Comparing Two Garbage Collectors for C++
Technical Report UCSC-CRL-93-20
For Electronic Distribution Only

Daniel R. Edelson

University of California
Santa Cruz, CA 95064
USA
daniel@cse.ucsc.edu

INRIA Project SOR
F-78153 Rocquencourt Cedex
France
edelson@sor.inria.fr

16 Jan 1992

Abstract

Our research is concerned with compiler-independent, tag-free garbage collection for the C++ programming language. This paper presents a mark-and-sweep collector, and explains how it ameliorates shortcomings of a previous copy collector. The new collector, like the old, uses C++'s facilities for creating abstract data types to define a *tracked reference* type, called *roots*, at the level of the application program. A programmer wishing to utilize the garbage collection service uses these roots in place of normal, raw pointers. We present a detailed study of the cost of using roots, as compared to both normal pointers and reference counted pointers, in terms of instruction counts. We examine the efficiency of a small C++ application using roots, reference counting, manual reclamation, and conservative collection. Coding the application to use garbage collection, and analyzing the resulting efficiency, helped us identify a number of memory leaks and inefficiencies in the original, manually reclaimed version. We find that for this program, garbage collection using roots is much more efficient than reference counting, though less efficient than manual reclamation. It is hard to directly compare our collector to the conservative collector because of the differing efficiencies of their respective memory allocators.

This paper is a longer, more thorough version of a paper that appears in POPL '92 under the title, *A mark-and-sweep collector for C++*.

1 Introduction

C++ is a modern, object-oriented, imperative programming language that is rapidly gaining acceptance in a wide variety of domains, both in industry and in academia [Str91]. As a hybrid programming language [Poh92], it combines the run-time efficiency of C [ISO90, ANS89, KR88, KP90] with object-oriented and abstractive mechanisms such as inheritance, overloading, parameterized types and limited run-time type checking. The language is quite successful at providing high-level features while retaining efficiency.

Programming in C++ is no simple task. The complexity of its semantics has been unfavorably compared to that of Ada. Furthermore, C++ is virtually alone among object-oriented programming languages in failing to provide some form of automatic dynamic storage reclamation, or garbage collection (GC). For example, Eiffel [Mey88], Smalltalk-80 [GR83], Self [CUL89], ML [Wik87], and Modula-3 [CDG+88] all provide GC.

The absence of GC from C++ results from various design goals of the language. A basic principle is localized cost (pay for what you use, when you use it). If support for GC slowed down *all* programs, even marginally, this would be unacceptable. Only those programs than can and choose to benefit from GC may be obliged to incur run-time overhead. Many garbage collectors lack this property. This criterion is normally applied at the level of the program, that is, either the program does or does not use garbage collection. However, it also applies at the level of the module. Some modules within a program may manipulate data structures for which garbage collection is desirable, while others do not. In this case, only the code in modules that use GC should be affected by the presense of GC.

There is already a moderately large body of existing C++ code. A collector that is object-code compatible with existing code, and that can be distributed in library form, will be most useful. In this respect we seek *loose coupling* between the collector and compiler [Det90].

Various garbage collectors have been proposed or can be used with C++ [EP91, Bar89, BDS91, Ken91, Det90]. A primary problem that all these collectors solve is locating pointers (roots) on the stack and in global data. We have previously presented a copy collector for C++ [EP91]. We have also briefly introduced a mark-and-sweep collector [Ede92]. This paper presents the architecture of and problems with the previous copy collector in the context of our goals. It also provides a detailed description of the latest version of the mark-and-sweep collector, and show how it remedies the shortcomings of the copy collector. We analyze the impact that the use of this collector has on an application program. We empirically compare the collector's efficiency to several other memory management alternatives.

The rest of this paper is organized as follows: Section 2 presents the garbage collection problem and provides motivation for solving it in C++. This section defines some terminology used in the remainder of the paper. Section 3 is provided to acquaint readers with various aspects of C++ that are important for understanding this work. In particular, our collectors make use of C++ abstract data types (ADTs) to supply their services at the level of the application. The second and subsequent subsections of this section will be review for readers who know C++, and may be skipped. Section 4 presents the previous copy collector and explains its problems. Section 5 describes the new mark-and-sweep collector. This section discusses ways that the new collector alleviates problems of the old one. Section 6 examines related work, and section 7 concludes the paper.

2 Garbage Collection

2.1 Motivation

Garbage collection is a programming language or environment service in which unreferenced dynamically allocated data is automatically deallocated. It is an intrinsic element of functional and logic programming languages that allocate the majority of their data dynamically. Many imperative programming languages also offer garbage collection, Eiffel [Mey88] and Modula-3 [CDG+88] being two examples. Garbage collection generally comes in one of two forms: reference counting and tracing [Knu73, Coh81]. Reference counting algorithms count the number of references to each object, and deallocate objects when their counts become zero. Pure reference counting algorithms cannot reclaim circular data structures. Tracing algorithms visit the data structure to determine which objects are accessible. Then, the inaccessible objects are deallocated. This paper concerns itself primarily with tracing algorithms, though the efficiency comparisons include reference counting as well.

The argument in support of garbage collection for imperative programming languages is that it increases both productivity and the reliability of the resulting software. The increase in productivity is because the programmer no longer needs to worry about deallocating dynamically allocated objects. This is critical for programs that manipulate (or *mutate*) generalized, dynamic graph data structures. In general, it is difficult to know whether there are outstanding references to an object. When garbage collection is not available, it is sometimes necessary to avoid having multiple pointers to dynamically allocated objects, simply because it becomes impossible to know when the object may be deallocated.

Garbage collection increases safety because it eliminates dangling references. These errors occur when a programmer deletes an object while a pointer to the object exists. With garbage collection, as long as a pointer exists, the object won't be deleted. Thus, dangling references are nominally impossible.

The main reasons for opposing the addition of GC to a programming language include the following:

1. "I rarely need GC and it slows down all programs, not just those that need it."

2. "Garbage collection isn't worth the overhead it imposes."

The first point illustrates how GC must be related to the C++ programming language. It is true that not all programs written in C and C++ need GC. It should be available for those that do, but must not impact those that do not. In particular, when the programmer chooses to manage dynamic data manually, then the impact of the garbage collector must be nil.

On the other hand, there are many programs that are ideal candidates for garbage collection clients. If garbage collection is available, the programmer may be able to either: 1) increase efficiency by allowing objects to be referenced by multiple pointers, or 2) increase productivity and simplicity by using automatic reclamation. In these programs, the presense of garbage collection can decrease development time and effort. If it permits sharing and shallow copying, then it can increase run-time efficiency as well. Finally, for general dynamic data structures, reclamation simply cannot be obtained without tracing. If garbage collection is unavailable, then the programmer is forced to implement it in his/her particular context.

Garbage collection should be made available to the C++ programming community. It is a valuable programming language feature for which there is no adequate substitute. However, the requirements on a implementation are significant. In particular, all GC costs must be localized. Also, it must be possible to avoid all overhead (and benefit) when the programmer so chooses.

2.2 Basic Algorithms

In this subsection we introduce the basic garbage collection algorithms. That done, we will later be able to show the difficulty of implementing them in C++.

The problem presupposes a *data structure* of dynamically allocated objects. The objects are represented as nodes in a directed graph. Objects may contain pointers; the pointers are represented as directed edges in the graph. We refer to those pointers as *internal pointers*. Other pointers are located on the stack, in the static area, and in the registers. These pointers are the only means the application has of accessing the data structure. These pointers are called the *roots*. Any object that can be reached by following a pointer sequence starting from a root is *reachable* or *accessible*. All other objects are *unreachable* or *inaccessible* or *garbage*. It is the garbage collector's task to identify the inaccessible objects. This organization is presented in figure 1.

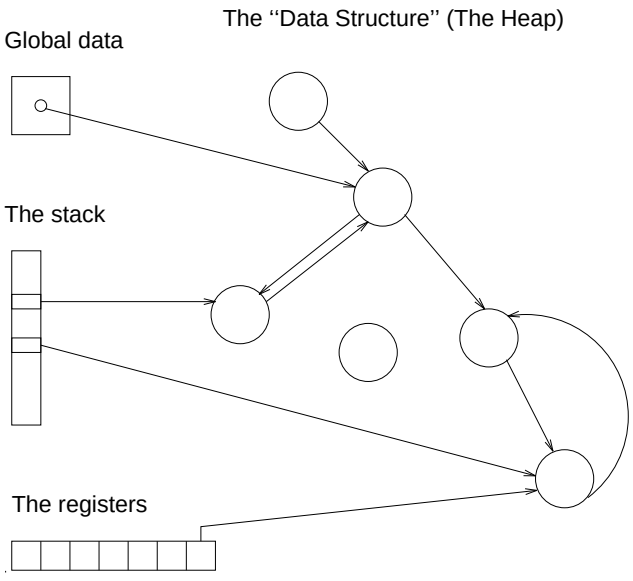

Figure 1: A representative data structure

The pointers in global data, on the stack, and in the registers collectively comprise the *roots*.

A program is typically modeled as a set of one or more application processes and one or more garbage collector processes. The application processes are called the *mutators*; the garbage collector processes are called the *collectors*. This terminology was introduced in [DLM+78] and has since become standard.

Garbage collection algorithms are typically based on one of two techniques: trace-and-sweep or copying.

2.2.1 Mark-and-sweep collection

A mark-and-sweep (or trace-and-sweep, the terms are equivalent) garbage collector iterates over all of the roots [Knu73]. From each root, it visits the subgraph reachable from the root. As each object is visited, a mark bit associated with the object is set. This is known as the *mark* or *trace* phase.

After the mark phase, the collector iterates over all of the allocated objects. For each object, if its mark bit is unset, then the object is deallocated. This second phase is known as the *sweep*. This process deallocates all unreachable objects.

Mark-and-sweep collection must be able to locate both the roots and the internal pointers. A form of mark-and-sweep known as *conservative* collection relaxes this requirement [BW88]. Basically, conservative collection treats as a pointer any properly aligned value whose value is such that it could possibly be a pointer.

2.2.2 Copying collection

Modern copy collectors are based on the work of Fenichel and Yochelson [FY69] and Minsky [Min63]. Incremental copy collectors are typically based on the Baker algorithm [Bak78]. Copy collectors allocate objects from one region and then copy all live objects into another region. These collectors *compact* the objects into the new region improving virtual memory performance. Mark-and-sweep collectors, by contrast, require a third pass to compact; conservative collection generally precludes compaction. Since copy collectors never deallocate individual objects, a very simple, fast storage allocator can be used.

