Dynamic Memory Allocation and Garbage Collection

Hans-J. Boehm, and Paul F. Dubois

Citation: Computers in Physics **9**, 297 (1995); doi: 10.1063/1.4823407
View online: https://doi.org/10.1063/1.4823407
View Table of Contents: https://aip.scitation.org/toc/cip/9/3
Published by the American Institute of Physics

DYNAMIC MEMORY ALLOCATION AND GARBAGE COLLECTION

Hans-J. Boehm

Department Editor: Paul F. Dubois
dubois1@llnl.gov

Many algorithms require dynamic allocation of memory while the program is running. For example, a text editor may choose to store text a line at a time. Typically, each line would be allocated as it is created or modified, so that it only consumes as much space as required, without imposing arbitrary restrictions on the length of lines. Or it may choose to allocate and store the file in even smaller contiguous pieces, to minimize the number of characters that need to be moved in memory after an update. Similarly, many numerical programs will need to allocate sections of memory whose size or number depends on the particular input to the program.

Probably one of the best arguments for the use of flexible data structures and dynamically allocated memory is made by all the programs that should have used them, but did not. Most of us have encountered something like the following dialogue on my Unix workstation:

```
siria% echo */*
Arguments too long.
```

Text editors are also frequent offenders. Many of them are incapable of editing text files I routinely use, either because of line-length limits or file-length limits. Compilers often fail to compile automatically generated programs because of arbitrary limitations on, for example, the number of cases in a C switch statement. Things sometimes get even worse if the offending code does not notice that it has run out of space:

```
ftp>get aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaa
Segmentation fault
```

The Morris Internet Worm that temporarily disabled much of the network in November 1988 relied heavily on a similar bug.[1]

Most programming languages provide facilities for allocating properly sized pieces of memory to avoid such problems. C provides a standard library function `malloc` for the purpose. C++ also provides `new`. Fortran 90 provides `ALLOCATE` statements, and Eiffel and Sather provide creation instructions and expressions, respectively. More "functional" languages such as ML and various dialects of Lisp, which rely less on updating data in place, generally provide many functions that return new results in freshly allocated memory.

These languages, and sometimes even libraries or implementations of the same language, differ significantly in the amount of help they expect from the programmer. Memory allocation in a language like Lisp or ML[2] is largely invisible. This is only slightly less true in Sather[3] and Eiffel.[4] Certain built-in functions allocate memory when they need it. This memory is reclaimed when the system can determine that it cannot possibly be used again. Normally this determination happens when there is no longer any way to refer to the allocated piece of memory, typically because the last location containing its address was overwritten, or because the last other piece of memory containing its address itself became inaccessible.

The process of reclaiming and reusing such useless memory is called "garbage collection." The routines that are occasionally called to perform this function are referred to as the "garbage collectors." Garbage collectors make it relatively easy to work with dynamically allocated memory and, in my experience at least, encourage more reasonably behaved application programs.

In contrast, implementations of languages like C, Pascal, Ada, and Fortran typically require the program allocating memory (the client) also to notify the system explicitly when the memory is no longer needed. In C, the `free` function is provided for this purpose. Fortran 90 provides the `DEALLOCATE` statement.

The presence or absence of a garbage collector can be as much a property of the implementation (the compiler and run-time support routines) as it is of the language itself.

Hans-J. Boehm is a member of the research staff in the Computer Science Laboratory at Xerox's Palo Alto Research Center. E-mail: boehm@parc.xerox.com

Languages like Lisp, ML, Sather, or Eiffel essentially assume automatic garbage collection, but even Lisp dialects without garbage collection have been developed for very special purposes (for example, Pre-Scheme in Ref. 5). We make available a storage allocator for nearly full standard C that includes a garbage collector.[6] At least one Fortran 90 implementation provides garbage collection.[7] The Ada standard explicitly provides for garbage-collecting implementations, though most implementations do not.

The following two sections explore the difference between automatic garbage collection and explicit deallocation a bit more and point out why the former is usually desirable. We then look at a possible implementation of dynamic storage allocation, and finally at a way to extend this storage allocator so that it automatically collects garbage.

