Why You Should Use Garbage Collection in Your Program

gene m. stover

created Sunday, 20 April 2003
updated Wednesday, 30 April 2003

Contents

1 Introduction

In an article I wrote for the *C/C++ Users Journal* ([Sto03b]), I described the basics of using the Boehm Collector in C & C++ programs, & I gave reasons that garbage collectors & destructor functions can coexist just fine – because the cases where they don't aren't very important.

In this essay, my second-to-last about garbage collection[1], I'll explain why you should use garbage collection, why the "performance penalty" isn't, & why some nifty ideas about variants on garbage collection aren't so nifty. Then I'll say no more about garbage collection. I've spent a lot of breath & keystrokes trying to convince people to use garbage collection, & if someone doesn't believe me now, he'll never understand (until garbage collection becomes the new vogue, at which time he'll ride the fad with proclamations of life-long devotion).

1.1 Terminology

One problem with garbage collection is its name. "Garbage collection" tells us we're cleaning up & throwing something out, but it doesn't tell us what. It also leaves to the imagination how much responsibility the cleaner (the garbage collector) has when the cleaning up analogy is transposed into programming fact.

A better name is automatic memory management. Not only is this name more descriptive, but it also reminds us of the limits of responsibility on a garbage collector. It manages memory, not objects, so it is responsible for honoring requests to allocate memory & for recycling memory when appropriate. It is not responsible for destroying objects or releasing other resources. This better name for the thing also reminds us that it's automatic, with the benefits & disadvantages that accompany automation: reduced cost in one area (programmer time) for an increased cost in another (run-time). The increased run-time isn't necessarily bad. After all, it would theoretically be possible to hand-compile programs to more efficient object code than a compiler can generate, so why don't people habitually hand-compile their programs?

2 Data Structures

Garbage collection allows you to use more complex data structures without writing extra code. Here's an example.

Let's say I want to make parallel, growing lists, & periodically I want to chop-off a list & reclaim the memory it occupied. (*Why?* Hush, it's an example.) Figure 1 shows pseudo-code to do it. This example is essentially the same as the performance test I showed in [Sto03a].

Notice the comment that says "Must reclaim memory here". In this example of singly linked lists, I suspect almost any programmer can see that looping over

[1] My last will describe a C++ allocator class template implemented for the Boehm collector.

```
; A node in a list
struct node { struct node *next; ... };

; An array to hold all the lists
struct node a[100];

; Return true 1/10th of the time.
boolean
one_in_ten () { ... };

while not end-of-program {
    for i = 0; i < 100; ++i {
        if one_in_ten () {
           ; Chop-off the list
           ; Must reclaim memory here!
           a[i].next = NULL;
        } else {
           ; Extend the list.
           struct node *tmp = malloc (sizeof *tmp);
           tmp->next = a[i].next;
           a[i].next = tmp;
        }
    }
}
```

Figure 1: Parallel, growing lists

the items in the list & calling `free` on each would get the job done just fine. Garbage collection isn't needed for this.

Watch what happens when the data structure gets just a little more complex. Let's say that besides extending & chopping a list, we can replace it with the list from another element of the array. Figure 2 shows pseudo-code for this new data structure.

Notice the two comments that say "Reclaim memory here, but be careful". The reason you must be careful when you reclaim a chopped list or a list that is chopped because of crossover is that a node in a chopped list could be in another list, too. You can't just free every node in a chopped list because a node might be in another list, too. That could lead to dangling pointers **and** multiple deallocations in the same program. (Just the kind of bugs that make me pull out my hair while I waste an entire weekend tracking them down.)

To free a chopped list from Figure 2, you'll need to implement some form of garbage collection. Reference counting would work just fine. If you program in C++, you could use smart pointers that know about reference counting. You might implement your own smart pointer library. It should take just a few days to debug it & get it right. Be sure to tell your project manager about it so she can pad the already tight schedule.

C programmers don't have smart pointers, so they might have to build the reference counting into `struct node`. Or maybe there's a clever solution with macros.

