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

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

[Gui91] Ron Guilmette, February 1991. Usenet comp.lang.c++ article: Re: Smart pointers and stupid people.

[HD90] Richard Hudson and Amer Diwan. A copying collector for Modula-3 and Smalltalk, 1990. private communication.

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

[Koe90] Andrew Koenig. Objects reclaimed by a copying collector must not have destructors, 1990. private communication.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

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

[Ste84] Guy L. Jr. Steele. *Common Lisp: The Language*. Digital Press, Burlington, MA, 1984.

[Str87] Bjarne Stroustrup. The evolution of C++ 1985 to 1987. In *Usenix C++ Workshop Proceedings*, pages 1–22, Santa Fe, NM, November 1987. Usenix Association.

[Ung84] David Ungar. Generation Scavenging: A non–disruptive high performance storage reclamation algorithm. In *ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, PA, April 1984. Association for Computing Machinery.

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

A longer report that describes this work can be obtained via anonymous ftp from midgard.ucsc.edu (128.114.14.6). It is in pub/tr/ucsc-crl-90-19.ps.Z. This includes a printed copy of the source code that implements the prototype. Printed copies of the technical report are available. Write to:

Jean McKnight
Technical Librarian
Baskin Center for Computer Engineering & Information Sciences
University of California
Santa Cruz, CA 95064

Internet: jean@cis.ucsc.edu

References

[Ame89] ANSI C Standard, 1989. American National Standard X3.159-1989.

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[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.

[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.

[Ede90] Daniel Edelson. Dynamic storage reclamation in C++. Technical Report UCSC-CRL-90-19, University of California at Santa Cruz, June 1990. M.S. Thesis.

[EP90] Daniel Edelson and Ira Pohl. The case for garbage collection in C++, August 1990. Workshop on Garbage Collection in Object-Oriented Programming Languages, in conjunction with OOPSLA/ECOOP '90.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, February 1990.

not enable an application programmer to prevent the creation of dumb pointers to a class.[2] In a compiler-based implementation this will not be an issue.

5.3 Summary

Efficient management of dynamic data is difficult. Only in simple cases can programmer-controlled deallocation be safe, efficient and simple. Reclamation of generalized dynamic graph data structures requires edge traversal to identify unused blocks. Unless blocks are reused, objects are scattered in memory causing excessive paging and using an unnecessarily large amount of backing store.

There are two predominant classes of reclamation algorithms: mark-and-sweep and copying. Both must normally be able to recognize pointers to collected objects. Mark-and-sweep collectors take one pass to mark all living objects and one pass to deallocate inaccessible ones. Another pass may be used to compact objects. Conservative collectors can be implemented without compiler support. These collectors need not differentiate between pointers and integers. They reclaim most, though not all, of the inaccessible memory.

This paper has presented a copying collector and its C++ implementation. The collector will work with lists, trees, DAGs, or cyclic graphs. It incorporates very fast allocation and a novel way of tracking roots that does not require tagged pointers or integers. The collector's work is proportional to the amount of living data, therefore when most objects die, it is highly efficient. This is the only copying collector for C++ known to the author that requires neither tags nor an object table, and that supports polymorphic type hierarchies.

This implementation of the collector requires no modifications to the compiler, but it does require assistance from the programmer. Another version in the compiler would require changes to the language. Such a collector could be more efficient than this one.

The system is composed of encapsulated data structures making it appropriate for an object-oriented imperative programming language. This collector shows a way that garbage collection can be made an efficient, non-invasive part of the C++ programming language. Under this scheme the efficiency of reclaiming a data structure depends exclusively on the complexity of the data structure.

In this system we have accomplished the following:

1. added efficient and reasonably convenient automatic storage reclamation to C++,
2. found an organization for garbage collection in C++ that remains within the philosophy of the language, and,
3. developed a platform for research into new techniques and algorithms, particularly in copying collection and virtual memory issues.

Availability

The prototype collector has not yet been made publicly available. When we have a more robust and easy-to-use implementation, it will be made available via anonymous ftp.

[2] Guilmette has proposed C++ language changes that would accomplish this [Gui91], but it is unclear whether or not the C++ standardization committee X3J16 will adopt the proposal.

Table 2: Collector Efficiency Measurements

Nodes have 12 bytes of data, 4 bytes of forwarding pointer, and 4 bytes of *vptr*.
The data structure of $2^{17} - 1$ nodes requires 2559k bytes.