Garbage collection may be invoked by the allocator when it runs out of available memory, or it may be explicitly invoked by the application. The first action of garbage collection is a *flip*, to cause subsequent allocations to be satisfied from a new memory space. Then, the collector locates all of the data structure's roots. From each root, the collector visits the reachable objects, copying every object that it encounters. The old version of every object contains a forwarding pointer whose value is initially NULL. When the collector visits an object, it examines the forwarding pointer to see if the object has already been copied. Whether the object was already copied, or if it is just now copied, the pointer that led to the object is modified to point at the object's new location. If the forwarding pointer

was previously NULL, meaning the object is just now being copied, then the forwarding pointer, too, is updated with the object's new address. The new versions of objects are identical to the old versions, except that in preparation for the next garbage collection, their forwarding pointers are initialized to NULL. Figures 2, 3 and 4 show a data structure being copied by a copy collector. In those figures, obsolescent pointers in from-space are omitted for clarity.

During the copy from each root, the data structure can be traversed breadth-first with a queue or depth-first with a stack. The to-space region can supply memory for the queue or stack. Objects are copied into to-space from one end of the region; the process of collecting compacts all the living objects. Since collection requires two full memory spaces, only half of the system's memory is usable by the mutator at any given time.

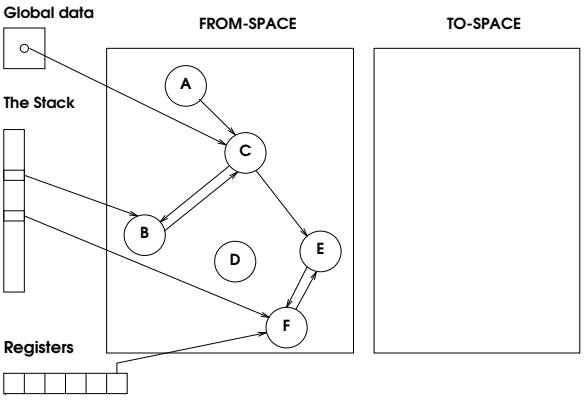

Figure 2: Before copy collection

2.3 Terminology

Any object that is reachable from some root by following a sequence of references is *live*. An allocated object that is not live is *garbage*. The job of the garbage collector is to locate and deallocate every garbage object.

A collector for a statically typed programming language is called *type-accurate* if every value that the collector interprets as a pointer is actually a pointer. The opposite of type-accurate is conservative [BW88]. Conservative collectors assume that any value that might be a pointer actually is a pointer. Partially conservative collectors such as [Bar89] and [Det90] are conservative in certain regions of memory and type-accurate in others.

Definitions of these and the other technical terms used in this paper are provided in appendix A.

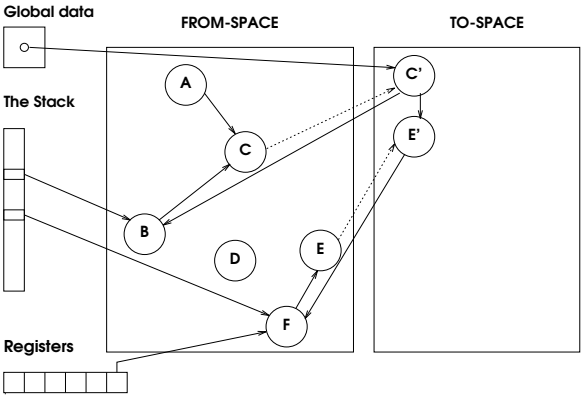

Figure 3: During collection: two objects have been compactly copied

2.4 Basic Techniques

There are certain problems that all garbage collection algorithms must solve. For example, both kinds of collectors (mark-and-sweep and copy) must locate the roots of the data structure. Mark-and-sweep collectors start from the roots in order to set the mark bit associated with every reachable object. Copy collectors start from the roots to copy the entire reachable data structure. In both cases, the roots need to be identified. This turns out to be a very hard problem to solve for a C++ garbage collector. Here are some potential ways of finding the roots:

Conservative scanning: This technique is used to provide GC in languages such as C and C++ in which minimal run-time type information is available. Conservative collection generally precludes copying collection because updating an integer

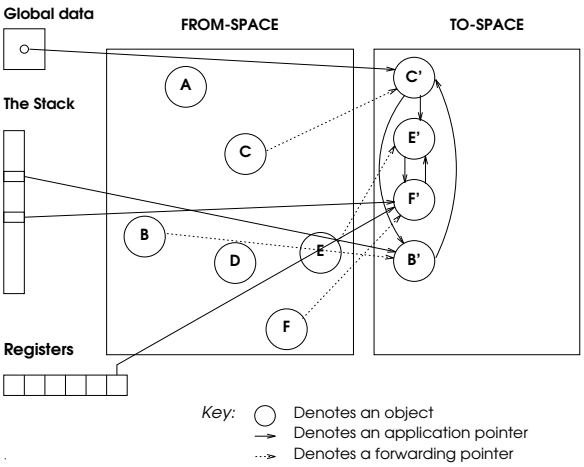

Figure 4: After collection: all objects have been copied

that was interpreted as a pointer would be incorrect. Some collectors such as [Bar89] are conservative in some regions of memory and type-accurate in others, allowing them to copy and compact a subset of the objects.

Tags: Collectors based on tags examine every word on the stack, in global data, and in the registers. Every word has a tag that indicates whether or not it is a pointer. Arithmetic efficiency is reduced for tagged integers; this violates the principle of localized cost. This solution is generally seen as undesirable for languages such as C and C++.

Stack-frame decoding: Garbage collectors based on stack-frame decoding require that the compiler provide map information describing the active roots in each stack frame. This **map** indicates what pointers are present as local variables or temporaries in that function invocation. The collector "unwinds" the stack, and interprets the map information that it finds in every activation record. Using this information, it marks the objects reachable from the roots present in that activation record. The map information may be maintained dynamically in the activation record [Wen88], or it may be generated statically, with the program counter used to locate the map corresponding to each activation record [Gol92, App89]. This solution permits source-level compatibility with existing code; it requires recompilation of the libraries.

Root registration: Collectors based on root registration record the addresses of the roots in auxiliary data structures, for example, the *protection stack* of [War87] and the *root lists* of [EP91]. Collectors based on this technique have the potential for object-level compatibility for existing code. However, there are disadvantages with root registration for C++ that are discussed later in this paper.

Root indirection: Collectors based on root indirection permit the application to manipulate only indirect pointers. Each indirect pointer references a direct pointer that is located in a *root table*. During garbage collection, the collector scans the root tables to find the roots. This method, too, has the potential for object-level compatibility between code that uses garbage collection and code that does not. Its disadvantages include the level of indirection and the cost of maintaining the tables. The *object table* of some Smalltalk-80 implementations [GR83, Ung86] constitutes root indirection, as do the *root tables* of [Ede92]. This method has advantages and disadvantages that are discussed at length in this paper.

3 The C++ Language

In order to understand our motivation and work, it helps to be familiar with C++. Therefore, this section provides an brief introduction to that language. The goal is to acquaint the reader with those features of C++ that permit us to write a garbage collector as a set of application-level abstract data types. That introduction develops a a small lexicon of C++-specific terminology that is used throughout the remainder of this paper. To understand this description it is not necessary to know C, but we will sometimes provide descriptions in the form of comparisons with C.

3.1 Overview and Motivation

C++ is based largely on the C programming language [ISO90, ANS89, KR88, KP90] from which it inherits run-time efficiency. C++ adds to C the notion of *classes*, a concept that it borrows from Simula-67 [DN66].

C is a good language for writing efficient, terse code. However, for a number of reasons, C is not a good programming language for writing *abstractions:*

- C data types are only minimally partitioned into interface and implementation. There is simply the definition of the data and the list of operations on the type. Therefore, implementation details of data structures are nearly always available. The minimal separation that exists is that function implementations don't need to be visible to client code, though most data definitions do.
- There is no such thing as inheritance in C. This makes it inconvenient for a programmer to take advantage of commonalities among related but distinct data types.
- There is no late (dynamic) function binding in C. This makes it less convenient to manipulate polymorphic data structures.
- There is a great deal of *predefined semantics* built into user-defined objects that the programmer cannot override. For example, it is reasonable to want to enfore *deep copy* assignment semantics on certain objects that contain dynamically allocated data. However, C does not provide a mechanism for overriding the default *shallow copy* assignment and initialization semantics.
- C has only primitive support for generic data structures; there is no language support for parameterized types.

In summary, C is not an object-oriented programming language. C++ retains C's ability to program close to the machine level [EP89, PE88] making it a good language for writing efficient code. However, it adds high-level abstraction and object-oriented mechanisms allowing it to ameliorate the aforementioned deficiencies of C.

We use C++'s abstraction mechanisms to provide a GC service. We use language features to define a *tracked-reference* abstract data type. By manipulating this abstract data type rather than built-in pointers, the programmer gives the garbage collector the information it needs to identify and reclaim the garbage. This adds garbage collection at the source level rather than at the implementation level. The benefits of this approach include:

nonintrusiveness Client code can take advantage of the service or not as the programmer chooses. Since the service is not in the compiler, code that does not use GC is completely unaffected.

correctness The collector is at the source code level. Therefore, aggressive compiler optimizations are unlikely to render the program's data incorrect. If that does happen, then the compiler is in error because it has transformed a valid C++ program into an invalid one.

portability The collector is highly portable and easy to disseminate.

modularity If we can keep the interface the same, it should be possible to provide different implementations of the garbage collector. This might, for example, allow the client to select either mark-and-sweep collection or copying collection depending on the characteristics of the application.

Of course, there are also disadvantages to working at the level of the source code. For one, the compiler has access to type information that the library does not. Additionally, there are numerous optimizations that can improve the efficiency of the garbage collector that are only possible at the machine level. By working in source code we make it much more difficult to implement these optimizations.

3.2 Classes

Classes are the mechanism whereby user-defined types are defined in C++. In this discussion we refer to an object whose type is "class X" (type "X" for short) as an *instance of X*. Similarly to a struct in C, or a record in Pascal [Wir71], a class describes the data that its instances contain. However, a class also can describe numerous other attributes of the type:

1. A class lists the methods (operations) that are provided for objects of the type.
2. A class lists *constructors* and a *destructor* that are used for initialization and de-initialization of instances.
3. A class may specify code that is used to copy its instances. Copy for assignment and copy for initialization are different operations; both are overloadable.
4. A class indicates the *overloaded operators* that permit its instances to be involved in expressions using standard operator notation.
5. A class definition indicates which data members and function operations are available to the general public (the interface), and which are reserved for use within the implementation of the class itself.
6. A class definition may be parameterized with one or more types or values. This gives C++ a facility for *generic* types.

3.2.1 Member functions

Consider the linked list shown in figure 5. The list is circular, allowing it to implement either a LIFO data structure or a FIFO one. It would be reasonable to use C definitions such as those shown in figure 6 to implement this data structure. These definitions would typically be stored in a header file. A client wishing to utilize the definitions would #include the header file.

The code that implements the list (figure 6) includes the following:

1. The type Value is defined as a generic pointer—generic in this case means *pointer to unknown*. This type is later used for values stored in the list. (This would not be necessary if C supported parameterized types.)
2. Types are defined for nodes and lists. A node contains a value and a link; a list contains a pointer, called head, to a node. (Figure 5 shows how head is used.) For notational convenience, we introduce the abbreviations Node and List for the types struct node and struct list.

