must not be affected by the presense of a module that does, and
3. The efficiency of garbage collecting must depend solely on the complexity of the data structure.

Item 1 is mandatory for any “industrial strength” programming language. Item 2 is dictated by the requirement that the programmer pay for only those

features they use. Some ways of violating this criteria would be: tagged integers and pointers, additional type bits (e.g., structure tags) in non-collected objects, or additional complexity in the global `new` or `delete` operators. Any of these would impact code that does not use collected objects. The third property is necessary for a collected module to be properly encapsulated in an OO system. Just as a C++ program is part of an open and extensible system [Str90], so is a component of a program. A class that utilizes garbage collection efficiently in a small program should not be rendered inefficient by moving the class a large program, provided the complexity of the data structure remains unchanged. This is really the same as property number 2: without this the designer of the class is being penalized for code that s/he did not write, namely, the rest of the program. According to this criterion conservative collection and many other forms of mark-and-sweep GC are inappropriate for C++ [BW88].

Does C++ Need GC?

Efficient GC would be complicated in C++. Unlike pure LISP there is no one universal garbage collected structure. Many forms of garbage collection would conflict with the design goals of the language. No form of GC is unambiguously best.¹ Furthermore C++ provides `new` and `delete` as overloadable operators capable of manipulating free store in a general way. This together with the special member functions—constructors and destructors—provides the class designer with the means for allocating and reclaiming store dynamically. Also GC can create restrictions because of unanticipated effects of interactions. This can cause difficulty in writing code for distributed systems and real-time applications.

One point of view is that the burden for reclaiming dynamic storage is properly placed on the class designer, using the tools already provided in C++. This then allows programmer intelligence to be augmented by application-specific knowledge. Each case is treated potentially in a unique manner to maximize efficiency. Overly general schemes can lead to unthinking class designs. In general LISP has not been suitable for product code and many AI companies have had to redesign and ship their software products in C.

However, in our view this ignores major advantages. These advantages

¹Copying collection is widely considered superior for large systems, but even that is arguable [Boe90].

involve ease of use, rapid prototyping, correctness and yes, possibly run-time efficiency gains. Its advantages include:

- robustness* general GC can be checked out more thoroughly than single, class-by-class implementations;
- efficiency* the system can monitor effects that are not available to ordinary user: load, paging, idle-times, history, self-adaptation; furthermore, compaction can be used to improve locality and reduce paging;
- prototyping* storage reclamation is frequently a sophisticated aspect of object management. As with LISP and SMALLTALK having it available makes prototyping new classes easier.
- code-reuse* providing GC in the language saves the programmer effort and creativity.

Summary

C++ is hybrid OOPL [EP89] (other ref). A central design tenet is “do not pay for what you do not use.” A further design element is the flexibility of constructor/destructor storage management, and the ability to define customized free-store operators. Unfortunately, the ability to define a customized `delete` operator does not help the programmer know when to apply it. Unlike Smalltalk which mandates GC for all objects, and Lisp which has (at least originally) a simple view of free store [GR83, MAE⁺85], C++ can mix and match. It is our view that C++ can benefit from the Modula-3 approach of traced and untraced pointer types. This allows a programmer to use a keyword or a compiler pragma to allow efficient hybrid GC.

We have presented an argument supporting GC as an option in C++. We have also described some characteristics that a GC implementation needs to be consistent with the design goals of the language. The feasibility of our approach is demonstrated in [Ede90] with an implementation and analysis of a copying collector that satisfies these requirements. Our system makes it convenient (and possible) to program with generalized dynamic graph data structures in C++. In many cases it is more efficient than manual reclamation. A garbage collected data structure can be a well-behaved component of a large system because the encapsulation is complete. We expect this form of dynamic memory reclamation to make C++ a more convenient programming language in the very near future.

References

- [B⁺73] Graham M. Birtwistle et al. *SIMULA begin*. Auerbach, Lund, Sweden, 1973.
- [Boe90] Hans-Juergen Boehm, July 1990. Private communication.
- [BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. 18(9):807–820, September 1988.
- [CDG⁺88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical report, Digital Systems Research Center and Olivetti Research Center, Palo Alto, CA, 1988.
- [Ede90] Daniel Edelson. Dynamic storage reclamation in C++. Technical Report UCSC-CRL-90-19, Computer and Information Science, University of California at Santa Cruz, June 1990. M.S. Thesis.
- [EP89] Daniel R. Edelson and Ira Pohl. Solving C's shortcomings: Use C++. *Computer Languages*, 14(3), 1989.
- [ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, February 1990.
- [GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Publishing Company, Reading, MA, 1983.
- [MAE⁺85] J. McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and M. Levin. Lisp 1.5 programmers manual. In Ellis Horowitz, editor, *Programming Languages: A Grand Tour*. Computer Science Press, second edition edition, 1985.
- [Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
- [Poh91] Ira Pohl. *C++ for Pascal Programmers*. Benjamin/Cummings, 1991.
- [Str90] Bjarne Stroustrup. Usenet news article, July 1990. Article 11010@alice.UUCP in comp.lang.c++.

- [Wir71] N. Wirth. The programming language PASCAL. *Acta Informatica*, 1:35–63, 1971.