Operation	Time to Complete	
	VAXstation 3520	SPARCstation 1
Collect "null", 1 root, 1 chunk, no data	$58.5\mu s$	$11.5\mu s$
Build a binary tree: $2^{17} - 1$ nodes	$4.2s$	$2.4s$
Build and collect the tree	$12.3s$	$4.2s$
Time to copy $2^{17} - 1$ binary nodes	$6.0s$	$1.8s$
Time per node (minus overhead)	$45.8\mu s$	$13.7\mu s$
Nodes copied per second	21,845	72,992
Kbytes copied per second	426	1425

5.2 Limitations

One problem, not with this implementation but with copying collection for C++, is that objects are never individually deallocated. Without deallocation of individual objects destructors cannot be called for collected objects. A programmer who provides a collected `class` with a destructor is likely to be surprised when the destructor is never invoked. Never examining dead objects increases the efficiency of copying collectors. This explains why, during a collection, copying collectors do work proportional to the number of living objects rather than number of dead plus living objects. If copying collection is preferred, this cannot be seen as a disadvantage. It is only a disadvantage because it is a divergence from the C++ style. As Koenig observed, destructors must be forbidden for copy-collected objects [Koe90].

The system does not support arrays of collected objects, though it supports arrays of pointers to collected objects. As a policy decision, all objects of a collected type must be dynamically allocated. This allows the garbage collector to avoid checking to see if each object is dynamically allocated. A global or `static` object can be simulated using a global `root` that always references the same object. Non-dynamic objects can be supported at the cost of reduced collection efficiency.

The collector has a scheme for working with foreign roots. However, since destructors are not called for collected objects, a foreign (doubly-linked) root must not be inside an object that is reclaimed with copying garbage collection. If a collected object contained a `droot` to another object, the root would continue to exist even after the containing object was deallocated. Objects containing foreign roots must be reclaimed another way, such as manually.

The prototype system can not prevent the programmer from creating *dumb* pointers. Dumb pointers are those that do not track themselves using the lists. Currently, C++ does

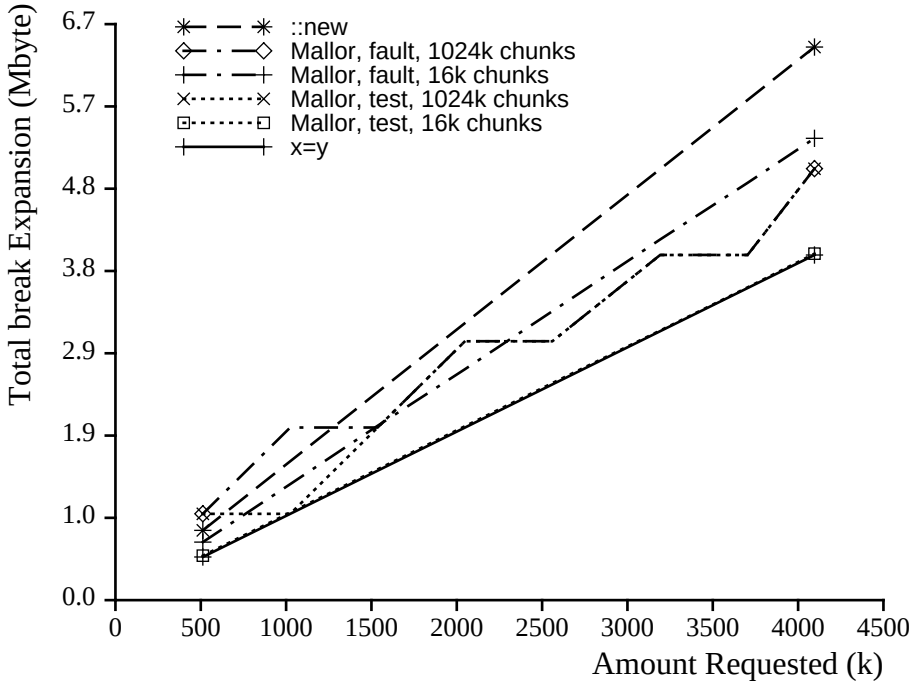

Figure 9: Total memory allocated, SPARCstation 1.

5 Conclusion

5.1 Advantages

Reclaiming dynamically allocated objects can be difficult. With the collector, inaccessible memory is recycled and live objects are compacted. This increases programmer productivity by removing from the application programmer responsibility for deallocating data. Compaction may reduce paging and improve virtual memory performance.

Many copying collector allocators fix the size of the free-store. This allocator will more closely track the actual amount of memory required since, after a collection, it is off by no more than one chunk. The behavior of other allocators is trivially emulated by using a very large chunk size and disallowing expansion.

This collector demonstrates a new way of tracking roots of the data structure. Allocating and initializing n roots in a stack frame requires approximately $n+2$ more memory references than simple pointer allocation and initialization. These roots must be initialized since following uninitialized roots would lead to errors. Iterative code, such as list traversal, when no roots are constructed within the loop, suffers no performance penalty under this scheme.