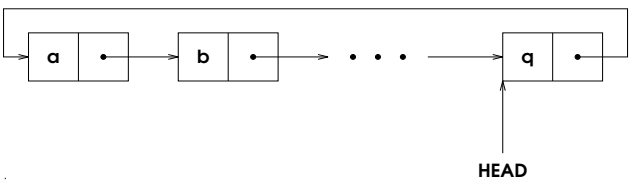

Figure 5: The singly linked list.

Lowercase letters represent the stored data.

```
/* file list.h */

/* A type to use for values in the list */
typedef void * Value;

/* A type for nodes in the linked list */
struct node {
    Value value;
    struct node * link;
};

/* The list type itself */
struct list {
    struct node * head;
};

/* Abbreviations for notational convenience */
typedef struct node Node;
typedef struct list List;

/* List of operations */
void  init(List *);
void  insert(List *, Value);
int   is_empty(List *);
Value remove(List *);
void  clear(List *);
```

Figure 6: C declarations for a linked list

3. Finally, there are declarations of the operations available on lists. For example, the declaration "Value remove(List *);" asserts that there exists a function named remove that takes a single argument of type *Pointer to* List and returns an object of type Value. Presumably, this function removes and returns the next object from the list.

Clients of this data structure need to be aware of implementation details, in particular, that the List type uses dynamic storage. For example, they must invoke init() to initialize lists before using them, and they should invoke clear() to free any remaining dynamic storage when done with a list. Failure to initialize a list will result in arbitrary values being dereferenced as pointers—an error that is both easy to make and easy to catch; failure to clear a list results in a nearly undetectable memory leak.

Copying these lists is dangerous; it results in two lists that share memory and can easily lead to dangling pointers. It is nearly always *pointers to list* that the programmer should be copying (or lists by reference), but there is no way in C to enfore this. On the other hand, it is quite reasonable to want *copy* to mean *deep copy*, that is, dynamically allocate a duplicate, identical data structure. In this case again the implementor of this data structure must depend on

```
/* file list.c */
#include <stdlib.h>
#include "list.h"

/* initialize list to empty */
void init(List * listp) {
    listp->head = NULL;
}

/* remove all elements */
void clear(List *listp) {
    while (!is_empty(listp)) remove(listp);
}

/* insert an element */
void insert(List * listp, Value value) {
    /* dynamically allocate and insert */
    ...
}

/* test for empty */
int  is_empty(List * listp) {
    return listp->head == NULL;
}

/* remove */
Value remove(List * listp) {
    /* remove and return an element */
    ...
}
```

Figure 7: A linked list implementation in Standard C

In the interests of brevity, some of the code has been omitted.

clients following a convention: deep copying can be invoked with code such as list2 = copy_list(list1). However, there is no way to make plain assignment using the assignment operator = mean deep copy.

To round out the example, the code in figure 7 implements the five functions that are listed, or *prototyped*, in figure 6. Typically, the type definitions and function prototypes are placed in a header file (e.g., list.h) and the function definitions are placed in a separately-compiled implementation file (e.g., list.c).

In summary, the defects of this C code are the following:

1. Genericity in the data structure is ad hoc. The list can only be used for objects that are the size of a pointer. For larger objects, a pointer to the object must be used. For non-dynamically allocated objects, this may require making a dynamic copy of the object and utilizing the resulting pointer.

2. Client code is perfectly capable of manipulating Lists at the implementation level, rather than through the documented interface.

3. Clients must be responsible for initializing and de-initializing instances of List. There is no way for the supplier of the type to guarantee initialization and reclamation.

4. Copying of Lists and initialization of Lists by Lists are undesirable operations. The language does not provide any means of either forbidding them or changing their semantics.

3.2.2 C++ code

C++ code that implements the same data structure is shown in figure 8. C++ supports parameterized types called templates that are used for generic data structures of this kind. However, the syntax for defining templates is somewhat complicated. Therefore, we present a C++ example that uses the same kind of ad hoc genericity as is used in the C example. A C++ implementation of the same data structure that uses templates to be generic is available from by email request from the author.

The core of the C++ code is the definition of class List. This type definition defines instances of List to consist of a single data member, head, and a suite of operations. The operations include the usual list operations: insert(), remove(), is_empty(), etc. However, in addition, the operations include *constructors*, a *destructor*, and an *overloaded assignment operator*.

Within the definition of List there are two member functions named List; they differ in their argument lists (and implementations). A function whose name is the same as the name of its class is a *constructor*. The function "List()" specifies the code that initializes Lists for which no initializer is explicitly supplied. The body of this function sets the head data member to the NULL pointer. This guarantees that uninitialized Lists are properly initialized to the empty list.

The constructor List(const List &) is called a *copy constructor*. It specifies how to copy a List. This function is called for any of three reasons:

1. to initialize a by-value function parameter with the actual argument,

2. to initialize the temporary containing a by-value function return value, and

3. to explicitly initialize one List with another.

This function guarantees that making a duplicate of a list makes a 'deep copy'.

The member function whose name is ~*class-name*, ~List() in this case, is called the destructor. It is how the programmer specifies de-initialization code for

```
#include <stdlib.h>

typedef void * Value;

struct Node {
    Value  value;
    Node * link;
};

class List {
public:
   void  clear() { while (!isempty()) remove(); }
   int   is_empty() { return head == NULL; }
   void  insert(Value value)  { /* omitted */ }
   Value remove()             { /* omitted */ }
   const List & operator=(List &);

    List()       { head = NULL; }
    List(List &) { /* 'deep' copy */ }
    ~List() { clear(); }
private:
    Node * head;
};

const List & List::operator=(List & other) {
 // 'deep copy' assignment: 'this' of 'other'
}
```

Figure 8: A linked list in C++

This example uses ad hoc genericity in order to facilitate comparison with the equivalent C code.

Lists. In this case, the destructor frees any dynamically allocated storage in use by the object.

The class definition is divided into two sections by public: and private:. These partition the class definition into the interface and the implementation. The data and function members that are in the public: part may be accessed by any code in the program. However, the data and function members that are in the private: part are reserved for the implementation of the ADT; they may only be accessed within the definitions of the class'es member functions. In the case of this ADT, the instance data head is private and all of the operations are public.

The main differences between the C implementation and the C++ one are the following:

1. The C type definition defines only the structure of the data; the C++ type definition defines the structure of the data and specifies the user-defined operations. (Certain other operations, such as *address of*, are available by default unless specifically forbidden in the class definition.)

2. The C++ implementation specifies user-defined code (semantics) for copying during initialization and assignment.

3. The C++ implementation provides guaranteed initialization and de-initialization of Lists though constructors and a destructor.

4. The C++ type consists of private data and public operations, whereas the C code has no concept of *private* within a structure definition.

Constructors, destructors, and overloaded operators provide excellent support for writers of abstract data types.

3.3 Inheritance and Dynamic Binding

The final C++ feature that needs to be presented is inheritance with dynamic binding. A class may be defined with one or more other classes as *base classes*. The syntax is the following:

```
class base {
    /* members of the base class */
};

class derived : public base {
    /* new members of the derived class */
};
```

The keyword public in a derived class definition specifies *public derivation*. This is the most commonly used form of inheritance in C++. It should be interpreted as meaning that derived is a subtype of base. This means that an instance of derived may be used anywhere an instance of base is expected. Instances of the derived class have all the function and data members explicitly listed in the definition of the class, *in addition to* all of the members of the base class.

Finally, to accompany subtyping and to really support polymorphism, some form of dynamic binding is required. In C++ this is based on the *virtual function*. For example, consider figure 9.

(In the remainder of this section, the C and C++ notation "type *" (read *pointer to* type) will be used as an abbreviation for the written words "*pointer to* type".)

The function whos_there() takes a single argument of type base *. However, the type derived is a subtype of base. A pointer to a subtype may be supplied anywhere a pointer to the base type is expected, therefore, a pointer to an instance of derived may be passed to whos_there() in place of a pointer to an instance of base.

Within the function whos_there(), the pointer parameter is used to invoke the id() method on the referenced object. Since, in all cases, the pointer's compile-time type is base *, it seems reasonable to expect the string "base" to be printed always. In fact, the function id() is declared to be virtual. Therefore, whenever id() is invoked through a pointer, the *dynamic* type of the pointer is used to bind the call. In this example, the first call to whos_there() passes a pointer to a base object. The dynamic type of that pointer is indeed base *, and so the string "base" is produced. The second call to whos_there() passes the address of an instance of derived, a derived * in other words. This pointer is implicitly converted to a base *, however, its dynamic type remains derived *. For this reason, the second invocation of id() from whos_there() causes the string "derived" to be printed.

```
#include <iostream.h>

class base {
    virtual void id() { cout << "base\n" }
};

class derived : public base {
    void id() { cout << "derived\n" }
};

void whos_there(base * basep)
{
    basep->id();
}

int main(void)
{
    base * bp = new base;
    derived * dp = new derived;

    whos_there(bp);
    whos_there(dp); // treat derived like base

    return 0;
}

// The output is
base
derived
```

Figure 9: An example of polymorphism in C++

4 A Copy Collector

We have used root registration to implement a copy collector [EP91]. It implements the basic copy collector algorithm as described in [FY69].

4.1 Locating Roots

Since this is a copy collector it moves objects and must be able to *modify* roots. This contrasts with a mark-

and-sweep collector in which is it sufficient to have a *copy* of every root. This collector is based on root registration: every time a root comes into existence, its address is recorded in an auxiliary data structure. When the root is destroyed, the address is removed. The garbage collector is able to locate the heads of the auxiliary data structures, therefore it is able to locate all the roots.

Many roots are created and destroyed in LIFO order. In particular, for roots created on the system stack, it is natural to track their address with an auxiliary stack. We use a set of *root stack*s to track such roots. There is one root stack for every static type of root. For example, if the program uses base $*$ and derived $*$ pointers, then the program requires one stack for each. These auxiliary stacks were implemented two ways: the array implementation of root-stacks is shown in figure 10 and the linked list implementation is shown in figure 11. The array implementation utilizes a separate array that contains pointers to the roots. The linked list implementation threads the stack between the roots—every root becomes two words, its value and its link.

Not all pointers can be tracked with a stack. In particular, we do not require that all dynamically allocated objects be allocated from our memory allocator. Thus, there may be dynamically allocated roots whose lifetime is not LIFO with respect to all the other roots. We track such roots with a general doubly linked list. A doubly linked list supports deletion of its elements in arbitrary order. This means that roots tracked with the doubly linked list can be destroyed in arbitrary order. The root stack, on the other hand, would immediately become inconsistent if an attempt was made to remove any element other than the last to be inserted. Figure 12 shows the doubly-linked list of root addresses.

For simplicity, the diagrams in figures 10, 11 and 12 each present only one of their respective data structures. In reality there might be multiple such data structures. In the case of multiple lists or stacks, there is a *meta-list* of the lists (or stacks).