But wait, it gets better. Notice that there are no cycles in the lists. What happens if we extend the "list crossover" program so that we can append a pre-existing list to another one. Here is the pseudo-code for the new example:[2]

```
; A node in a list
struct node { struct node *next; ... };

; An array to hold all the lists
struct node a[100];

; Return true 1/10th of the time.
boolean
one_in_ten () { ... };

; Find the last node in a[i], then make
; its 'next' point to the same element
; some other a[j].next points to.  In
; other words, it appends the a[j] list
; to the a[i] list.
; When finding the last item in a[i],
; must take care not to loop forever
; in case a[i] is circular.
```

[2]I would have made it a figure, like the previous two examples, but it is too big for a page of paper.

```
; A node in a list
struct node { struct node *next; ... };

; An array to hold all the lists
struct node a[100];

; Return true 1/10th of the time.
boolean
one_in_ten () { ... };

while not end-of-program {
    for i = 0; i < 100; ++i {
        if one_in_ten () {
           ; Chop-off the list
           ; Reclaim memory here, but be careful.
           a[i].next = NULL;
        } else if one_in_ten () {
           ; Crossover.  Drop this list & replace
           ; it with the one from another element
           ; of the array.

           ; Reclaim memory here, but be careful.
           a[i].next = NULL;

           ; Select another element from 'a',
           ; make a[i] point to its list, too.
           int j = rand () % 100;
           a[i].next = a[j].next;
        } else {
           ; Extend the list.
           struct node *tmp = malloc (sizeof *tmp);
           tmp->next = a[i].next;
           a[i].next = tmp;
        }
    }
}
```

Figure 2: Parallel, growing lists that can cross-over. Requires garbage collection.

```
void
append_random (int i) { ... }

while not end-of-program {
    for i = 0; i < 100; ++i {
        if one_in_ten () {
           ; Chop-off the list
           ; Reclaim memory here, but be careful.
           a[i].next = NULL;
        } else if one_in_ten () {
           ; Crossover.  Drop this list & replace
           ; it with the one from another element
           ; of the array.

           ; Reclaim memory here, but be careful.
           a[i].next = NULL;

           ; Select another element from 'a',
           ; make a[i] point to its list, too.
           int j = rand () % 100;
           a[i].next = a[j].next;
        } else if one_in_ten () {
           ; Append a randomly selected list
           ; from the array to a[i].
           append_random (i);
        } else {
           ; Extend the list.
           struct node *tmp = malloc (sizeof *tmp);
           tmp->next = a[i].next;
           a[i].next = tmp;
        }
    }
}
```

In this new program, there can be circular lists. Maybe more complex, more general structures can happen. I don't know. In any case, reclaiming the memory occupied by a chopped list got a lot more difficult. Reference counting won't hack it now. Do you know an algorithm that would? How long would it take to find one by research? How complex will that algorithm be? How long will it take to implement? Will it be fast enough? How will it interact with other parts of a program? Such an algorithm is effectively a general purpose garbage collector, so why not use a pre-existing, general-purpose garbage collection library?

My point is that a lot of classes of data structures which were infeasible with manual memory management suddenly become accessible when you use a garbage collector.

The three examples in this section are from an abstract programming world, not from an "I've actually needed to do that" practical programming world. I wanted to show how simple changes to a simple program can lead to data structures that are infeasible to use without garbage collection.

Now for some more practical examples. They're still hypothetical, but they are more practical.

3 Practical Examples

3.1 Object Database

Think about an object-oriented database of all the people & assets in my company. It contains employees, computers, chairs, desks, offices to contain them, the projects that occupy them, schedules, pay rates, benefits plans, times when employees will be on vacation, & who knows what else.

Right here, there is a need for garbage collection. Given an employee, I probably want to know what benefit plan he chose, which implies links from employees to their benefits plans. However, given a benefit plan, I might want to know which employees chose it, which implies links from benefit plans to employees. So right there, the database has memory management concerns; I can't just delete an employee & expect things to work. Since the circularity is shallow (employee to benefits, with no intermediate objects), I could write a special "delete an employee" function, but how about deleting a benefit plan?

What about networks of circularly linked objects that are not as apparent? What if, for each employee, my database keeps an object for all that employee's relatives. (Maybe I use it to suggest benefits plans or to estimate the expense for an employee to move to another state (from which I could infer the likelihood that he'll quit if I don't give him a raise), or what if I'm just a Big Brother bastard who wants to keep all the information I can on all my employees.) Now, let's say that I hire an employee's child. Now we have a multiple-reference issue because one person record (the employee's child) is the target of links from the employee & from my database's set of employees. What if the child becomes the employee's boss or if they work for different departments which communicate, & I record inter-departmental communication paths in the database? Now I have circular references.

What if there are more devious, less easily recognizable cycles in the data? Do I pay database architects to find them all & write special code for them? How much code will that require? Too much for the database? What if they miss a type of cycle? What if they make another mistake (which we all do)? If the database crashes, what happens to my company?

The risk can be greatly reduced, maybe eliminated, by using garbage collection.

3.2 Graph Problems

Some problems from computing theory are often expressed as graph problems. A whole lot of problems from NP-completeness are like this. Sure, there are techniques for flattening the data structures of many of these problems into arrays that don't require much memory management effort, but what if the flattening technique obscures the real solution algorithm? Wouldn't it be easier to rely on a garbage collector to manage the memory so you could manipulate data that is most directly & simply related to the problem?