The allocator supplied with this package is fast. The measurements of §4.2 show that the allocator supplied with this system is much faster than the allocator provided in the standard libraries. This is unsurprising since copying collection permits very efficient allocation.

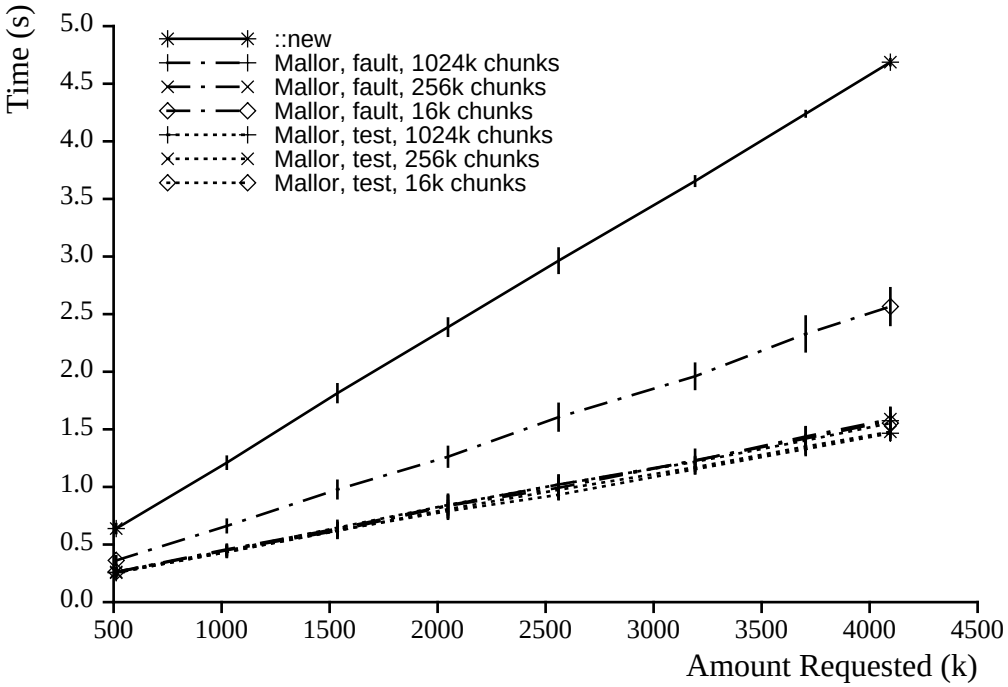

Figure 8: Time to Allocate and Touch: 20 Byte Requests, SPARCstation 1

Mallor: means the custom memory allocator
fault: the version that uses write-faulting to avoid the test
test: the version with an explicit bound check

n roots will use $3n$ instructions, however, optimization reduces that to $n + 2$ instructions.

4.4 Collecting

A garbage collection pass visits every root, and then copies every node reachable from the root. A garbage collection includes processing for the *flip* plus the deep copy of the data structure. The flip is constant time and fast. Deleting from-space is very fast (tens of instructions, not hundreds.) Deeply copying the data structure requires a virtual function call per pointer and an allocation/copy per object. It is difficult to estimate the amount of time this will take because the application defines the copy constructor.

Calls to the allocator take 6 instructions, except when a fault occurs. With 256k chunks faults will be very rare, there would be four if a megabyte were copied. Twenty byte objects, such as those benchmarked in this paper, would require 5 instructions each to copy. If many objects are copied this expense will be significant. The key in copying collection is to wait to collect until many objects have died. That way copying is very fast.

For comparison purposes the collector was benchmarked collecting a graph of 20 byte nodes on both the VAXstation and a SPARCstation. The results are shown in table 2.

```
struct node {  /* sizeof node == 20 */
    static mallor * heap;
    void * operator new(size_t size) { return heap->get(size); }
    void operator delete(void * p)   { }
    char data[20];
};
```

Figure 7: The object used in allocator tests.

and temporaries.

Creating a pointer on the stack normally requires two instructions: decrementing the stack pointer and initializing the root. The stack pointer allocation is performed with one subtraction for all the local variables to the function. Destroying such a variable normally requires no instructions because the stack pointer is restored during the normal function return sequence.

Replacing the pointer with a `root` adds three instructions: two for maintaining the stack and one to initialize the root to NULL. Unlike a normal pointer variable, a `root`'s pointer component must not be left uninitialized. The root destructor requires one instruction. Finally, the compilers use for these tests move an unnecessary value into `r0` for the expression value. Creating, initializing and destroying a root, which should take four instructions, takes 5 under these compilers. The figures in table 1 show the performance of root allocation and destruction.

Table 1: Efficiency of creating `root`s compared to simple pointers.

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

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 by a simple pointer.