We will assume that we are getting almost no help from the compiler and that all the work is being done by routines called to perform an allocation. This is not an optimal arrangement, but it might keep things clearer. And it illustrates that all this is possible with conventional languages and compilers. (We also assume that the compiler's optimizer does not do anything to confuse the collector. This is usually true in practice but not guaranteed.[8])

Some preliminaries

A memory-allocation function or statement such as C's `malloc` gives the client program a pointer to the newly allocated memory. Effectively this pointer, which can be used to access the memory, is just an address in the computer's memory. We will use the terms "pointer" and "address" interchangeably.

A simple data structure that makes use of dynamically allocated memory is a "linked list." A linked list consists of several memory blocks, with a program variable containing a pointer to (or the address of) the first block, and each block containing a pointer to the next one. The last block contains a special pointer value NIL to indicate that it is the last one. Each block also contains some other useful data. For example, a text editor might represent a file as a linked list, such that each block contains a pointer to the next line and one line of text. (This is not a very good representation choice, but it serves to illustrate the point.) Using an arrow from *a* to *b* to represent the address of *b* stored in *a*, a three-line file might be represented, as in Fig. 1.

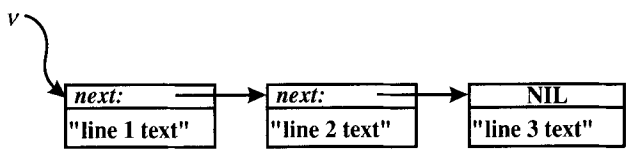

Figure 1. Linked list.

Adding an element to the front of the list involves (1) allocating a new list element *n*, (2) setting the pointer field inside *n* to point to what used to be the first element, that is, assigning *v* to the pointer part of *n*, and (3) assigning the address of *n* to *v*. Deleting the front element of the list is no harder. Both are fast operations whose execution time is independent of the length of the list.

The individual elements of the list may be fixed in size (for example, if the list represents a sparse vector of floating-point numbers) or variable in size (for example, if the list represents a sequence of lines in a text file, and the individual lines are stored as contiguous arrays of characters). Unlike a normal array, a linked list can grow to accommodate more elements until the machine runs out of memory. No arbitrary size limit is imposed.

Linked lists are a common ingredient of more complicated data structures used by applications programs. As we will see later, they are also an important data structure that is used within the memory allocator itself to keep track of unused pieces of memory.

An example

As a more interesting data structure, consider the problem of representing strings, that is, (mostly) finite sequences of characters. Assume for now that concatenation, that is, appending one string to another, is the most frequent operation and thus the one for which we want to optimize. The only other operation we are concerned with is writing the string to a file. (There are applications that need such simple strings. Reference 6 includes a much more complete string implementation and a simple text editor based on the same ideas.)

We will represent a string as a pointer to a block of memory. The memory blocks are of two types:

A type 1 memory block contains the number 1, followed by a contiguous sequence of characters. We will assume that the last character is special, or we have some other way of recognizing the end.

A type 2 memory block contains the number 2, followed by two pointers, representing the first and last parts of the string (which are in turn represented in the same way).

For example, the string "abcd" might be represented as in Fig. 2. (In reality, we would like the type 1 blocks to contain somewhat longer strings, to reduce memory overhead.)

The function to concatenate strings *x* and *y* now does the following:

1. Allocate enough memory for an integer and two pointers.

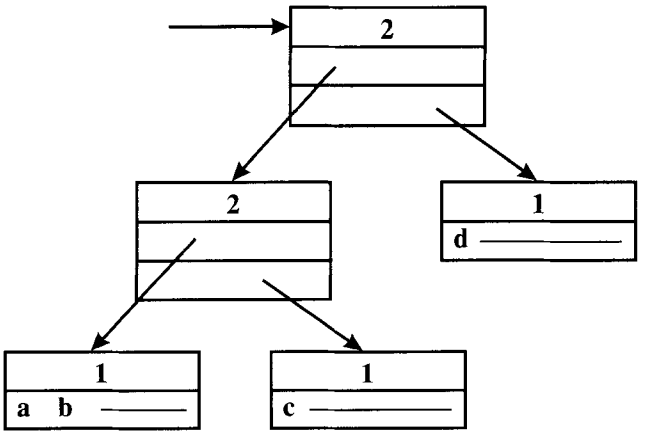

Figure 2. String "abcd."