4.2 Language Interface

The root data structures presented in the last subsection are implemented at the C++ source code level using classes. To utilize this garbage collection service, the client code must replace all uses of *raw* pointers with these *smart* pointers. Assume the name of the garbage-collected class is Foo. Then, the definition of the smart pointers for class foo looks like the following. The linked-list implementation of stackable pointers is presented.

```
template<class Foo>
class Root<Foo> { /* Type for roots to Foo */
```

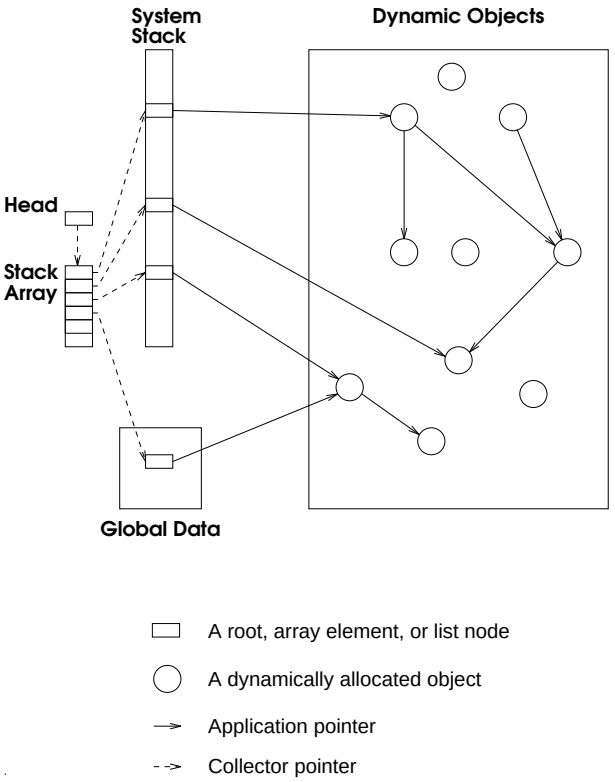

Figure 10: The root stack: Array implementation

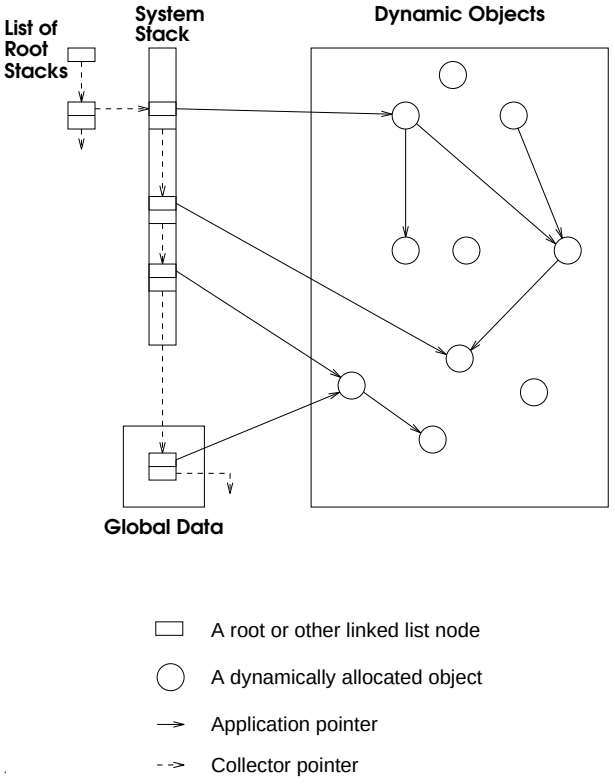

Figure 11: The root stack: Linked implementation

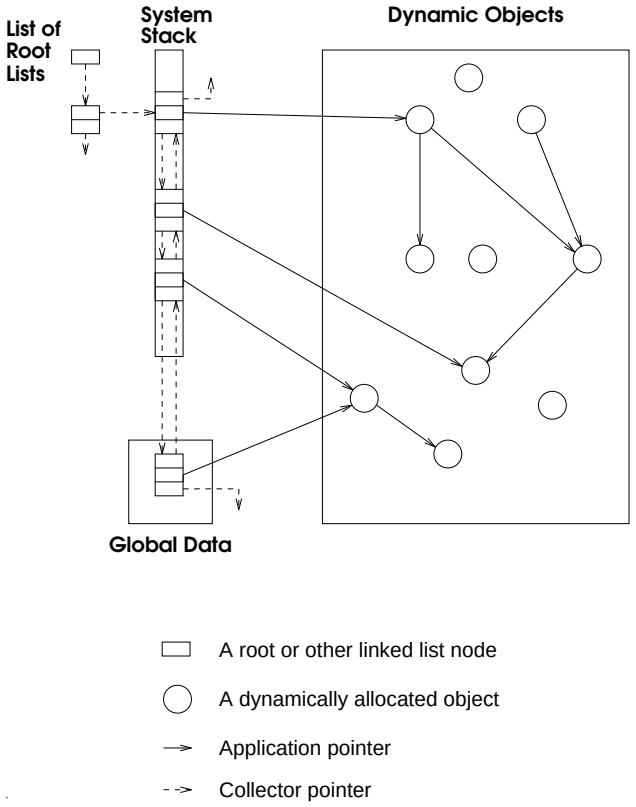

Figure 12: The doubly-linked root list

```
    // the list head
    static root_list_head<Foo> head;
    Foo * ptr;
    Root<Foo> * link;
public:
    Foo * &   get_ptr()          { return ptr;  }
    Foo   &   operator*()        { return *ptr; }
    Foo   *   operator->()       { return ptr;  }
    void      operator=(Foo * p) { ptr = p;     }
              operator Foo * ()  { return ptr;  }

    Root<Foo>(Foo * p) : ptr(p),link(head.list)
        { head.list = this; }
    Root<Foo>() : ptr(NULL), link(head.list)
        { head.list = this; }
    ~Root<Foo>() { head.list = link; }
};
```

4.3 Overview and Efficiency

This technique allows the garbage collector to locate all the roots in the system. Whenever possible, roots will be *stackable*, and will be referenced by the root stack. Some roots are not stackable, and they are tracked with the doubly linked list. The doubly linked root list could be used for all roots, however, stack insertion is considerably more efficient that doubly linked list insertion. Therefore, the root stack is present as an optimization.

Table 1: Copy collector root efficiency

Operation	Time
Create/init/destroy 2,000,000 pointers	$11.8s$
Create/init/destroy 2,000,000 roots	$14.2s$
Create/init/destroy 2,000,000 droots	$23.1s$
Function call overhead	$11.0s$
Time per pointer	$0.4\mu s$
Time per root	$1.6\mu s$
Time per droot	$6.1\mu s$

Utilizing a stackable root requires a linked list operation for creation and another for destruction. (In the case of the array implementation of stackable roots, the equivalent of a linked list operation is still required to adjust the bounds of the array.) Creation of the stackable root requires code to: 1) initialize the pointer, 2) initialize the link value, and 3) adjust the list head. Destruction of the stackable root requires a single linked list operation to restore the list head. By contrast, utilizing a raw pointer requires only a single operation: initialization of the pointer. Thus, we expect stackable roots to be about four times as expensive as raw pointers.

Utilizing a doubly-linked root requires two linked list operations each for initialization and for destruction. This entails roughly four times the number of memory references as for a stackable root. Therefore, on a RISC architecture, we expect a doubly-linked root to be about four times as expensive as a stackable root.

The figures in table 1 show the measured performance of root allocation and destruction.

As predicted, creating roots is on the order of four times more expensive than creating pointers. It requires three machine instructions per root that are not required for a simple pointer.

4.4 Problems with the Copy Collector

After we had implemented the copy collector, we found three problems with it.

The correctness of stackable roots, i.e., consistency of the data structure, depends on the fact that root destructors and constructors are invoked in proper, LIFO order. Since stackable roots are used for roots that are allocated on the system stack, we expected this to be a reasonable assumption. In fact, it turned out that certain compilers would not invoke the destructors in the expected LIFO order.

Consider figure 13. This figure is a diagram of a system with two roots on the stack. These two roots

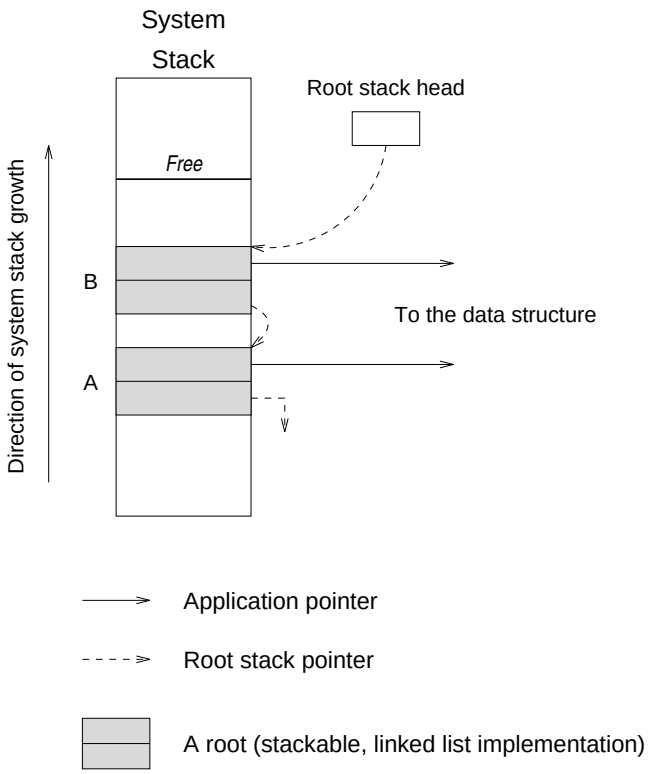

Figure 13: Two roots (smart pointers) on the stack

have been labeled A and B in the figure. The root labeled B is closer to the top of the stack, therefore it was created more recently than A. Since they reside on a stack, B will necessarily be deallocated before or with A. However, a compiler could contrive to invoke the destructor for A before invoking the destructor for B. For consistency of the linked list stack, it is mandatory that B be destroyed (have its destructor invoked) before A. If not, the stack operation will render the stack inconsistent. We found that under many circumstances A could be destroyed before B. In fact, the language does not require that the objects be destroyed in LIFO order, even though they are allocated on the stack. For this reason, it became necessary to use doubly-linked roots in place of stackable roots in many cases. This reduces the efficiency of the system. It is worth noting that an implementation in the compiler would be able to ensure LIFO order in many cases, and thus would be able to use stackable roots.

Another problem involves the this pointers of member functions. In C++, whenever a method (member function) is invoked on an object, a pointer to the object is passed to the method on the stack. This pointer is called the this pointer. Through the this pointer the method can access the object's instance data. Figure 14 shows a representative member function invocation; figure 15 shows the stack layout for the call.

```
class example {
public:
    int foo(int a, char c) { }
};

void func(void)
{
    // assume obj is at address 0x1000
    example obj;

    obj.foo(7,'x'); // invoke foo() on obj
}
```

Figure 14: A call of a member function

This code defines a class example and an instance named obj. The member function foo is invoked on obj and passed two arguments, the integer 7 and the character 'x'.