Multiple roots will be created and destroyed when there are multiple local variables of type root. An unsophisticated compiler might not optimize their construction and destruction. However, the compiler could safely eliminate all but one `mov` to set the list head, and all but one `mov` to restore the old list head. Thus, the obvious code to construct and destroy

```
/* Coded by the application programmer in node.h */
class node {
    ... // whatever required by the application
    ... // normal constructors, no destructor,
    COLLECTION_MEMBERS(node)
    virtual void copycollect(node * &);
};

/* Invoke parameterized macro to define the root class */
ROOT(node)

/* Coded by the application programmer in node.c */
DECLARE_GC(node)         /* define static data required for GC */
```

Figure 5: Defining a garbage collected class.

```
void node::membfunc()
{
    SAVE_THIS(node)
    ...
}
```

Figure 6: Defining a garbage collected class member function.

program's dynamic storage space limit. It grows toward higher addresses as the program obtains memory from the operating system. The `sbrk` system call with an argument of zero returns the current *break* without changing it. The total memory obtained from the operating system by each allocator is shown in figure 9.

In these tests the standard allocator was roughly 50% space efficient. In general, it's overhead depends on the request size. When requests are equal to or slightly larger than a power of two, the allocator suffers from severe fragmentation, obtaining $2n$ bytes from the operating system to provide n bytes to the application. When objects are slightly smaller than a power of two its overhead is very small.

4.3 Roots

4.3.1 Creating and Destroying a Root

Creating a root means allocating and constructing a pointer variable. Global and `static` variables are created once at the beginning of the program. The efficiency of constructing them contributes little to the efficiency of a program. The critical roots are those allocated on the stack including: local variables, by-value function parameters, function return values,

```
/* Supplied in gc.h */
#define COLLECTION_MEMBERS(T)                                   \
private:                                                        \
    static mallor freestore;                                    \
    static void   gc();                                         \
    struct forward_addr {                                       \
        T * forward;                                            \
        forward_addr() : forward(0) { }                         \
    } fa;                                                       \
    void    set_forward(void * p) { fa.forward = p; }           \
    void *  get_forward()  { return fa.forward; }               \
public:                                                         \
    void *  operator new(size_t n)                              \
                { return freestore.get(n); }                    \
    void    operator delete(void * p)   { }
```

Figure 4: The necessary members of a collected type.

- collecting a data structure

4.2 The Allocator

The time measurements shown in the graph of figure 8 are obtained as follows. Three allocators are compared: the testing version, the faulting version, and the standard `operator new` memory allocator with *malloc*. Each allocator was used to allocate 20 byte objects such as that shown in figure 7. After allocation, one byte was initialized in each object. This seems fair because uninitialized dynamically allocated objects should be rare. The test repeatedly allocates new objects until a fixed amount of memory has been obtained. The test was run to obtain 512k through 4M bytes. The custom allocators are parameterized by their internal chunk size. Chunks of 16k, 256k and 1M bytes were tested. Timing information was obtained with the `wait3` system call. The times reported are user time plus system time. The vertical bars show 95% confidence intervals.

The data presented in the graph of figure 8 show little difference between the two versions of the custom allocator. Both are roughly 2-3 times faster than the standard allocator. The faulting allocator with 16k chunks is slower than the other configurations due to the overhead of handling frequent write-faults.

4.2.1 Space Efficiency

In this context, space overhead is memory obtained from the operating system but not made available to the application. Examination of the process' *break*, before and after all the allocations, indicates how much space each allocator wastes. Under UNIX[1] the *break* is the

[1] UNIX is a trademark of Bell Laboratories.

```
class dnode : public node {
    node * center;
    ...
    virtual void copycollect(node * &);
    virtual void copycollect(dnode * &);
};

void dnode::copycollect(node * & n)
{
    ...
    n = new dnode(*this);  // safe, implicit conversion
    ...
}

void dnode::copycollect(dnode * & n)
{
    ...
    n = new dnode(*this);  // No conversion needed
    ...
}
```

Figure 3: Overloaded copycollect functions for a derived `node` type.

compiler dependent. Member functions that *cannot* directly or indirectly cause collections do not require this, however, omitting it is not recommended.

4 Efficiency

4.1 Overview

This dynamic memory management organization does not impact code that does not use it, thereby satisfying the first important efficiency criterion. The other is that code that does use the collector is also efficient.

The tests reported here were compiled with optimization enabled on a two processor VAXstation 3520, cfront 2.0 and the Ultrix 3.0 C compiler. For comparison purposes many tests were also run on a Sun SPARCstation 1 running SunOS 4.0.3 and cfront 2.0 and are so indicated.