Many programmers believe that these types of problems are purely in the realm of academic theory & not useful to the working programmer. I know those programmers are wrong. Solutions for these types of problems can be useful to the working programmer on a daily basis. One reason working programmers don't recognize it might be that the tricks for flattening the data obscure the algorithm & its general usefulness.[3]

4 Interfaces

Another advantage of garbage collection is that it simplifies your interfaces. C & C++ programmers are intimately & sometimes painfully familiar with the issue of "ownership". That is, we often have to define rules for which container owns some dynamically allocated objects & will be responsible for deleting them. Sometimes, it's just a case of declaring who the owner is, but at other times, it's a whole lot more difficult.

Garbage collection gets rid of that in two ways. First, you don't need to create rules about ownership. The garbage collector will recycle the memory when appropriate. With garbage collection, it becomes easier to allow an object to exist in multiple containers or to be "known" by other objects. In fact, the issue of containment shifts slightly. It's no longer a case of which collection contains some object; it's an issue of which collections can be used to navigate to the object. I suspect thereis a performance improvement because you don't need to copy objects as often.

Garbage collection also simplifies function calls. If you want to allow another object to access this object's state, just return a pointer to the collection. You don't need to make a copy of the collection, so you might get a performance improvement. You don't need to define cumbersome interfaces that copy all the objects into a collection specified by the caller. Your interfaces become simpler & more efficient.

I wish I could think of an example because I'm sure that most people who have not used a garbage collector have difficulty imagining these benefits I'm describing. I can't think of a good example. Hopefully some people will try it for themselves. I realized on my own that garbage collection simplifies interfaces, &

[3] Another explanation is ignorance. Most programmers appear to have slept through their classes.

one other programmer I know who advanced to using garbage collection made the same observation independently.

5 Performance

Garbage collection is slower than static allocation or stack-based alocation. That's a fact, but it's *not* the problem people assume it is.

Let's say that some program uses a data structure that can be statically allocated, no run-time allocations or deallocations at all. Then you didn't need a garbage collector at all. You didn't need dynamic allocation at all. Good for you.

A program with slightly more complex data structures might be able to use stack allocation, which is a primitive form of dynamic allocation. Again, a garbage collector wasn't needed at all, & again, congratulations on writing such an efficient program.

What about a program that requires dynamically allocated data, but you can determine at programming time, with statically placed `free` function calls, when to deallocate memory? Again, you don't need a garbage collector at all.[4]

All of these have been cases in which the programmer was able to statically determine the times at which to deallocate memory. In the first case, he did that by not allocating dynamic memory at all. In the second case, he let the compiler figure it out. In the third case, he wrote explicit code. In all cases, the programmer "pre-computed" the memory management strategy. In any case where you need to maximize the work done at run-time[5], you want to pre-compute whatever you can. So the programmer should use these types of data structures whenever possible.

What if the program requires – absolutely requires – multiple (but not circular) references to objects? That program must do some kind of run-time memory management. Reference counting will be just fine. Reference counting is surprisingly, amazingly fast. It just decrements a counter & deallocates a block if the counter reaches zero. Implement it yourself (if you have the time & the budget), or use a garbage collection library that implements reference counting instead of a more general algorithm.

What if we have a program that relies on data structures that can contain general graphs.[6] Now imagine that program running without a garbage collector. Its programmers wrote special memory-management functions to handle the general graph of memory allocations. What memory management algorithms did they code? They coded exactly the kind of algorithm a general purpose garbage collector uses because those are the only algorithms that can manage memory allocated in such a general, complex fashion. he nature of

[4] I would argue that this is the same as the stack allocation case, but the compiler requires that the amount of memory to allocate from the stack is known at compile-time, so you have to use dynamic allocation.

[5] or to minimize the hardware required to do it

[6] A general graph can contain cycles.

manual time	automatic time	iterations	program	comment
0.0	0.0	0	demo0000.c	static allocation
0.0	0.9	56173	demo0001.c	individual blocks
0.9	16.3	491	demo0002.c	individual lists
0.3	15.1	56173	demo0003.c	blocks in random order

Figure 3: Number of seconds for each memory management technique, for each program

managing memory that can be in a general graph is the same whether you write custom functions to do it or you use a general-purpose garbage collector. It is inescapable. It's not the garbage collector's fault. If it's not fast enough, the problem is that someone chose a data structure that is inappropriate for the requirements.