These pointers are in fact roots. Recall that this is a copy collector. As such, during a collection it changes the addresses of all the dynamically allocated objects. At the time of a collection, any active member function invocations will have this pointers on the stack. As part of collection, those pointers must be updated. Failure to do so leads directly to dangling references and an incorrect program.

If the collector can identify the this pointers, then it can update them. It can find them it if has their addresses. Thus, the solution to this problem is, upon every member function invocation, store the address of the this pointer in a data structure analogous to the root stack. However, the C++ language definition [ANS93, Str91] forbids taking the address of this pointers. Thus, the collector can only be implemented in a customized compiler; it cannot be implemented and distributed in a library, as is our goal.

The copy collector requires the *addresses* of the this pointers. As shall be seen later, the mark-and-sweep collector, because it does not move objects and alter pointers, requires only the values of the this pointers, not their addresses.

The final problem with the copy collector involves the use of two kinds of roots: stackable and doubly-linked. A stackable root requires two words, one for the pointer and one for the link. A doubly-linked root requires three words, one for the pointer and two for links. Suppose an object contains a root as instance data. What kind of root is required? The answer depends on how the object is allocated. To be consistent and safe, doubly-linked roots must be used.

It turns out that, with an implementation in a library, stackable roots can only be used for variables

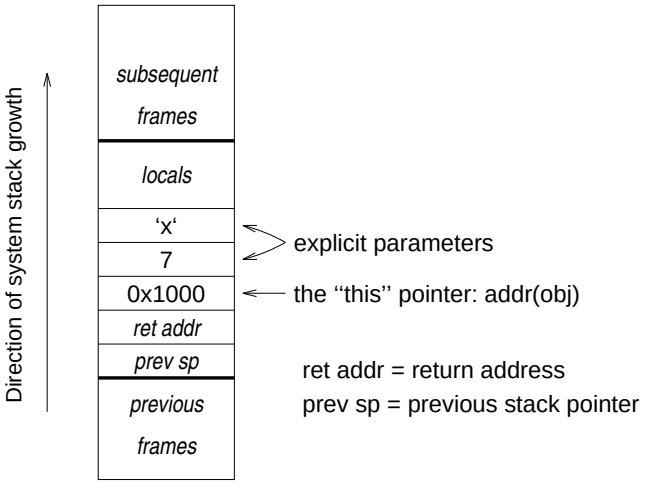

Figure 15: Stack layout for the call to obj.foo()

The two explicit parameters have been pushed onto the stack for the call to foo(). In addition, before the two explicit parameters, is the implicit this pointer to the object on which the operation has been invoked.

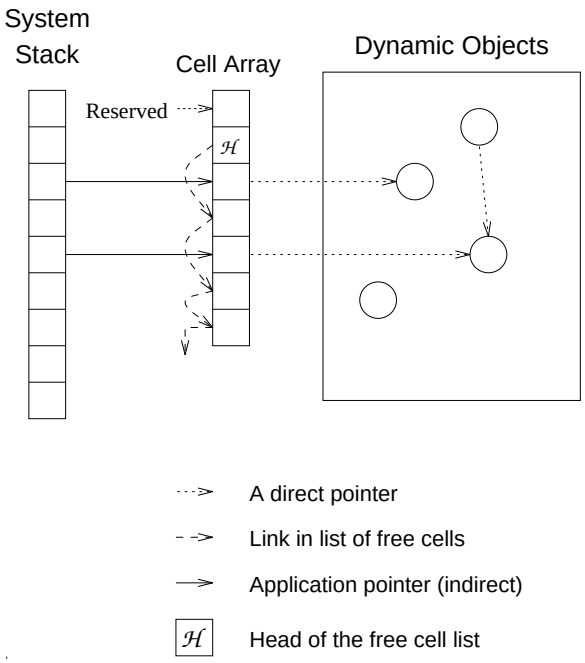

Figure 16: Root indirection

The mutator's roots are indirect through the root table. The root table contains *active* cells and free cells. The free cells are linked into a free list. The first cell of each table is reserved for making the list of tables.

that are local to a function. They cannot be used for function parameters, function return values, and temporaries, as we had expected.

5 A Mark-and-Sweep Collector

The problems with the copy collector have now been presented. To provide garbage collection, while avoiding those problems, we developed a mark-and-sweep collector. It consists of a memory allocator, parameterized type definitions, and parameterized functions. It is implemented outside of the compiler.

5.1 Indirect Root Tables

This collector is based on root indirection. Every root that the application manipulates is indirect through a *root table*. Each root table is an array of cells. A cell that is in use, and therefore that contains a direct pointer, is called *active*; a cell that is free is called *inactive*. The inactive cells are linked into a linked-list. Whenever a cell is required, one is taken from this free list. When no free cell is available, a new root table is allocated. The root tables are linked into a linked list. Figure 16 illustrates a single table. Figure 17 illustrates the list of tables.

When allocating a new cell, it is necessary to be sure that one is available. Normally, this would require a test and a conditional branch. Upon obtaining the pointer to the next free cell, the root table management code would check to be sure that the pointer is not NULL. However, the first thing that is ever done with the new cell is to read it's value to obtain the next link in the free list of cells. We can use that read operation to eliminate the test and conditional branch.

There is always one root table that was the last to be allocated. This root table is special, in that its inactive cells are *sequentially* linked into a free list. This must be true, because when the table is allocated, all of its cells are free, and they are all linked into a free list. We *read protect* the last page of this table [AL91]. During program execution, when we attempt to load the link stored in the first cell on the read protected page, the program receives and handles a signal. That signal tells the system to allocate and link in a new root table. A new diagram of a root table is presented in figure 18. The shaded area shows the read-protected region.

The roots themselves are defined as a parameterized C++ class. They are used in the following way. For example, a recursive depth first search, searching for a particular node in a graph might be coded similarly

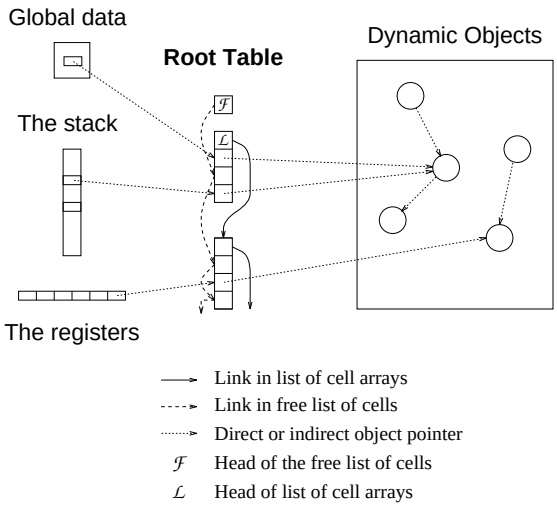

Figure 17: The list of root tables

The root tables are themselves linked into a list. The first word of each table contains its link.

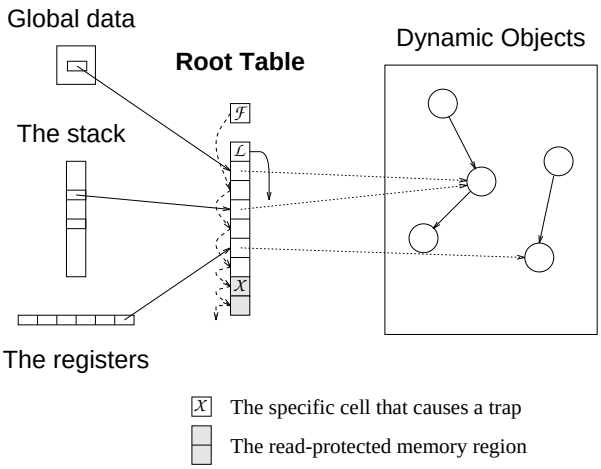

Figure 18: A root table's protected page

The most recently allocated root table has its last full page read protected. When the protection violation occurs, a new table is allocated and linked to the others.

to the code in figure 19.

Using roots, there would be only a single difference. The parameter to the function, instead of being a raw pointer, would be a root. This code is presented in figure 20.

By declaring them as Root<node> rather than as node *, the programmer causes the parameter to be indirect through a root table. This allows the garbage collector to find it and mark the nodes that it references, should a garbage collection take place.

There is an important optimization possible at this point. This is a noncompacting mark-and-sweep collector, not a copy collector, therefore, objects do not move. As long as there is a single root to a node in the graph, there may also be arbitrarily many raw pointers to the same node. The presense of the single root is sufficient to keep the object from being reclaimed. With this in mind, the previous depth-first-search function might be coded as shown in figure 21.

When new nodes are allocated, they should be referenced by roots, not by raw pointers. This guarantees that if a garbage collection occurs, they will not be reclaimed. However, since in this case no new nodes are being allocated, it is safe to optimize away the indirection for the majority of the function.

Note the recursive calls to dfs():

```
if (dfs(fastptr->left_child,key,value))
   return 1;
return dfs(fastptr->rightptr,key,value);
```

The actual function arguments to the two calls, in the first argument place, are raw pointers. This is so because they are internal pointers, contained within a node in the data structure, and this collector does not require any special type for such pointers. However, the formal function parameter type for that argument is Root<node>, a different type. In C++, the provider of an ADT can define implicit type conversions to that

```
// Store value return 1 if found, else return 0
int dfs(node * root, String key, int& value) {
   if (root == NULL) return 0;  //dead end
   if (visited(root)) return 0; //been here?

   if (root->key() == key) {
      value = root->value();
      return 1;
   }

   if (dfs(root->leftptr,key,value)) return 1;
   return dfs(root->rightptr,key,value);
   return 0;
}
```

Figure 19: A depth first search using pointers

```
int dfs(Root<node> root, String key, int& value){
   if (root == NULL) return 0;  //dead end
   if (visited(root)) return 0; //been here?

   if (root->key() == key) {
      value = root->value();
      return 1;
   }

   if (dfs(root->leftptr,key,value)) return 1;
   return dfs(root->rightptr,key,value);
}
```

Figure 20: A depth first search using roots

```
int dfs(Root<node> root,String key,int& value){
   register node * fastptr = root;

   if (fastptr == NULL) return 0;  //dead end
   if (visited(fastptr)) return 0; //been here?

   if (fastptr->key() == key) {
      value = fastptr->value();
      return 1;
   }

   if (dfs(fastptr->leftptr,key,value)) return 1;
   return dfs(fastptr->rightptr,key,value);
}
```

Figure 21: An optimized DFS using roots

ADT type. In this case, the root classes can be constructed from raw pointers. So, on entry to the function, a new root is created, meaning a direct table cell is allocated from the root table for class node. That cell is initialized with the raw pointer, actual argument value.

5.2 Marking and Internal Pointers

During the mark phase of a garbage collection, the collector traverses internal pointers of root-referenced objects and marks their referents. There are a number of ways that internal pointers can be identified. Bartlett's collector requires that the internal pointers be located at the beginning of the object, and that a count of the pointers be made available to the collector when the object is allocated [Bar89, Det90]. Detlefs' collector stores map information in an allocator header located immediately before the beginning of the object; the collector interprets this information to locate the internal pointers. Boehm and Weiser's collector scans the entire object conservatively: The size of the object is available in the allocator header preceding the object [BW88].