In analyzing the performance of individual components we examine the following operations:

- allocating an object
- creating and destroying roots, both stacked and doubly-linked

```
class node {
    node * left, * right;
    ...
    virtual void copycollect(node * &);
};

void node::copycollect(register node * & nodep)
{
    if (nodep = (node*) get_forward()) return;    // already done?

    nodep = new node(*this);                      // copy object
    set_forward(nodep);                           // set forward ptr
    if (left) left->copycollect(nodep->left);     // copy children
    if (right) right->copycollect(nodep->right);  // copy children
}
```

Figure 2: A copy function for a `node` type.

enough static type information is available for correct pointer conversion. A derived class must have a copy function for every type of pointer that can lead to such an object. A simple derived class based on the `node` class of figure 2 is shown with its `copycollect` functions in figure 3.

3.4 A Collection

The collector is nonincremental and copying. It is invoked by the allocator when an allocation request cannot be satisfied. The collector traverses the root stacks and "visits" each root. From each root, it initiates a depth-first copying collection with a call to `copycollect` through the root. After it has collected, it returns control to the allocator.

3.5 Using the Collector

Under this scheme the attribute *collected* is a characteristic of an object. This contrasts with languages such as Modula-3 [CDG+88] in which *tracked* is an attribute of a pointer.

Figure 4 shows the necessary members of a garbage collected type in the prototype implementation.

Figure 5 demonstrates the definition of a garbage collected type. After this, the programmer does not manipulate "`node *`" pointers. Instead, smart pointers of type `R_node` are used. The final item coded by the programmer is a root for the `this` pointer in every member function. This is required so that, when a collection occurs, `this` pointers on the stack will be updated. An example of this is shown in figure 6. The macro `SAVE_THIS` constructs a root that references the `this` pointer. Unfortunately, current implementations of *cfront* prohibit taking the address of `this`. Therefore, the implementation of this macro is

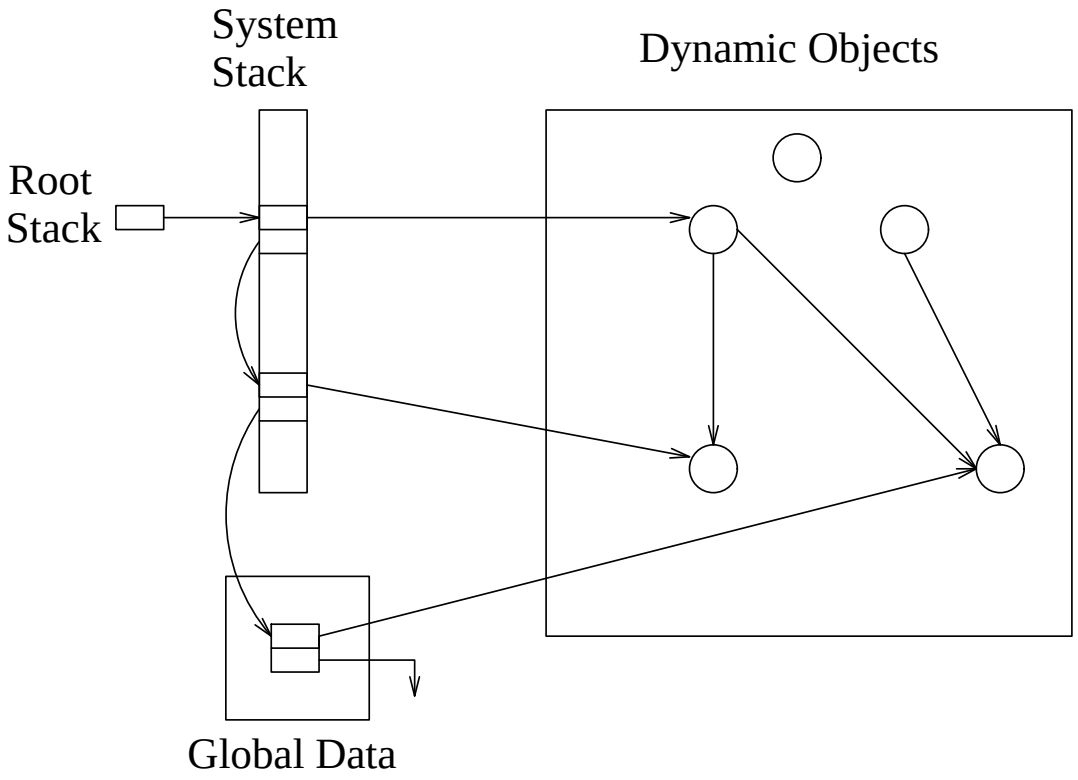

Figure 1: Runtime organization of the copying collector.

The linked-list implementation of the root-stack is shown.
No doubly-linked roots are shown.