2. Write the number 2 in the first slot.
3. Write x and y in the remaining slots.
4. Return a pointer to the newly allocated memory.

This has the very desirable property that the amount of time it takes is independent of the length of x and y. Concatenating two million-character strings does not take any longer than concatenating two single-character strings. Building an n-character string by repeatedly concatenating a character to either side takes time proportional to n.

Writing the string to a file is a bit more complicated but can be accomplished with a very short recursive routine. If each type 1 string representation is at least one character long, it still takes time proportional to at most the length of the string. We will just look at concatenations from now on.

When should a block of memory allocated to a string be deallocated? Assume program variable `X` points to the string "abcd," represented as in Fig. 2. We then assign another string, say "ef," to `X`.

Can we deallocate the "abcd" representation? The answer is sometimes "yes," sometimes "no," and sometimes "partially." To see an example of the last case, assume the actual sequence of statements executed is as follows (assume `typelrep` returns a pointer to a type 1 representation holding the given string, `concatenate` works as above, and "=" denotes assignment):

```
Y = concatenate(typelrep("a"), typelrep("b"));
X = concatenate(Y, typelrep("cd"));
(We're here)
X = typelrep("ef");
write Y to a file;
```

Now `Y` contains a pointer to part of the representation of "abcd," and this part will be used after the second assignment to `X`.

In situations like this, it becomes hard to deallocate memory explicitly. In reality, things are even worse, in that the assignments to `X` might be in a separate routine, and `X` might be one of its parameters. The author of this routine should not have to know which parts of the representation of the argument are shared with variables in other parts of the program.

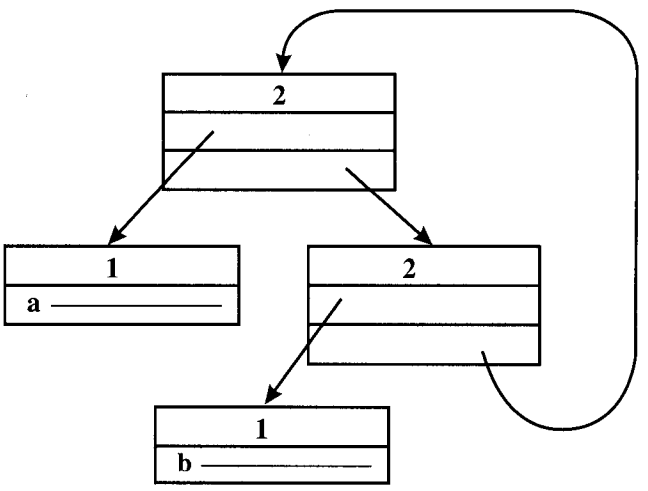

Figure 3. Infinate string "ababab...."

Furthermore, any mistake may result in the use of a block of memory by two completely different program parts simultaneously, leading to mysterious and nearly untraceable errors.

One interesting solution to this problem is to have each kind of string representation contain a count of the number of pointers that point to it. When the count reaches zero, that memory object may be deleted, and counts in objects it referenced can then also be decremented. This is a simple form of garbage collection known as "reference counting."

Reference counting has the advantage that it is sometimes easy to implement by the application programmer, especially in a language like C++. Unfortunately, this is an incomplete solution. Counter updates can be frequent enough to result in substantial processor-time overhead. Worse yet, not all inaccessible objects will have a zero reference count. For example, consider the infinite string "ababab...," which we can represent as in Fig. 3. Even if the entire string is unreferenced, each block of memory is still pointed to by another one.

A memory allocator

We will first consider memory allocation and deallocation. Although routines to perform these tasks are normally supplied by the compiler provider, it is instructive to look at how we might implement them.

We will assume that we are given a fixed region of memory H (for "heap") to allocate. We will also assume that the size of this region, expressed in number of memory words, is a power of 2.

If all allocated objects have the same size, everything is simple. We initially build a linked list of all possible memory objects. This means that we logically divide the memory into as many objects as will fit and then store the address of each object in the first word of the preceding one. We store the address of the first object in a variable `free_list`.