My point with all these examples is that the question "Is garbage collection fast enough for me" is inappropriate. The question should be "Have I chosen appropriate data structures for my program". Different data structures require different amounts of memory-management work at run-time. If you have more than a trivial amount of memory management work at run-time, you should use a garbage collector because you already are, whether you use a pre-existing one or integrate your own into your program.

6 Performance Measurements

Here are the results from some performance measurements I did. A lot of people will see that garbage collection is slower than manual memory management & immediately conclude that garbage collection sucks, but a lot of people draw conclusions without due contemplation. Please be sure you understand the real question to ask[7] before dismissing garbage collection on the basis of these measurements.

First, the measurements themselves, then a discussion of how they were obtained, their interpretations, & their significances. Each test program performs some algorithm that allocates & releases memory. It performs the algorithm twice; once with manual memory management & once with the Boehm collector. Figure 3 shows the number of seconds each program required. Each program is a line in the table. The first column is the number of seconds that program's algorithm required using manual memory management. The second column shows seconds for garbage collection. Figure 4 is like Figure 3 except that it shows the number of iterations per second for each algorithm.

The *bzip2*ed, *cpio*ed archive of the programs is `wgcperf.cpio.bz2`. To extract it, run "`bzcat wgcperf.cpio.bz2 |cpio -i`". The individual source files are:

[7] That question is performance. See Section 5.

manual rate	automatic rate	program	comment
0.0e+00	0.0e+00	demo0000.c	static allocation
2.8e+06	6.5e+04	demo0001.c	individual blocks
5.5e+02	3.0e+01	demo0002.c	individual lists
1.9e+05	3.7e+03	demo0003.c	blocks in random order

Figure 4: Iterations per second for each memory managment technique, for each program

- wgcperf/COPYING[8]
- wgcperf/Makefile[9]
- wgcperf/demo0000.c[10]
- wgcperf/demo0001.c[11]
- wgcperf/demo0002.c[12]
- wgcperf/demo0003.c[13]
- wgcperf/make-report[14]
- wgcperf/this.c[15]
- wgcperf/this.h[16]

6.1 Interpretation & Significance

In the most extreme of those demo programs (`demo0003.c`), manual memory management is about 50 times faster than automatic memory management (garbage collection). Sounds horrible for garbage collection, but what is the significance of that?

`demo0003.c` iterated through its algorithm 56,173 times. Using manual memory management, the average iteration required $\frac{0.3 second}{56173} \Rightarrow 5.3 \times 10^{-6} second$. That's 5.3 microseconds for the mean iteration. Garbage collection is a woeful fifty times slower, requiring $\frac{15.1 second}{56{,}173 iteration} \Rightarrow 26.88 \times 10^{-6} \frac{second}{iteration}$.

Think about an interative application with a graphical user interface. If you use manual memory management, the average operation could take $X second + 5.3 microsecond$. If you use garbage collection, the operation would require $X second + 26.88 microsecond$. Correct me if I'm wrong, but isn't $\frac{1}{3720}^{th} second$

[8] `COPYING`
[9] `Makefile`
[10] `demo0000.c`
[11] `demo0001.c`
[12] `demo0002.c`
[13] `demo0003.c`
[14] `make-report`
[15] `this.c`
[16] `this.h`

imperceptible to humans? And as X increases from about zero (what it is for `demo0003.c`) to a few seconds (for a database query), the difference between manual memory management's 5.3 microseconds & garbage collection's 26.88 microseconds vanishes.

Those calculations assume that the time required for garbage collection sweeps is distributed evenly across all iterations of the algorithm. In fact, the garbage collector runs a sweep only periodically. So most allocations happen in a very short amount of time; only some are lengthened by a garbage collection sweep. How long is one of those unfortunate allocations?

Maybe the Boehm collector has an interface that allows programs to monitor individual garbage collection sweeps, but I did not use it. By watching the progress lines of `demo0003.c`, it looks like there were about five garbage collection sweeps. If the time required for all the garbage collection sweeps is distributed evenly among those five allocations (which is not the same as being distributed evenly among all allocations), & if the total running time of the entire program is 15.1 seconds, then the average garbage collection sweep requires less than $\frac{15.1 second}{5 sweep} \Rightarrow 3.02 \frac{second}{sweep}$. I'll grant it that a 3 second pause after I press a button on a GUI could be unsettling to a user, but such a collection sweep doesn't happen very often. In `demo0003.c`, there are about $\frac{56{,}173 iteration}{5 sweep} \Rightarrow 11,234$ iterations between sweeps. Is a user likely to run an interactive application that long? What's more, if the garbage collector allows a program to monitor individual collection sweeps, the interactive application could display some type of "hang on a moment (less than five seconds)" message in a pop-up window during a collection sweep. If such a message happened about every 11,234 times a user clicked on a button, I don't think the user would complain much. If the interactive function required a few seconds, anyway (such as a database query or generating a report), the garbage collection sweep would merely double the time required for the function – every $11,234^{th}$ time.