3.3 Copying

The current implementation requires that every collected type have a `copycollect()` virtual function. This function copies the object and stores a forwarding pointer in the old copy. Then, `copycollect()` updates the pointer that led to the object with the object's address and recursively copies the descendents. A sample function is shown in figure 2. The explicit type conversion required by this function is both safe and necessary; the reasons are explained in [Ede90].

When an object is copied the pointer that led to the object must be updated with the new address. When the object is of a derived class type, the pointer requires conversion from *derived* `*` to *base* `*`. This conversion, in the presence of multiple inheritance, may change the value of the pointer.

To preserve type-safety, i.e., ensure that the conversion is safe, `copycollect` is overloaded based on the type of the pointer that references the object. This ensures that

In the prototype implementation, roots of a data structure are `class` objects of a parameterized type that behave as *smart pointers* [Str87]. For nodes of type T, roots are of type `R_`T. For example, the programmer does not currently manipulate local variables of type `node *`; objects of type `R_node` are used instead. When the collector is implemented in the compiler the programmer will appear to use normal pointers: smart pointers will be generated instead.

The root classes are currently generated with parameterized `#define` macros because templates [ES90] are not available. When the collector is in a compiler neither macros nor templates will be required. In the remainder of this paper, *roots* are these parameterized class objects. A root can be converted to a normal pointer type. It can be dereferenced like a normal pointer. Roots have constructors and destructors that track their lifetimes.

The roots of the collection are located using two data structures. Most roots are either global or `static` variables, or `auto` variables. These roots can be tracked with a stack. When such a root is created its address is pushed onto a stack; when it is destroyed its address is popped. Other roots, in particular those contained in other dynamic objects, cannot be tracked using a stack. Their addresses are kept in a doubly linked list.

There will generally be multiple stacks (and doubly-linked lists) created, one for each class in the polymorphic type hierarchy. However, that fact is not important in this discussion and is omitted for clarity.

3.2.1 Stackable Roots

When a global root is created, or a root is allocated on the runtime stack, the root's address is pushed onto a stack. All the global and local roots in a program can be found by traversing the stack. The stack was implemented two ways: with linked lists and with arrays.

In the array implementation a separate array of root pointers tracks the addresses of all the roots. Root construction and destruction push and pop addresses from the arrays, respectively. In the linked-list implementation, the list is threaded in the runtime stack, or in global data for non-auto roots. Each root consists of two words, its pointer and its link. A representative runtime organization is shown in figure 1. This figure presents a single root stack, implemented as a linked list, and no doubly-linked roots.

3.2.2 Doubly-Linked Roots

Dynamically allocated roots that are not in the data structure may not have LIFO lifetimes, therefore they cannot be tracked with a stack. This collector tracks non-stackable (foreign) roots with a doubly-linked list. The doubly linked list permits deletion of any list element to support non-last-in first-out (LIFO) insertion and removal.

These roots must have a distinct type. Dynamic roots to objects of type T are of type `DR_`T. They are called *droots* because they insert themselves into a doubly linked list when they are created (doubly-linked roots.) As with stacked-roots there is one list for every kind of root. The collector can find all the lists.

following components: a memory allocator; GC-related members of collected types, for example, overloaded `new` and `delete` operators, and; *smart pointer* [Str87] types.

3.1 The Memory Allocator

The strategy used in this allocator is based on the fast block allocation scheme described in [App87]. It uses a variation that supports discontiguous spaces. A *chunk* is a large block with which the allocator satisfies allocation requests. A *space* is a linked-list of chunks. The active space, that is, the one currently satisfying allocation requests, is known as *to-space*. When a chunk is exhausted the allocator may either begin a collection, or obtain a new chunk and continue allocating. Which one the allocator chooses is a policy decision that is outside the scope of this paper.

The allocator is encapsulated in a C++ `class` called a `mallor`. The common term `malloc` was not used to preclude confusion with the ANSI C [Ame89] library routine of that name. In this discussion the noun *an allocator* denotes an instance of `class mallor`.

Individual objects are not individually freed, therefore dynamically allocated blocks do not requite headers and footers. An entire space is deallocated by returning its chunks to the low-level allocator.

An allocator is constructed with a function pointer to a collection routine. When it runs out of space it takes the following steps:

1. *flip*, i.e., begin allocating from a new space,
2. invoke a garbage collection by calling through the `gc` function pointer, upon return,
3. free the old space, and
4. resume the interrupted allocation request.

This is transparent to the application programmer.

An allocator is a `static` member of the base class of the collected data structure. The overloaded `new` operator of the class obtains memory from the `mallor`.