We can then allocate and deallocate as follows. At each stage if we start at `free_list` and follow the pointer stored in the first word, we will encounter all objects that are not currently allocated.

Allocate *(remove first element from* `free_list`*).* Replace `free_list` by the address in the object to which it points. Return the original address. (If `free_list` was originally NIL, notify the application program that there is no more memory.)

Deallocate *(add the unneeded object to* `free_list`*).* Replace `free_list` by the address of the unneeded object. Store the old value of `free_list` in the unneeded object.

Things become much more interesting when the user is allowed to request objects of arbitrary size. Many algorithms have been proposed for allocating such objects quickly, with minimal waste of space.[9]

Some allocators and garbage collectors contain provisions for relocating stored information that is currently in use, making it much easier to avoid wasting space. But this relocation is often impractical, because it requires that all pointers to moved objects be changed correspondingly. For now, we will assume that objects remain stationary.

Without moving objects, no matter what algorithms we use, we may be forced to waste a significant amount of space,

irrespective of our choice of algorithm. In particular, it may take about $N \log N$ of space to satisfy requests, even if no more than N bytes are ever in use at one time.[10]

Here we give a simple and reasonably fast algorithm for which the worst-case space consumption is similar to the theoretical limit. This amount of space consumption is usually acceptable in practice, although some other algorithms use less. This algorithm is essentially the same as the algorithm that is used in some versions of Berkeley Unix.

The basic idea will be to adapt our fixed-size allocation algorithm. To keep things manageable, we will only allocate objects with sizes that are a power of 2. If another size is requested, we will round it up to the next power of 2. (In the worst case, this will not quite double our memory requirements.)

We will keep a separate free list for each possible object size, that is, for each power of 2 up to our memory size. We now have an array of pointers F to these lists of free objects, indexed by the base-2 logarithm of the object size. Figure 4 depicts some of the data structure for a 256-word heap.

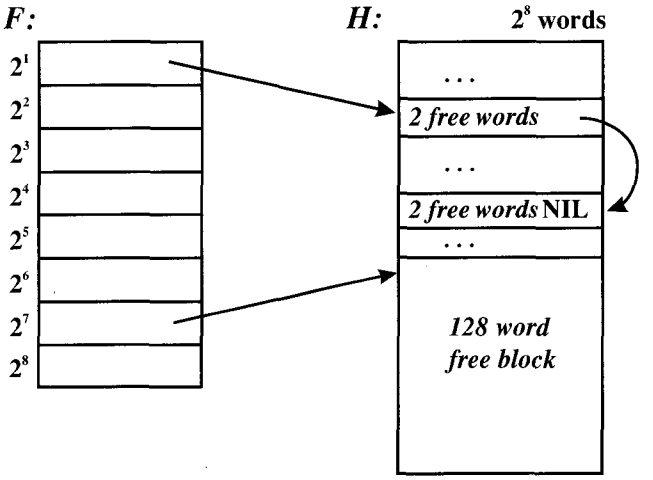

Figure 4. 256-word heap of free list.

Initially all entries but the last in F contain NIL, indicating that they are empty. The last entry contains the first address of H. The entire memory is one block on the last free list.

We will assume that we know the size of each object. This knowledge can be guaranteed by always allocating enough extra space, storing the size at the beginning of each object, and having the allocator hide this fact by returning the address just after the size field.

Allocation and deallocation now work as follows:

Allocate. Round the required object size up to the next power of two. Compute the logarithm l of the result. If F_l is not empty, remove the first entry and return it as the result. If F_l is empty, recursively use this procedure to allocate an object of size 2^{l+1}, return one half, and add the second half to the front of the list F_l. (If we try to allocate an object larger than the entire heap, we need to notify the client that we are out of memory.)

Deallocate. Round the required object size up to the next power of two. Compute the logarithm l of the result. Add the object to the free list F_l.

An obvious improvement to the space utilization of this algorithm would be to recombine adjacent free blocks of the same size, but we will not pursue that here.

We conclude with some interesting properties of this algorithm that we need later:

1. Deallocation time is constant, independent of object size. The same is true for allocation time, if we assume a sufficiently long-running program. (We leave it to the reader to formalize and prove that statement.)
2. An object of size 2^l has a starting address within H that is divisible by 2^l, or equivalently, the low-order l bits of its address within H are zero.