Our collector associates a *mark* function with every collected type. While marking roots, the appropriate mark function is selected using the language's compile-time function overloading. This is satisfactory for builtin types, such as integers and character arrays, and for user-defined types that are not part of a polymorphic type hierarchy. When marking objects in a class hierarchy, the statically selected mark function must then call a virtual mark function, in order to properly mark the object based on its dynamic type.

One benefit of this approach is that compile time overloading may be used in place of dynamic type checking wherever possible. This allows objects such as integers and character strings to be marked, because an appropriate mark function is chosen statically. Such objects can not have virtual function members, nor any other members, not being aggregate objects. Our previous work required a virtual mark function for every collected type; thus, it could not work with such objects. The drawback to this approach is that when dynamic type checking is required, we make two function calls, one statically chosen and the other dynamically, instead of a single dynamic call.

In the current implementation, mark functions for objects without internal pointers, e.g., character strings or numeric arrays, are generated automatically. Types that have internal pointers must have mark functions supplied by the application programmer. We supply appropriate data structures, such as efficient yet unbounded, type-parameterized stacks and queues, to make it convenient for the programmer to code an iterative traversal to mark the data structure.

For example, later in this paper we will present experiments with a polynomial class. Polynomials are quite simple, being implemented as singly linked lists of terms. The mark function for terms is shown in figure 22.

A mark function for a more complicated type, one that requires explicit stacking, is shown in figure 23.

5.3 Marking and Sweeping

Marking the live objects is accomplished in the following way: the collector examines every cell in the indirection tables. The status of a cell, *active* or *inactive*, is determined by the cell's value. For every active cell, the sub-data structure that it references is marked. First the cell's referent is marked. Then, the referent's descendents are marked. Our first implementation of marking was recursive for simplicity. We have since developed iterative code with explicit stacking (or queuing) to increase mark efficiency.

```
struct term {
    double coefficient;
    int exponent;
    term * next;
};

void mark(register term * p)
{
    while (p) {
        // set mark bit and check old value
        if (gc_mark(p))
            return; // already marked
        p = p->next;
    }
}
```

Figure 22: Marking a 'linked list of terms' type

```
class object {
    ...
    object * left;
    object * middle;
    object * right;
};

void mark(object * ptr)
{
    Stack<object *> stack;

    stack.insert(ptr);
    while (stack.remove(ptr)) {
        if (ptr && !gc_mark(ptr)) {
            stack.insert(ptr->left);
            stack.insert(ptr->middle);
            stack.insert(ptr->right);
        }
    }
}
```

Figure 23: Marking a ternary node type

Sweeping is accomplished with a call to the allocator's sweep() routine. That causes every object that is both allocated and unmarked to be deallocated. In the process, all of the mark bits are cleared. In addition, as described in §5.5, a function call may be performed on each object to *finalize* the object immediately before it is deallocated.

5.4 The C++ Interface

As does the copy collector, this collector uses parameterized C++ classes as smart pointers. When one of these smart pointers is created, it allocates a cell in a root table. The indirect root references the cell which in turn directly references the object.

The type cell<Foo> is the type for one of the cells in the root table. This type is a C++ union. A union is a form of variant record: It may contain any one of its members at a time. This is appropriate for the type cell because it contains either a direct pointer, or a link to another cell, but never both at the same time. <Foo> is an unbound type parameter, giving these cells a generic, parameterized type. The type parameter is the type of object that the cells reference, when they contain a direct pointer. The definition of cell<Foo> is the following:

```
template<class Foo>
union cell {
    Foo * ptr;
    cell<Foo> * link;
};
```

Each table is an array of cells, and one table pointer. The table pointer is that table's link in the list of tables. Every time a root table is created, its constructor links all its cells into a linked list. This is show in figure 17. Note that while a table has a constructor to put its cells into a linked list, it does not require a destructor. A table is defined as follows:

```
template<class Foo>
struct table {
    /* the direct pointer cells */
    cell<Foo> vec[TABSIZE];

    /* our link in the linked list of tables */
    table<Foo> * next;

    /* constructors */
    table();
    table(table<Foo> * initial_next);
};
```

The actual RootTable consists of the initial table and some linked list pointers. The linked list pointers indicate the head and tail of the linked list of cells. These pointers allow a new array of cells to be allocated, with its cells linked to the end of the existing linked list. The pointer to the head of the linked list of free cells is shown in figure 16; the tail pointer is not shown. The root table definition is shown in figure 24; some members and operations have been elided.

5.4.1 Using roots

When a smart pointer is created, it allocates a root table cell. When it is destroyed, it returns its cell to the free list of cells. Every smart pointer's contents is a pointer to a cell in a cell array. That cell contains the corresponding direct pointer. The smart pointer

```
template<class Foo>
class RootTable {
 private:
   cell<Foo> * head; // free list head
   cell<Foo> * tail; // free list tail
   table<Foo> tab;   // array of direct pointers
   void mark();      // to mark from every root

 public:
   RootTable();      // constructor
   ~RootTable();     // destructor

   cell<Foo> * get(Foo * initial); // get a cell
   cell<Foo> * get();
   void put(cell<Foo> * cellp); // release a cell
};
```

Figure 24: The Root Table definition

This is the definition of the parameterized root table class. In the interests of simplicity, numerous function members are not shown.

class has the "dereference" and "dereference-member-select" operators, * and ->, overloaded. Those operators dereference the direct pointer rather than the indirect pointer. Therefore, dereferencing a smart pointer is one indirection more expensive than dereferencing a raw pointer.

The assignment operator is overloaded for the smart pointer. Another smart pointer, a raw pointer, or a constant NULL pointer may be assigned to a smart pointer. Doing so does not change the indirect pointer; that always references the same cell. Instead, the assignment changes the contents of the cell. For this reason, assignment to a smart pointer entails a level of indirection. The comparison operators are implemented analogously to assignment: comparison of a smart pointer to another smart pointer, to a raw pointer, or to a NULL pointer constant compares the direct pointer value in the root table, rather than the indirect pointer value. The majority of the class definition for these smart pointers is shown in figure 25. Some members have been omitted to make the class easier to understand at a glance. The class contains a single static data member, named tab of type RootTable. This declares the direct table for these roots. This direct table is shared by all of the smart pointers of this particular type, Root<Foo>.

One of the most significant advances of C++ over C is support for polymorphic type hierarchies. A garbage collector for C++ must operate with polymorphic data structures to be most useful. This collector supports polymorphic type hierarchies by using

```
template<class Foo>
class Root {
 private:
   /* the table of direct pointers */
   static RootTable<Foo> tab;

   /* the indirect pointer */
   cell<Foo> * iptr;

 public:
   ~Root() { tab.put(iptr); }
   Root()  { iptr = tab.get(NULL); }
   Root(const Foo * p) { iptr=tab.get(p);}
   Root(Root<Foo>& i)
      { iptr = tab.get(i.iptr->ptr); }

   operator Foo * ()       { return iptr->ptr; }
   class Foo& operator*()  { return *iptr->ptr; }
   class Foo* operator->() { return iptr->ptr; }
   int operator==(Root<Foo> r) { /* compare */ }
   int operator!=(Root<Foo> r) { /* compare */ }

   const Root<Foo> & operator=(Foo * p)
      { iptr->ptr = p; return *this; }
   const Root<Foo> & operator=(Root<Foo> r)
      { iptr->ptr = r.iptr->ptr; return *this; }
};
```

Figure 25: The indirect Root class

one root type for every type in the hierarchy. The root types support implicit type conversions corresponding to the valid conversions of the raw pointer types.

5.4.2 Memory allocation

In C, a library writer can supply an alternate memory allocator and require clients to use that memory allocator in place of the standard malloc() memory allocator. However, this is unsafe because there is no way to guard against the client accidentally calling malloc(). In C++, access to the dynamic memory allocator is through the new and delete operators, rather than through ordinary function calls. A class provider is able to overload those operators for allocating objects of the class type. That way, the standard syntax for allocating objects dynamically causes space to be obtained from the class-specific memory allocator, rather than from the standard one. Access to the standard memory allocator is still permitted for objects such as strings and vectors. Figure 26 shows a sample class declaration of a collected class that includes the overloaded free-store operators. These operators obtain memory from and return it to the garbage-collection specific memory allocator rather than the standard one.

```
class anything {
    ...
public:
    virtual void mark();
    void * operator new(size_t sz)
                { return gc_malloc(sz); }
    void   operator delete(void * p)
                { gc_free(p); }
};
```

Figure 26: A sample collected class.

Our memory allocator is derived from Lea's *libg++* allocator [Lea91]. It uses lists of blocks whose sizes are powers of two as well as intermediate sizes. To satisfy an allocation request, the smallest block size that is large enough is used. If no block of the desired size is available, a larger block is broken up. We have modified Lea's allocator to support mark bits and sweeping.

The allocator maintains a bitmap with mark bits for all the objects that it can allocate. The mark bits are stored contiguously to improve garbage collection locality.

These functions have been added to the allocator to support collection:

mark(p) marks the object referenced by p and returns the previous value of its mark bit.

marked(p) returns the value of p's mark bit.

sweep() iterates over all the objects, deallocating every allocated, unmarked object—all mark bits are cleared.

5.4.3 Raw pointers

This system does not attempt to completely prevent the programmer from acquiring raw (direct) pointers. Indeed, the careful use of direct pointers represents a useful (if dangerous) optimization. This also provides *weak* pointers. A weak pointer is a reference that does not cause an object to be retained during garbage collection [Mil87]. The object will be retained only if it is also referenced by a non-weak pointer. Ideally, an access through a weak pointer can detect when the object has been deallocated. That is not possible when using raw pointers for weak pointers.

The programmer must take care never to have an object whose *only* references are direct pointers; the object would be erroneously garbage collected.

5.5 Finalization

C++ supports initialization and destruction of objects though constructors and destructors. The constructor initializes an object and the destructor de-initializes it. Destructors are essentially synchronous: They execute when the object becomes inaccessible. In the case of local variables, for example, this is when the variable leaves scope.

The use of constructors is perfectly consistent with garbage collection. Destructors, however, present a problem. Precisely when garbage collected objects become inaccessible is generally unknown. This renders synchronous destruction impossible [Str91]. Instead, we provide *finalization* [Lam83].

Finalization is essentially the equivalent of an asynchronous destructor. When an object is garbage collected, immediately before its storage is deallocated, the object is finalized. Thus, finalization is used to "clean-up" after an object. For instance, if an object has hardware resources associated with it, finalization is used to free those resources when the object is collected.

Previously, our copy collector would invoke a destroy() function on objects when they were collected. Destroy could be defined as an inline nil function if the user wished to avoid this overhead. However, this resulted in an undesirable source code dependency and loss of generality in the collector. We discuss other implementations of finalization when we discuss the status of the collector, in §5.9.