3.1.1 Implementation

The allocator has two implementations that differ in how they detect that the current chunk is exhausted. One performs an explicit bound check during each allocation. The other uses virtual memory protection as suggested by Appel [App87] to avoid the explicit test. It write-protects the last page of the chunk and allocates until a write-fault occurs. It writes to each object to force the fault. The two versions of the allocator are respectively called the *testing* version and the *faulting* version. Experiments reported in [Ede90] show that in many cases write-faulting is not efficient enough to justify its added complexity.

3.2 Locating Pointers

A "root" of the data structure is a pointer on the stack, in a register, or in global data. This collector uses auxiliary data structures to track the roots.

with their pages set to the next_space value.

After finding roots, the collector scans the promoted objects looking for pointers to objects still in current_space. This scan is not conservative. The Scheme collector uses tags to identify the pointers; the C++ collector requires that the programmer provide a function to locate the pointers. Every object that is still in current_space is compactly copied to next_space. A forwarding pointer is left in the old copy and the pointer that led to the object is updated with the new address. Garbage collection is complete when all the objects in next_space have been scanned for pointers into current_space. When the algorithm has moved or copied all the living objects it sets current_space to next_space and resumes the interrupted allocation request.

The collector was originally non-generational and intended for use with Scheme; Bartlett later added generations and C++ support [Bar89]. Generations are implemented in the page counters. To collect C++ objects, the user provides a function to identify internal pointers. During a collection the collector calls this function to get the offsets of internal pointers. It then traverses the internal pointers and copies the objects.

As described in [Bar89] the collector does not consider the fact that a pointer may point at a derived class object. Therefore, it does not support C++'s flavor of polymorphism.

3 A Copying Collector

Two problems in implementing GC in C++ are: identifying global pointers, and, identifying pointers within objects. C++'s flavor of polymorphism compounds these problems by allowing a pointer of particular static type to point at objects of different dynamic types. Any complete solution must address and solve these problems in the context of polymorphic type hierarchies.

The scheme described in this section is compatible with existing code and libraries, efficient, encapsulated, and fully consistent with C++'s flavor of polymorphism. It is *type accurate* [HD90], meaning that a value it interprets as a pointer is statically typed to be a pointer. This contrasts with conservative collection which interprets integers as pointers. This stop-and-copy collector has the following characteristics.

- The system is modular, encapsulated and very flexible. The application may communicate with the memory manager to configure it appropriately. Any number of collectors and allocators can exist concurrently in an application on disjoint data structures.
- The allocator is fast. Allocation requests compile inline to 7 VAX instructions of which 4 are executed in the common case. It supports discontiguous chunks.
- The collector is a copying collector that collects and compacts in one pass.
- The components are small, simple and efficient. In many cases it is more efficient than a standard manual memory allocator such as the global `new` and `delete` operators or *malloc* and *free*.

A prototype of the collector has been implemented in application code outside of the compiler. An implementation within a compiler is underway. The system consists of the

2 Related Work

2.1 Conservative Collection

The simplest way to accomplish automatic storage reclamation in general C++ programs is to use conservative garbage collection [BW88]. Conservative GC requires virtually no compiler support. It is a variation on traditional mark-and-sweep storage reclamation that does not differentiate between pointers and integers.

When an allocation request cannot be satisfied due to insufficient free storage, the memory allocator invokes a reclamation pass. In the *mark* or *trace* phase, the collector examines the stack, the registers, and global data searching for values that might refer to dynamically allocated objects. Any such value may be either a pointer, an integer, or some other type of data, but it is interpreted as a pointer. These perceived pointers constitute the *roots* of the garbage collection. The objects they reference are marked. Each marked object is itself conservatively scanned for possible pointers. Perceived pointers are followed and objects are marked until there are no more objects or pointers to examine. The marked objects are a superset of the actual reachable data.

After the mark phase the collector examines every dynamically allocated object. Each unmarked object is deallocated. Memory allocators that work with mark-and-sweep collection add a header and footer to every allocated object containing the actual size of the memory block. Thus, given a pointer to a dynamically allocated object, the allocator can determine the size of its memory block [Knu73].

The advantage of conservative collection is that it does not require compiler assistance. However, if there are any integers that look like pointers it retains inaccessible storage. Conservative collection precludes compaction since pointers cannot be altered.

2.2 Partially Conservative Garbage Collection

Bartlett describes a garbage collector that is conservative on the stack and copying in the free-store [Bar88]. The collector was intended primarily for Scheme, but has been adapted for use with C++ programs. The allocator allocates memory from consecutive, uniform size, chunks of memory called *pages*. This page size is unrelated to the hardware page size. Pages are assigned to spaces; a page's space is indicated by an associated space identifier. There are two special spaces, *current_space* and *next_space*; they fulfill the roles of from-space and to-space in the standard copying collector algorithm. That is, a collection moves live objects from current_space to next_space, and then sets current_space equal to next_space. Any page whose space is not equal to either current_space or next_space is currently free.