A garbage collector

We assume that in addition to the heap H from which we allocate memory, we have two additional arrays M and S. Each of these arrays contains one bit for each word in H. M_i and S_i denote the ith bit in M and S respectively, while H_i denotes the ith word. On a machine with 32-bit words, M and S together would occupy 1/17 of the required memory. We also assume again that the garbage collector has some way of determining the size of a memory object, given the address of the first word of the object.

Our garbage collector will use the traditional "Mark-Sweep" algorithm. As the name indicates, it operates in several phases:

0. Clear the entire M array. Replace all free list entries in F with NIL.

1. Mark all memory objects that can still be legitimately accessed by the program. The array M is used to mark objects as accessible. We will mark an object starting at heap word H_i by setting the bit M_i. To do this, we scan all program variables containing pointers. Whenever we find the address of an object that is not yet marked, we set the corresponding mark bit in the array M, and recursively scan that object, again following any pointers it contains.

2. Reclaim inaccessible memory by sweeping the heap. Scan the entire array heap H, one element at a time. (It is easy to find all the objects in the heap, because we know the size of each object and hence can find the start of the next one.) Whenever we find an object of size 2^l without the corresponding mark bit set, we add it to the front of the list pointed to by F_l.

The above algorithm is all that is required to implement a simple garbage collector under the following assumptions:

a. We know exactly where pointer variables are located and where pointers inside heap objects are located.

b. Objects can only be accessed if they can be reached from these variables by following a chain of pointers, each of which is the starting address of another object.

In practice, we can sometimes make both true with enough help from the compiler and, perhaps, a small slowdown of executable code. Traditional implementations of functional languages often use this approach. But we can weaken these assumptions to the following:

a′. We know where all pointer variables might possibly be located. That is, we know a set of locations that includes all pointer variables but might also include many other values.

b′. Objects can only be accessed if they can be reached by following a chain of pointers from a program variable, each

of which is an address somewhere inside another object.

These assumptions are sufficiently weak that they are usually satisfied by a typical C program compiled by a typical C compiler. (They must be satisfied if we slightly restrict C programs and C-compiler optimizations and slightly modify the allocator.)

To weaken assumption a to a′, we simply consider all possible locations of pointer variables and all possible positions of pointers in the heap. If the bits stored at that location happen to be an address inside the heap H, we treat the address as a pointer and mark the object it points to. If this treatment turns out to be a mistake, we will mark some extra objects and hence waste some memory. Somewhat surprisingly, this waste rarely turns out to be a problem, especially if the allocator and garbage collector take some additional precautions.[11,12]

To allow assumption b to be weakened to b′, we need to be able to find the beginning of an allocated object in the heap given an address of some word within the object. We use the array S for this purpose. The bit S_i will be one exactly when H_i is the first word of an allocated object, or the first word of an object on one of the free lists. It is easy for the allocator to maintain H correctly, and we will assume it does so.

Let us assume that we have an address a, and we want to find the starting address of the object containing a. We can do this by looking at S_a, S_{a-1}, S_{a-2}, ... until we find one that is set to one. In fact, by the last observation of the last section, we can do this much more efficiently by looking at only S_a, $S_{a\#1}$, $S_{a\#2}$, ..., where $a \# n$ has been used to denote a with the least significant n bits cleared [that is, $(a \setminus n)$ (n in C terminology]. We again leave the proof as an exercise to the reader.

Thus our revised marking procedure will look at a bit pattern stored in a location that might possibly contain a pointer and first determine whether it is an address somewhere inside H. If not, we ignore it. If so, we find the start s of the corresponding object by the method of the last paragraph, set M_s, and then recursively examine the object starting at s.

We can now modify the allocator of the preceding section such that it calls the garbage collector and retries whenever it would otherwise have run out of memory. The resulting allocator will function reasonably even if the client never explicitly deallocates any memory.

Refinements

The algorithms from the preceding sections are a little naive in several respects. Here we list the problems and briefly outline some possible solutions.

Fixed heap size. A real allocator would normally obtain more memory in large chunks as needed, by making calls to the operating system. Obtaining memory in this way complicates the allocator and collector in that the heap may no longer consist of one contiguous piece of memory. But it does not fundamentally change matters.