5.6 Efficiency

There are two components to the efficiency of the system: garbage collection efficiency and overhead due to the use of indirect pointers. We consider the last component to be the more critical, because using roots is a more frequent operation than garbage collecting.

For root efficiency, we present measurements in terms of instruction counts. This is a good measure of complexity because it is independent of system load, clock speed, or other architecture-dependent parameters. On common RISC processors, instructions that reference memory tend to be more expensive than pure register operations, because memories are slower than processors [Pat85]. Likewise, branches are disruptive because they interrupt instruction pipelines. Therefore, in our tables of complexity, we indicate not just the number of instructions, but also the number of memory references and whether or not a branch is involved.

The statistics reported are from the SPARC architecture, having been compiled with the AT&T 3.0 Cfront C++ compiler and the Sun C compiler, with optimization enabled. Before compiling the statistics, we examined the generated assembly code to check

that reasonably efficient code was being generated. In places where the Sun C compiler performed poor register allocation that resulted in unnecessary memory accesses, we report the number of instructions and memory references that the operation *should* take. (This was only an issue in one case, where the compiler generated sub-optimal code for assignment of reference counted pointers.)

We always present the reader with a comparison. For example, we compare the overhead of roots to that of raw pointers, and to that of reference counting. We present *static* instruction counts. Initially, we intended to present the common case dynamic instruction counts as well, but they were nearly always equal to the static counts, and they reduced readability of the tables, so we have omitted that information. The only times when dynamic counts differ from static counts is when a reference counted pointer has a NULL value. In that case, the corresponding reference count does not need to be loaded, incremented or decremented, and stored.

The following four subsections report the complexities of four common operations on roots: copy-construction, assignment, dereference, and destruction. As mentioned in §3.2.2, copy-construction is the initialization of one object with another of the same type, for example, the initialization of a function formal parameter with the actual argument, the copying of a function return value to a temporary in the caller's context, and the explicit initialization of one object with another of the same type. Assignment is simply the assignment of one object with another. Dereferencing is, of course, using the various types of pointers to access the referenced object. Destruction is the instruction count required to destroy the pointer. For raw pointers, there is no required destruction code. Reference counted pointers and roots both have associated destruction requirements.

For reference counted pointers, before adjusting a reference count it is first necessary to determine whether or not the pointer is NULL. If the pointer is NULL, two memory references are avoided at the cost of a conditional branch.

5.6.1 Copy constructor efficiency

Table 2 presents data on the number of instructions it requires to initialize one pointer with another. This is (by definition) the complexity of the copy constructor. We report the machine language instruction counts for copying the three kinds of pointers: raw pointers, reference count pointers, and roots. The complexity of copying one of these pointers depends on where the operands are stored. In particular, all the operations are less efficient if the operands are in memory than if they are in registers. Therefore, for each pointer-type, we present a range of values indicating the *possible* costs: The low cost assumes both operands are in registers and the high assumes both are in memory.

Table 2: Copy Constructor Instruction Counts

Pointer Type	SPARC Instructions				
	instructions		memory references		branches taken
	low	high	low	high	
ptr	1	4	0	2	0
ref	6	9	2	4	up to 1
root	5	9	4	6	0

ptr means a raw pointer.
ref means a reference counted pointer.
root means one of our indirect roots.

The *low* case is for two register operands.
The *high* case is for two memory operands.

5.6.2 Assignment efficiency

The next comparison examines the efficiency of assignment between pointers of the various types. Between two raw pointers, assignment means copying the pointer value. Between two reference counted pointers, assignment can require adjusting both reference counts and possibly deallocating one object, depending on whether either pointer is NULL and if a reference count becomes zero. Between two roots, assignment requires two indirections. Table 3 indicates the instruction counts for the various flavors of assignment.

Table 3: Assignment Instruction Counts

Pointer Type	SPARC Instructions				
	instructions		memory references		branches taken
	low	high	low	high	
ptr	1	4	0	2	0
ref	13	23	4	7	up to 3
root	2	6	2	4	0

ptr means a raw pointer.
ref means a reference counted pointer.
root means a root.

The *low* case is for two register operands.
The *high* case is for two memory operands.

Assignment of reference counted pointers is complex because if the destination pointer's previous referent has a count of one, then that object must be deallocated.

Note how much more complex it is to assign one reference counted pointer to another, versus initializing

one with another. The difference is due to the fact that in initialization, it is known that the destination pointer does not reference an object. Therefore, there is no need to examine that reference count, neither to decrement it nor to see if the object must be deallocated.

Assignment of reference counted pointers is quite a complex operation. The statistics we present assume that these operations are compiled into inline code. However, 23 instructions make a very long inline operation. It could be better to make assignment of reference counted pointers a function call, which would reduce code size but add yet additional branches and memory references.

Before taking these instruction counts, we conducted execution-time tests of all these operations. We found that in many cases the compiler optimized away the indirection required in assignment of roots. The instruction count tables do not reflect that it is both easy and *safe*, in many cases, for the compiler to optimize away the indirection associated with the assignment and dereferencing of roots.

5.6.3 Dereferencing

One of the most important pointer operations is dereference. Since they are direct pointers, reference counted pointers are identical to normal pointers in terms of the efficiency of this operation. Our roots, on the other hand, add a level of indirection. Table 4 presents this information.

Table 4: Pointer Dereferencing

Pointer Type	SPARC Instructions				
	instructions		memory references		branches taken
	low	high	low	high	
ptr	1	3	1	2	0
ref	1	3	1	2	0
root	2	4	2	3	0

ptr means a raw pointer.

ref means a reference counted pointer.

root means a root.

5.6.4 Destruction

Table 5 presents the number of instructions required to destroy the indicated pointer type. For plain pointers, no instructions are required. For reference counted pointers, if the pointer is non-NULL, then the count must be decremented. If the count drops to zero, then the object must be deallocated. For roots, it is necessary to load the head of the free-list of roots, and store it in the root table cell of the pointer being destroyed. Then, the address of that cell is stored as the new head of the free list.

Table 5: Pointer Destruction

Pointer Type	SPARC Instructions				
	instructions		memory references		branches taken
	low	high	low	high	
ptr	0	0	0	0	0
ref	7	9	2	3	up to 2
root	4	5	3	4	0

ptr means a raw pointer.

ref means a reference counted pointer.

root means a root.

5.6.5 A note on efficiency

Much of the information presented in the preceding four subsections makes roots appear much more desirable than reference counting, particularly in terms of code size. There are a few aspects that should be kept in mind before jumping to this conclusion. Chief among these is the cost of external fragmentation in the root tables. This is discussed in §5.8.

5.6.6 An application

In a previous paper [Ede92], we empirically compared the efficiency of an early version of this mark and sweep collector to that of the Boehm/Weiser conservative collector. The application we used was a VLSI CAD application called *ITEM* that performed logic minimization on If-Then-Else tuples. That application was built with the *g++* Gnu C++ compiler. Our current collector requires C++ templates, which are currently only implemented in a small number of compilers, not including g++. Furthermore, because of differences between the compilers, it is not feasible to compile ITEM with Cfront 3.0, the compiler we use. For these reasons we are no longer able to perform our tests with the ITEM application.

We have tested various reclamation algorithms in a small C++ program that creates and manipulates a *polynomial* abstract data type. Polynomials are represented as linked lists of <coefficient,exponent> pairs. The test program creates and sums a large number of polynomials. In the process it allocates about 5 megabytes of data, though for the problem sizes we tested, there is never more than a megabyte in use at any given time.

The polynomial application was originally coded to use manual reclamation. We modified it to use reference counting, our garbage collector, and the

Boehm/Weiser conservative collector. (In the process we found and patched several memory leaks.) Then, we executed each version of the program and graphed the results. The execution times are presented in figure 27.

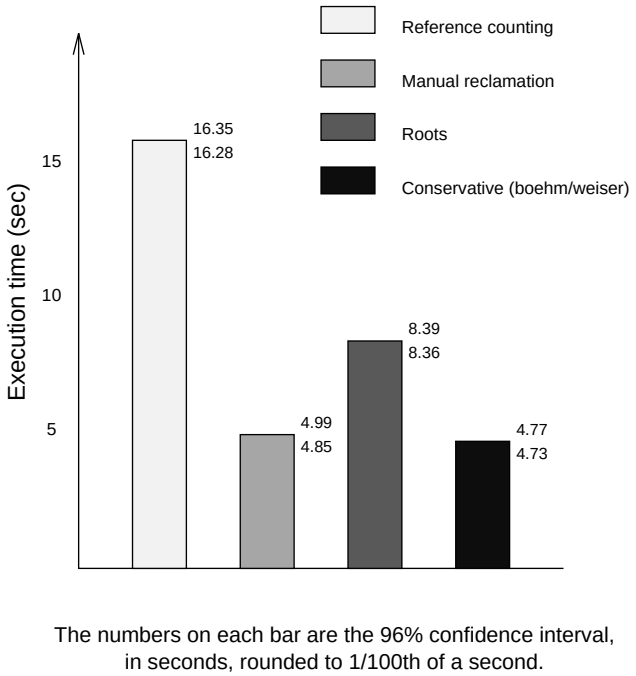

Figure 27: Execution time for the polynomial application

Of the four executions compared in table 27, the version using conservative garbage collection uses the memory allocator supplied with the Boehm/Weiser conservative garbage collector. The other three versions all use the memory allocator that is supplied with our collector. In separate tests, we have found the Boehm/Weiser allocator to be about 20% faster than our allocator. This efficiency comparison is not what we are interested in, since nearly any allocator should be usable with either garbage collection technique.

5.7 Contrast with Conservative Collection

For the experiments that we report in this paper, conservative collection using the Boehm/Weiser collector appears to be the most efficient solution. However, the differing efficiencies of the memory allocator parts of the collectors mask important aspects of the efficiency. Furthermore, there are risks associated with the use of conservative collection. For example, Wentworth has found that conservative collection can perform poorly in terms of retention of garbage in densely populated address spaces [Wen90]. In [Ede92], we found that conservative collection was unable to reclaim any data from 4-8 megabyte garbage data structures. The reason was that the garbage data structures were strongly connected, and a few false pointers in global data into the data structure prevented any garbage from being reclaimed. By contrast, since our collector is not conservative, it does not suffer from this problem.

5.8 Problems

One difficulty with our approach is the possibility of fragmentation in the root tables. The root indirection tables can grow without bound, yet once expanded, there is no way to shrink them. This leads to the pathological case in which large numbers of roots are needed in one table for one type of root, then in another table for another root type, ... We have no way of allowing memory assigned to one table to be used for another.

One possible solution to this problem would be to have only a single table for all roots. The advantage to having multiple tables is that each table can be associated with one particular type of object, and then static function overloading can be used to select the appropriate mark function for each table. When a single table is used for all roots, static overloading can no longer select the mark function; some other, dynamic, technique is required. There are various possible implementations of this: adding structure tags, for example, would enable the collector to select the appropriate mark function at runtime.