The algorithm garbage collects after half of the free-store pages are used. Next_space is set to current_space plus one modulo a large number. All living objects will be copied to pages labeled with the next_space value.

Initially, the collector conservatively scans the stack, the registers, and, if appropriate, global data, looking for roots. As in other conservative garbage collection algorithms, it assumes that any value on the stack that might be a pointer is actually a pointer. The referent of each root cannot be moved, because the root might be an integer. These objects are "moved" from current_space to next_space through having the space identifiers associated

the system, and section five concludes the paper with advantages and limitations of the system.

1 The Difficulty with GC in C++

1.1 How Garbage Collection Works

A garbage collector scans the program's global state searching for pointers to dynamically allocated objects. This entails examining the stack, the registers, and the global data. Each time the collector locates a pointer, it traces out the data structure reachable from the pointer. Any object that is reachable from some pointer is *live*. All unreachable dynamically allocated objects are garbage and should be deallocated.

In mark-and-sweep garbage collection each object is marked as it is visited. After all accessible objects have been marked, a *sweep* phase visits all the objects and deallocates the unmarked ones. In copying garbage collection [Bak78,LH83,FY69,Ung84], when an object is visited, it is copied to a new memory space. After the traversal the old memory space is recycled *en masse*.

Garbage collectors face two main problems: locating every global pointer, and, locating every pointer from one object to another.

1.2 Alternatives

Traditional garbage collection schemes use tagged pointers. When the collector examines a value (on the stack, for example), a tag indicates whether the value is a pointer or an integer. Pointers inside objects are also tagged. Thus, this solves both of the problems that we identified in the previous section. Unfortunately for GC proponents, tags are inconsistent with the C++ philosophy. Tagged integers have reduced range and lower efficiency, or else require hardware support. Data in the standard libraries must be tagged, reducing efficiency and penalizing all users of the language.

Alternatively, objects may be allocated from segregated memory pools, with integers, themselves, allocated from one of the pools. In this model everything is a pointer. Pointers to collected objects are precisely identifiable by their values. The added level of indirection reduces the efficiency of integer arithmetic and is another unacceptable solution.

An object table makes pointers precisely identifiable because exactly all the direct pointers are in the table. Pointers that the programmer manipulates are indirect through the object table. Objects' motion does not affect the application. This solution adds a level of indirection to every pointer dereference. Ungar found that eliminating the object table from a Smalltalk system improved efficiency dramatically [Ung86]. It is unlikely that an object table-based implementation of garbage collection would yield satisfactory efficiency.

Another alternative is conservative collection. This is a technique that does not need to differentiate between pointers and integers. Any quantity that might be a pointer, is assumed to be a pointer. Conservative collectors cannot normally move objects. This type of collection is described in §2.1.

The last choice we discuss is garbage collection by pointer tracking. This forms the basis for our collector and is described in §3.

A Copying Collector for C++*

Daniel R. Edelson
daniel@cis.ucsc.edu

Ira Pohl
pohl@cis.ucsc.edu

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

27 February 1991

Abstract

Garbage collection is an extremely useful programming language feature that is currently absent from C++. The benefits from garbage collection include convenience and safety because the programmer is not responsible for freeing dynamically allocated storage. Many reclamation schemes improve efficiency by compacting objects in memory improving locality and reducing paging. Some reclamation techniques are more efficient than manual reclamation for important classes of data structures.

This paper presents a copying collector for C++ that supports polymorphism and does not require indirection through an "object table." This memory reclamation scheme is one of few that is philosophically consistent with the design goals of the C++ programming language: one must not be penalized for features that are not used. This report includes performance measurements from a prototype implementation.

Introduction

Garbage collection (GC) is a programming environment feature that removes the programmer's responsibility for freeing dynamically allocated storage. It is a fundamental component of a Lisp or a Smalltalk-80 system. Garbage collection is also provided in some imperative object-oriented languages such as Eiffel [Mey88] and Modula-3 [CDG+88].

There is widespread belief in the C++ community that GC is useful and beneficial. While there are superficially valid reasons why GC should not be part of C++, overall, it is a desirable feature [EP90]. However, there is no consensus as to what kind of garbage collection is most appropriate. The alternatives include conservative collection [BW88], partially conservative collection [Bar88,Bar89], and non-conservative copying or mark-and-sweep collection [Ede90].

In this paper we examine the alternatives and present a copying collector. Section one explains why garbage collection in C++ is a difficult problem. Section two considers related work. Section three introduces a copying collector. Section four discusses the efficiency of

*This paper appears in Proc. of the 1991 Usenix C++ Conference, April 1991, Washington, DC, USA.