The client program cannot run during the sweep phase. Certain allocation requests take much longer than others, because they initiate a garbage collection. This time requirement can be annoying for some interactive programs, especially if the program is trying to maintain a smooth animation. The sweep of the entire heap to find unused blocks can easily be broken up into smaller chunks of work by only sweeping enough of the heap to satisfy the allocation request. Later allocator calls then complete the sweep as needed. There are also techniques that more cleverly avoid the sweep phase altogether, at the cost of some additional overhead.[13]

The client program cannot run during the mark phase. This is a more serious problem. It is not safe to simply run the mark phase of the collector in small bits at each allocation. The garbage collector could be confused by the client's changes while it is in the middle of marking, and it may reclaim accessible memory. Many known solutions to this problem exist, but they all involve informing the collector of changes to pointers stored in the heap, and not just of allocation requests. These solutions also require help from either the compiler or the operating system, or possibly the programmer.[14,15] The result is an incremental garbage collector.

This allocator is space-inefficient. The allocation strategy we described here is fast, and its worst-case space usage is relatively good. But under typical conditions, it uses more space than some competing algorithms.[17] Dedicating large chunks of memory for objects of a given size is usually a better strategy, making it unnecessary to round up allocation sizes to a power of 2 and usually making it less expensive to find the beginning of an object.

Nonpointers are too likely to look like pointers and cause memory retention. Though this is less of an issue than one might guess, it can still be important. There are many ways to reduce this effect. If the compiler can cheaply identify pointer locations, that information can be used. At a minimum, the collector should know about large heap objects that contain no pointers. These might be allocated with a different allocation function. The allocator should avoid allocating objects at addresses that are likely to be referenced by "false pointers"; it can easily identify such locations in earlier mark phases. These and some other techniques are discussed in Ref. 11.

The mark phase needs to look at almost the entire heap, perhaps causing excessive paging. It is possible to have most collections look only at objects allocated since the last collection, leaving mark bits set for those objects found reachable during the last collection. Empirically, recently allocated objects are much more likely to become inaccessible than "old" objects, and so this technique can still be a very effective and special case of "generational" garbage collection.[18]

The recursive procedure calls required for the mark phase potentially consume lots of stack space. We actually use a mark procedure with an explicit stack. Various tricks can be used to reduce the stack size, and recovering from a mark stack overflow by scanning the entire heap is also possible.

All of the above refinements (and a few others) are implemented in Ref. 6, which is available by anonymous FTP and is designed to be used with C or C++ programs.

Performance

I argued at the beginning that garbage collection is usually desirable and not terribly difficult to implement, at least in rudimentary form. So what does it cost?

The execution time associated with a garbage-collecting allocator is significantly different from that for an allocator requiring explicit deallocation. This conclusion is already

mostly apparent from our simple allocators.

With explicit deallocation, the time required for allocation and deallocation is independent of the size of the allocated object. (Of course, this statement is unlikely to be true for whatever else we subsequently do with the object.) In the garbage-collecting case, the fixed deallocation cost (to restore the object to the free list during the sweep phase) is likely to be appreciably less, for slightly subtle reasons. Objects are deallocated en masse, and so the addresses of data structures like F are likely already to be in machine registers, and we will probably save at least one procedure call per deallocation. In a concurrent system, we may also encounter substantially less synchronization overhead than with explicit deallocation.

On the other hand, each allocation increases the frequency of garbage collections, and we have to include garbage-collection time in the cost of allocation. The cost of the mark phase typically dominates, and so we will look at it. If the heap size is kept so that we collect whenever $3n$ bytes are allocated, and about $2n$ bytes are accessible at any point, then the marker must scan $2n$ bytes for every n bytes allocated, or 2 bytes for every byte allocated. Thus the real allocation cost is clearly proportional to the number of bytes allocated.

In our experience,[6] a garbage-collected allocator for C can often outperform the standard vendor-supplied allocators for very small object sizes (for example, 8–16 bytes). For larger objects, the performance will be a bit slower. But the extra overhead is comparable to that required to initialize the allocated memory to some known value.

