# Draft: Precompiling C++ for Garbage Collection

Daniel R. Edelson*

INRIA Project SOR
Rocquencourt, BP 105
78153 Le Chesnay Cedex
France
*edelson@sor.inria.fr*

26 March 1992

## Abstract

Our research is concerned with compiler-independent, efficient and convenient, garbage collection for C++. Most collectors proposed for C++ have either been implemented in a library, or in a compiler. As an intermediate step between those two, this paper proposes using precompilation techniques to augment a C++ source program with code to allow type-accurate garbage collection. In this way, the garbage collector can be more portable and distributable than a collector within a compiler, while simultaneously more convenient (i.e., more practical) than a type-accurate collector that is implemented entirely within a library.

The collector that is under development is based on precompiler-generated *smart pointers* as a replacement for raw pointers in the C++ program. The precompiler emits the smart pointer definitions, and the user is required to utilize them in place of raw pointers. These smart pointers supply functionality that allows the collector to locate all of the roots in the program. The precompiler also generates code that allows the collector to locate internal pointers within objects. This paper describes the architecture of the system, whose first implementation as a simple mark-and-sweep collector is underway. The paper also describes how the collector may eventually be extended with generations.

**Keywords**

C++, garbage collection, compilers, precompilers, smart pointers, mark-and-sweep, memory management

*Author's other affiliation: Computer and Information Science, University of California, Santa Cruz, CA 95064, USA, *daniel@cse.ucsc.edu*

## 1 Introduction

C++ is nearly alone among modern object-oriented programming languages in not providing garbage collection. The lack of GC decreases productivity and increases memory management errors. This situation persists principally because the common ways of implementing GC are deemed inappropriate for C++. In particular, tagged pointers are unacceptable because of the impact they have on the efficiency of integer arithmetic, and because the cost is not localized.

In spite of the difficulty, an enormous amount of work has been and continues to be done in attempting to provide garbage collection in C++. The proposals span the entire spectrum of techniques including (not exhaustively):

- concurrent atomic garbage collection implemented in the cfront C++ compiler [Det90],
- library-based object-management including reference counting and mark-and-sweep [Ken91],
- library-based mostly copying generational garbage collection from ambiguous roots [Bar89],
- library-based reference counting through *smart pointers*[1] [Mae92],
- library-based mark-and-sweep garbage collection using smart-pointers [Ede92a]
- compiler-based garbage collection using smart pointers [Gin91],
- library-based mark-and-sweep or copying collection using macros [Fer91], and
- library-based conservative generational garbage collection [BW88, DWH$^+$80].

[1]Smart pointers [Str91, Str87, Ede] are discussed later in this paper.

The vast number of proposals, without the widespread acceptance of any one, reflects how hard the problem is.

The goal of our research is to make type-accurate garbage collection available to the C++ community. This ideal imposes strong restrictions on the collector. Given the speed with which C++ compiler technology and the C++ language definition are advancing, any particular version of any compiler quickly becomes obsolete. In order to be carried along with the evolution of the state-of-the-art, and to be usable by anybody regardless of what compiler they choose, the collector must not be implemented in the compiler.

In the past, we have proposed implementing GC strictly in application-code. It would be something like "GC implemented in a library." The problem with this approach was that it required too much effort from the user. They had to first customize/instantiate the library (a substantial piece of work), and then follow its rules. Overall, this was a tedious and error prone process.

To solve our goal of compiler-independence, while keeping the associated complexity to the user to a minimum, we are now proposing *precompiling* C++ programs to augment them for garbage collection. In essence, the precompiler performs the necessary customization on the C++ program every time it gets compiled. The user still needs to cooperate with the collector, but the number of things to remember (and thus the likelihood of errors) is greatly reduced.

In this paper we discuss our collector architecture and related techniques for supplying garbage collection at the C++ source code level. Then, we describe the transformations that augment a C++ program with the necessary code for it to utilize garbage collection. The remainder of the paper is organized as follows: Section 2 discusses related work in garbage collection and memory management for C++. Section 3 provides an overview of the major techniques that we utilize to implement garbage collection. Section 4 describes the transformations that the proposed precompiler carrys-out to augmented a program for GC. Finally, section 5 concludes the paper.

## 2 Related Work

There is a significant body of related work, in the general field of GC, in C++ software tools, and specifically in collectors for C++.

### 2.1 Conservative GC

Conservative garbage collection is a technique in which the collector does not have access to type information so it assumes that anything that might be a pointer actually is a pointer [BDS91, BW88]. For example, upon examining a quantity that the program interprets as an integer (in a register, perhaps), but whose value is such that it also could be a pointer, the collector would assume the value to be a pointer. This is a useful technique for accomplishing garbage collection in programming languages that don't use tagged pointers, and in the absence of compiler support.

Boehm, Demers, et al. describe conservative, generational, parallel mark-and-sweep garbage collection [BDS91, BW88, DWH$^+$80] for languages such as C. Russo has adapted these techniques for use in an object-oriented operating system written in C++ [Rus91a, Rus91b]. Since they are fully conservative, during a collection these collectors must examine every word of the stack, of global data, and of every marked object. In addition, Boehm discusses compiler changes to preclude optimizations that would cause a conservative garbage collector to reclaim data that is actually accessible [Boe91].