5.9 Status

Everything except for finalization works. We had previously implemented finalization very efficiently, but in a way that coupled the memory allocator very closely to the application. There are a number of possible alternate implementation of finalization, such as passing either an integer tag or a function pointer to the memory allocator when objects are allocated. We have not yet chosen a replacement implementation.

We have taken extensive care to make getting and putting a root table cell as efficient as possible for the common case. These are critical operations, since they are invoked every time an indirect pointer is created or destroyed. By contrast, we have not substantially optimized either the memory allocator, or the sweep phase of the garbage collection. The allocator that is part of the Boehm/Weiser collector is much faster than ours. The combination of their allocator with our collector technology would have substantially better efficiency than our current implementation. Likewise, a little bit of tuning in the collector should result in better collection efficiency.

5.10 A More Efficient Implementation

In the current implementation, obtaining a direct table cell (get()) requires four memory references and releasing it (put()) requires two. Of these memory references, a small change of design plus one global, dedicated register could eliminate four of the six.

As described, the collector uses one root table, itself comprised of a linked list of pointer tables, per type of smart pointer. This allows all mark functions to be selected at compile time using function overloading based on the pointer/root type. As a result, this implementation does not require tagged structures to select the mark function. The trade-off of this approach is that every get() and put() operation of a direct pointer cell must load the free cell list head for the appropriate root table.

It would be better to implement this using only a single root table for all smart pointers. Using a single root table, the free list head could be kept in a register throughout the duration of a program's execution. This would reduce by two-thirds the number of memory references required for managing the smart pointers. However, given a single root table, the mark functions would no longer be selectable using static type information. It would be necessary to use dynamic type information, structure tags, for example, for marking. The added efficiency for root operations almost certainly makes this worthwhile.

6 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++. Several of these collectors have been made publically available, as ours will be in the near future.

Boehm et al. have conducted research in conservative garbage collectors [BDS91, BW88]. Their garbage collectors work without any compiler support in languages like C and C++. These collectors are sequential and parallel non-generational mark-and-sweep collectors. Russo has adapted these techniques for use in Choices, a C++ object-oriented operating system toolkit [Rus91, RMC90]. Since they are fully conservative, during a collection they must examine every word of the stack, of global data, and of every marked object.

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. The root-referenced objects must not be moved because the roots can not be modified. Those objects are promoted by having the space identifiers of their pages advanced. Then the root-referenced objects are (type-accurately) scanned; the objects they reference are compactly copied to the new space. This collector works only with non-polymorphic data structures.

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.[1]

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.

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 semantics and reference counting on its referent. OATH uses a three-phase mark-and-sweep algorithm. 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 root set for a standard mark-and-sweep garbage collection.

Goldberg describes tag-free garbage collection for polymorphic statically-typed languages using compile-time information [Gol92] building on work by Appel [App89]. Goldberg's compiler emits functions that know how to collect garbage at various points in the program. Upon a collection, the collector follows the chain of return addresses up the run-time stack. As each stack frame is visited an associated 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.

[1] 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.

The collectors by Boehm, Bartlett and Kennedy are implemented in libraries. Goldberg's and Detlefs' collectors are compiler-based.

7 Conclusions

Garbage collection for C++ is a difficult and important problem. The language philosophy does not permit traditional techniques such as tagged pointers. There is a wide variety of alternatives that have been proposed or are possible. These include Bartlett's Mostly Copying collector, Boehm's conservative collectors, Kennedy's reference counting collector, our mark-and-sweep collector, and others. There is not yet a type-accurate, generational collector that supports polymorphism, nor has any particular collector gained widespread use.

Conservative garbage collection is an important and useful technique. It can be very efficient and easy to use. However, it is not the ultimate solution to the problem of automatic dynamic memory reclamation. For certain types of applications, conservative collection can fail to reclaim satisfactory amounts of garbage. Such applications require type-accurate garbage collection. More information about the *average reclamation rate* of conservative garbage collection in real applications is needed.

We have presented the techniques that support our mark-and-sweep collector for C++. This has been implemented in a library and we are currently testing and improving it. Our short term goal is to improve its efficiency. Our long-term goals are resolving the root-table fragmentation problem, parallelizing it, and adding generations. This promises to yield a C++ garbage collector that is consistent with the language and useful in a wide variety of applications.

8 Availability

The code described in this paper is available via anonymous ftp from nuri.inria.fr (128.93.1.26). It is in ~ftp/scratch/edelson/m-s-code.TAR.Z. The polynomial application is in the file poly.C in the same subdirectory.

Appendices

A Lexicon

accessible A object is accessible if it can be reached by following a sequence of pointers beginning with any root.

collector This term refers to the garbage collector. It can refer to either the algorithm or to a concurrent garbage collection process (thread).

conservative Some garbage collectors do not distinguish between pointers and integers. A collector is called conservative if it decides whether or not a word is a pointer without any type information. Any word that might be a pointer is so interpreted.

constructor A C++ method that is used for initialization: If a classhas one (or more) constructors, then every time the class is instantiated a constructor is executed to initialize the newly created object.

data structure This is the graph of all the dynamically allocated objects.

destructor A C++ method that is used to deinitialize objects.

forwarding pointer Copy collectors move objects, in the process storing a pointer to the new location at the old location. That pointer is called a forwarding pointer. It is used to prevent objects from being copied more than once.

inaccessible Any allocated object that is not accessible is inaccessible. Intuitively, this means that the application is unable to reference the object, and thus the object should be deleted.

internal pointer An internal pointer is a reference contained *within* the data structure. That is, it's a reference contained within a dynamically allocated object.

garbage The term "garbage" refers to an inaccessible object, or collectively to all inaccessible objects.

mutator The mutator is the application. It acts to change, or *mutate*, the data structure. If the application is a parallel program then there are *mutators*.

reachable This term is synonymous with "accessible".

root A root is a pointer that the application can use to access the data structure. Thus, the roots are all the pointers on the stack, in static data, and in the registers, that reference objects in the heap.

structure tag A structure tag is a key field contained within an object that identifies the type of the object. This facilitates garbage collection because the collector can use this field to infer where within the object there are internal pointers. C++ does not use structure tags.

tagged pointer A system based on tagged pointers uses one bit to distinguish between pointers and integers. A garbage collector can then locate pointers by searching for words with their bit set (or unset, as per the implementation.) This either requires custom hardware, or else reduces the range of integers.

type-accurate A type-accurate garbage collector is able to determine which values are pointers and which are not. Only values that are pointers are interpreted as pointers.

unreachable This is synonymous with "inaccessible".

virtual function A C++ function that is not fully bound at compile time. Such functions are used to implement polymorphism.

References

[ACM91] ACM. *Proc. Programming Languages Design and Implementation*, June 1991. SIGPLAN Notices 26(6).

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, April 1991. SIGPLAN Notices 26(4).

[ANS89] ANSI X3.159-1989, 1989. American national standard for the C programming language.

[ANS93] Draft proposed international standard for information systems—Programming language C++, January 1993. ANSI document X3J16/93–0010, ISO document WG21/N0218.

[App89] Andrew W. Appel. Runtime tags aren't necessary. In *Lisp and Symbolic Computation*, volume 2, pages 153–162, 1989.

[Bak78] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation, Western Research Laboratory, Palo Alto, California, February 1988.

[Bar89] Joel F. Bartlett. Mostly copying garbage collection picks up generations and C++. Technical Report TN–12, DEC WRL, October 1989.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proc. Programming Languages Design and Implementation* [ACM91], pages 157–164. SIGPLAN Notices 26(6).

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 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.

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, September 1981.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Proc. Object-Oriented Programming Systems Languages and Applications*, pages 49–70, October 1989. SIGPLAN Notices 24(10).

[Det90] David Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie Mellon, 1990.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–974, November 1978.

[DN66] O. J. Dahl and K. Nygaard. Simula—An Algol-based simulation language. *Communications of the ACM*, 9:671–678, 1966.

[Ede92] Daniel R. Edelson. A mark-and-sweep collector for C++. In *Proc. Principles of Programming Languages*, pages 51–58. ACM, ACM, January 1992.

[EP89] Daniel R. Edelson and Ira Pohl. Solving C's shortcomings: Use C++. *Computer Languages*, 14(3), 1989.

[EP91] Daniel R. Edelson and Ira Pohl. A copying collector for C++. In *Proc. Usenix C++ Conference* [Use91], pages 85–102.

[FY69] R. Fenichel and J. Yochelson. A LISP garbage-collector for virtual-memory systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[Gol92] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proc. Programming Languages Design and Implementation* [ACM91], pages 165–176. SIGPLAN Notices 26(6).

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Publishing Company, Reading, MA, 1983.

[ISO90] ISO 9899-1990, 1990. International standard for the C programming language.

[Ken91] Brian Kennedy. The features of the object-oriented abstract type hierarchy (OATH). In

Proc. Usenix C++ Conference [Use91], pages 41–50.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison, Wesley, Reading, Mass., 1973. Second ed.

[KP90] Al Kelley and Ira Pohl. *A Book on C: Second Edition*. Benjamin/Cummings, Redwood City, California, 1990.

[KR88] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, Englewood-Cliffs, N.J., 2nd ed. edition, 1988.

[Lam83] Butler W. Lampson. A description of the Cedar language: A Cedar language reference manual. Technical Report CLS-83-15, Xerox PARC, 1983.

[Lea91] Doug Lea. A memory allocator for *libg++*. Private communication, 1991.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Mil87] J. S. Miller. *Multischeme: A Parallel Processing System Based on MIT Scheme*. PhD thesis, MIT, 1987. MIT/LCS/Tech. Rep.-402.

[Min63] M. L. Minsky. A LISP garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project Mac, MIT, Cambridge, MA, December 1963.

[Pat85] David Patterson. Reduced Instruction Set Computers. *Communications of the ACM*, 28(1):8–21, 1985.

[PE88] Ira Pohl and Daniel R. Edelson. A–Z: C language shortcomings. *Computer Languages*, 13(2), 1988.

[Poh92] Ira Pohl, 1992. Private communication.

[RMC90] Vincent Russo, Peter W. Madany, and Roy H. Campbell. C++ and operating systems performance: A case study. In *Usenix C++ Conference*, pages 103–114. Usenix Association, April 1990.

[Rus91] Vincent Russo. Using the parallel Boehm/Weiser/Demers collector in an operating system, 1991. Private communication.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[Ung86] David Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. The MIT Press, Cambridge, MA, 1986.

[Use91] Usenix Association. *Proc. Usenix C++ Conference*, April 1991.

[War87] The Soft Warehouse. *muLISP Reference Manual*. Honolulu, 1987.

[Wen88] E. P. Wentworth. *An environment for investigating functional languages and implementations*. PhD thesis, University of Port Elizabeth, South Africa, 1988.

[Wen90] E. P. Wentworth. Pitfalls of conservative garbage collection. *Software – Practice and Experience*, pages 719–727, July 1990.

[Wik87] Ake Wikstrom. *Functional programming using standard ML*. Prentice Hall, 1987.

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