The garbage-collected allocator will require some extra space in the heap to keep collection frequency tolerable. But non-garbage-collected programs that maintain interesting dynamic data structures typically also require space (and time) overhead to keep enough information to deallocate objects at the right time. A garbage-collected program probably does require slightly more space, but a direct comparison is difficult.

Garbage-collected programs may occasionally appear to pause during a garbage collection. (So may programs that explicitly deallocate large data structures at once, but typically for somewhat shorter periods.) On a typical workstation, this pause may become noticeable for an interactive application if much more than approximately 1 Mbyte needs to be scanned during a garbage collection. For normal interactive applications, the pause can be eliminated with the techniques of the previous section. For applications that require very predictable response, other techniques can be used,[13,18] at some cost in other overheads.

More possibilities

Many other storage-allocation algorithms have been developed.[9] Most of these make more efficient use of space under typical conditions, though some require more execution time for each allocation.

We have discussed only garbage-collection algorithms that do not move objects. If pointers are precisely identified, so that we can adjust pointers when objects are moved, other alternatives arise. It is possible to "slide" all objects to one end of the heap after a collection, thus greatly improving space utilization. (This action requires a different allocator, and makes incremental collection much more difficult.)

Many implementations of functional languages use "copying" garbage collectors.[19-21] In their pure form, these require at least twice as much memory as there are reachable objects. All objects are allocated in one half of memory. A very simple and elegant nonrecursive algorithm copies all live objects to the other half of memory, leaving unreachable objects behind. This algorithm interacts well with generational garbage collection, in that it is now possible to keep recently allocated objects in a separate area of memory, concentrating the garbage collector's attention on that region and resulting in garbage collectors that are much more efficient in some situations (for example, for some functional-language implementations).

An excellent survey of garbage-collection techniques is given in Ref. 22.

The collector implementation in Ref. 6 runs on many machines and operating systems, but it needs specialized code for each of them, both to identify possible pointer variable locations, and to aid the incremental collector. It is also somewhat vulnerable to aggressive compiler optimization,[8] though this vulnerability has not been a problem in practice. Several authors have proposed techniques for implementing garbage collection in portable C++, purely by using existing language facilities for redefining pointer operations.[16,23] Similar techniques have also been proposed for Ada. These techniques could potentially result in more-portable code that is immediately usable on many machines. However, they require some programmer discipline and require considerably more overhead, because the generated code has to maintain data structures to identify pointers to the garbage collector. The issues involved in providing garbage collection for C++ are discussed at length in Ref. 24.

Commercially supported garbage collectors for C and C++ are available from Geodesic Systems [`sales@geodesic.com` or (800) 360-8388] and Kevin Warne [`kevinw@direct.ca` and (604) 683-0977 or (800) 707-7171].

References

1. E. Spafford, "Crisis and Aftermath," Communications of the ACM **32** (6), 678–687 (1989), special section on the Internet Worm.
2. R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML* (MIT Press, Cambridge, MA, 1990).
3. S. Omohundro and D. Stoutamire, *The Sather 1.0 Specification,* `ftp://icsi.berkeley.edu/pub/sather/manual.ps`.
4. B. Meyer, *Eiffel: The Language* (Prentice Hall, Englewood Cliffs, NJ, 1991).
5. R. A. Kelsey and J. A. Rees, "A Tractable Scheme Implementation," Lisp and Symbolic Computation **7** (4), 315–335 (1994).
6. H. Boehm and A. Demers, *A garbage collector for C and C++*. Available from `ftp://parcftp.xerox.com/pub/gc/gc.tar.Z`, that is, by anonymous FTP from `parcftp.xerox.com`, in file `/pub/gc/gc.tar.Z`. An overview World-Wide Web page is available at URL `ftp://parcftp.xerox.com/pub/gc/gc.html`
7. *The Numerical Algorithms Group Fortran 90 compiler,* `http://www.nag.co.uk:70/`.

Feedback

I have often mentioned in discussion with physicists that one of the reasons I prefer Eiffel is that it is garbage-collected. This often results in the question, "What is garbage collection?" So I am pleased to present this month's article, which I hope you find interesting and informative. Using a garbage-collected language improved my approach to programming in a significant way. I hope the article at least helps you to understand what is going on underneath calls to `malloc` and `free`. (Fortran persons just getting into Fortran 90, read ALLOCATE and DEALLOCATE).