Conservative collectors sometimes retain more garbage than type-accurate collectors because conservative collectors interpret non-pointer data as pointers. Often, the amount of retained garbage is small, and conservative collection succeeds quite well. Other times, conservative techniques are not satisfactory. For example, Wentworth has found that conservative garbage collection performs poorly in densely populated address spaces [Wen90, Wen88]. Russo, in using a conservative collector to reclaim dynamic storage used by an object-oriented operating system, has also found that inconveniently large amounts of garbage escape collection [Rus91a]. Lastly, we have tested conservative garbage collection with a CAD software tool called ITEM [Kar89, Ede92b, Ede92a]. This application creates large data structures that are strongly connected when they become garbage. A single false pointer into the data structure keeps the entire mass of data from being reclaimed. Thus, our brief efforts with conservative collection in this application proved unsuccessful.

As these examples illustrate, conservative collection is a very useful technique, but it is not a panacea. Since it has its bad cases, it is worthwhile to investigate type-accurate garbage collection.

### 2.2 Partially Conservative

Bartlett has written the *Mostly Copying Collector*, a generational garbage collector for Scheme and C++ that uses both conservative and copying techniques [Bar89, Bar88]. This collector divides the heap into logical pages, each of which has a *space-identifier*. During a collection an object can be promoted from from-space to to-space in one of two ways: it can be physically copied to a to-space page, or the space-identifier of its present page can be advanced.

Bartlett's collector conservatively scans the stack and global data seeking pointers. Any word the collector interprets as a pointer (a root) may in fact be either a pointer or some other quantity. Objects referenced by such roots must not be moved because, as the roots are not definitely known to be pointers, the roots can not be modified. Such objects are promoted by having the space identifiers of their pages advanced. Then, the root-referenced objects are (type-accurately) scanned with the help of information provided by the application programmer; the objects they reference are compactly copied to the new space. This collector works with non-polymorphic C++ data structures, and requires that the programmer make a few declarations to enable the collector to locate the internal pointers within collected objects.

Detlefs generalizes Bartlett's collector in two ways [Det90]. Bartlett's collector contains two restrictions:

1. Internal pointers must be located at the beginning of objects, and

2. heap-allocated objects may not contain "unsure" pointers.[2]

Detlefs' relaxes these by maintaining type-specific map information in a header in front of every object. During a collection the collector interprets the map information to locate internal pointers. The header can represent information about both sure pointers and unsure pointers. The collector treats sure pointers accurately and unsure pointers conservatively. Detlefs' collector is concurrent and is implemented in the *cfront* C++ compiler.

## 2.3 Type-Accurate Techniques

Kennedy describes a C++ type hierarchy called OATH that uses garbage collection [Ken91]. Its collector algorithm uses a combination of reference counting and mark-and-sweep. In OATH, objects are accessed exclusively through references called *accessors*. An accessor implements reference counting on its referent. Thus, the first reclamation algorithm available for OATH objects is reference counting. In addition, the reference counts are used to implement a three-phase mark-and-sweep algorithm that can collect cyclic data structures. The three-phase algorithm proceeds as follows. First, OATH scans the objects to eliminate from the reference counts all references between objects. After that, all objects with non-zero reference counts are root-referenced. The root-referenced objects serve as the roots for a standard mark-and-sweep collection, during which the reference counts are restored.

[2]An unsure pointer is a quantity that is statically typed to be either a pointer or a non-pointer. For example, in "`union { int i; node * p; }x;`" `x` is an unsure pointer.

In OATH, a method is invoked on an object by invoking an identically-named method on an accessor to the object. The accessor's method forwards the call through a private pointer to the object. This requires that an accessor implement all the same methods as the object that it references. Kennedy implements this using preprocessor macros so that the methods only need to be defined once. The macros cause both the OATH objects, and their accessors, to be defined with the given list of methods. While not overly verbose, the programming style that this utilizes is quite different from the standard C++ style. Additionally, current compiler technology renders long macros, such as those required for OATH, quite difficult to debug. A precompiler would have substantial benefits over a preprocessor for a system like OATH.

Goldberg describes tag-free garbage collection for polymorphic statically-typed languages using compile-time information [Gol91], building on work by Appel [App89]. Goldberg's compiler emits functions that know how to locate the pointers in all possible (necessary) activation records of the program. For example, if some function $\mathcal{F}$ contains two pointers as local variables, then another function would be emitted to mark from those pointers during a collection. The emitted function would be called once for every active invocation of $\mathcal{F}$, on the stack, upon a collection, to trace or copy the sub-datastructure reachable from each pointer. Upon a collection, the collector follows the chain of return addresses up the run-time stack. As each stack frame is visited, the correct garbage collection function is invoked. A function may have more than one garbage collection routine because different variables are live at different points in the function. Clearly, this collector is very tightly coupled to the compiler.

Yasugi and Yonezawa discuss user-level garbage collection for the concurrent object-oriented programming language ABCL/1 [YY91]. Their programming language is based on active objects, thus, the garbage collection requirements for this language are basically the same as for garbage collection of Actors [Dic, KWN90]. Their position paper describes a process very similar to the one proposed in this paper, namely, translating a source program into another source program that is augmented for GC. The programming paradigms for C++ and ABCL/1 [ANS91, Yon90] are quite different; each introduces its own problems that the collector needs to solve.

Ferreira discusses a C++ library that provides garbage collection for C++ programs [Fer91]. The library supplies both incremental mark-and-sweep and generational copy collection, and supports pointers to the interiors of objects. The programmer renders the program suitable for garbage collection be placing macro definitions at various places in the program.