Please let me know (`dubois1@llnl.gov`) if you liked or did not like this kind of article. I have just started myself into the nightmare of Windows 3.1 and am learning about DLLs, etc. I would be interested in knowing what kind of articles on PC programming would interest people. (For Christmas I received a key chain made out of a defective Pentium and am trying to program it to display on my watch.)

In a previous Feedback I mentioned that information about my Basis project was now available on the Web. I submitted that copy knowing that it would not appear in print for at least two months, and I was being told that Basis would be up "Real Soon Now." Wrong. Never underestimate the delays of a bureaucracy. My apologies to any of you who tried and failed. Now, however, the Basis Home Page really exists at `http://www-phys.llnl.gov/X_Div/htdocs/basis.html`.

The source for the latest distribution is at `ftp-icf.llnl.gov` under `pub/basis`.

(If you go to the main LLNL page, `http://www.llnl.gov`, you can get to the Basis page under "Programs, Projects, Centers, and Consortia." You can always look at the cool Clementine pictures of the Moon while you are there.)

Paul F. Dubois

8. H. Boehm and D. Chase, "A Proposal for GC-Safe C Compilation," The Journal of C Language Translation **4** (2), 126–141 (1992). Also available from `ftp://parcftp.xerox.com/pub/gc/boecha.ps.Z`.
9. D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (Addison-Wesley, Reading, MA, 1973).
10. J. M. Robson, "An Estimate of the Storage Size Necessary for Dynamic Storage Allocation," Journal of the ACM **18** (3), 416–423 (1971).
11. H. Boehm, "Space Efficient Conservative Garbage Collection," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, SIGPLAN Notices **28** (6), 197–206 (1993).
12. Hans-J. Boehm and Mark Weiser, "Garbage collection in an uncooperative environment," Software Practice & Experience **18** (9) 807–820 (1988).
13. Henry G. Baker, "The Treadmill: Real-time Garbage Collection Without Motion Sickness," ACM SIGPLAN Notices **27** (3) 66–70 (1992); also see `ftp://ftp.netcom.com/pub/hb/hbaker/home-page.html`.
14. Andrew Appel, John R. Ellis, and Kai Li, "Real-time Concurrent Collection on Stock Multiprocessors," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices **23** (7), 11–20 (1988).
15. H. Boehm, A. Demers, and S. Shenker, "Mostly Parallel Garbage Collection," *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, SIGPLAN Notices **26** (6), 157–164 (1991).
16. D. Detlefs, "Garbage Collection and Run-Time Typing as a C++ Library," *Proceedings of the 1992 USENIX C++ Conference*, 1992.
17. D. Detlefs, A. Dosser, and B. Zorn, "Memory Allocation Costs in Large C and C++ Programs," University of Colorado, Boulder, Technical Report CU-CS-665-93. Also available from `ftp://cs.colorado.edu/pub/techreports/zorn/CU-CS-665-93.ps.Z`.
18. David M. Ungar, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SIGPLAN Notices **19** (5), 157–167 (1984).
19. Henry G. Baker, "List Processing in Real Time on a Serial Computer," Communications of the ACM **21** (4) 280–294 (1978).
20. C. J. Cheney, "A Nonrecursive List Compacting Algorithm," Communications of the ACM **13** (11), 677–678 (1970).
21. Joel F. Bartlett, "Compacting garbage collection with ambiguous roots," Lisp Pointers **1** (6), 3–12 (1988).
22. P. Wilson, "Uniprocessor Garbage Collection Techniques," *Proceedings of the International Workshop on Memory Management* (St. Malo, France, September 1992, Springer LNCS 637), pp. 1–42. An expanded version and related materials are available in `ftp://cs.utexas.edu/pub/garbage`.
23. D. Edelson, "A Mark-and-Sweep Collector for C++," *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, NM, January 1992), pp. 51–58.
24. J. Ellis and D. Detlefs, "Safe Efficient Garbage Collection for C++," Xerox PARC Technical Report CSL-93-4, September 1993. Also available from `ftp://parcftp.xerox.com/pub/ellis/gc/gc.ps`.