Also notice that all the demonstration programs use data structures that are appropriate for manual memory management, so of course manual memory management is both faster & easily implemented. What if a program requires more complex data structures? If the programmer insists on manual memory management, either those data structures must be abandoned, or the programmer must use a garbage collector, whether he uses a general-purpose, optimized, debugged one or he writes & debugs his own that is built into the application. That brings me back to two earlier points: Garbage collection makes complex data structures feasible (Section 2), & it's not a question of whether garbage collection is fast enough (Section 5).

7 Interactive Time

"Interactive time" is my own term for requirements that are neither batch nor real-time. In interactive time, you have a user (human or otherwise) who might get upset if your program takes too long to do something. It's not like batch time (where no one cares how fast any particular step is as long as the whole thing

gets done in time – overnight or in less than a second or behind the scenes), & it's not like real-time in which the requirements state acceptable response time ranges. In interactive time, your program needs to be fast so as not to annoy the user.[17]

Programs with graphical user interfaces (GUIs) are examples of interactive programs. Some GUI programmers tell me that garbage collection isn't appropriate for their program because the user will get upset if they have to wait after they click on a button.

Let's imagine the innards of a GUI app. Maybe the user can enter some things in some text fields & click a "Do It" button. The app fetches some things from a database, does some calculation with them, & displays some results in a window.

How much memory could be allocated & then forgotten each time the user clicks the "Do It" button? All the GUI widgets, including the text fields, have life-spans longer than the operation; they last for the entire duration of the program, so the "Do It" operation doesn't generate free-able memory from them. The connection to the database probably doesn't, either, but let's err on the conservative side by assuming each query generates 1 kilobyte of free-able memory from overhead.

The results of the query will be free-able when we're done computing something from them. A big query would be about 1 megabyte, though in most cases, it's dumb to request that much data from a database.

The calculation on the query results probably generate less data than the query results themselves. Let's call that $\frac{1}{2}$ megabyte.

The plotting operation probably generates very little free-able memory because it's stuffing data into a GUI object.

By summing all that, it looks like one "Do It" generates about $1\frac{1}{2}$ megabytes of free-able memory.

A modern desktop computer could easily have hundreds of megabytes of real memory. The 2-year-old laptop I'm typing on now has 128 megabytes of real memory. The modern computer also has virtual memory, but let's ignore that. Let's also assume the user is on an old clunker with a measly 64 megabytes & that half of that is already occupied by programs (ours & others). That leaves 32 megabytes of memory for data.

If each "Do It" consumes $\frac{3}{2}$ megabytes of that 32, then the garbage collector will need to run at least every 21^{st} time. If the data doesn't have many pathologically chaotic cycles in it, my own experience shows that the garbage collector can run in about 1 or 2 seconds (again on my dinky old laptop). The garbage collector might make a pass more frequently, but if so, then each pass will take less time.

So for every 21^{st} query, your program takes up to 2 seconds longer. "Unacceptable" scream many programmers, but how long did your query take in

[17] People think real-time means "fast", but people are ignorant. Consider the program which fires the deceleration rockets for a ship traveling from the Earth to the Moon. From launch time, that program has days(!) before it needs to fire those rockets, but when it's time to fire those rockets, it has a narrow window in which to do it or people die. That's real-time.

the first place? The program fetched a megabyte of data from the database for god's sake. That takes about 2 seconds to transmit over an Ethernet LAN. There must have been processing time in the database (unless you requested a whole table, which would be dumb); let's be very kind to the database & harsh on the garbage collector by calling it 1 second. The analysis of the data must have taken time; call it 1 second, also. I don't know much about graphics, but let's say the picture could be drawn in 1 second, too.

So the total time for the "Do It" operation, not counting garbage collection, is $2+1+1+1 \Rightarrow 5$ seconds. If the garbage collector runs for 2 seconds every 21^{st} time, then garbage collection increases the mean time for "Do It" by a whopping 0.095 seconds to 5.095 seconds. Worst-case time is 7 seconds; compared to 5 seconds, the cost is probably in the "eh, big deal" range.

What's more, the example was contrived on the side of pessimism. Every database interface library I've used allows me to fetch a chunk of records at a time & to re-use the same block of memory for each chunk. Voila! I just reduced the frequency that the garbage collector runs & also the mean response time. If I can treat the chunk-holding block of memory as flat (instead of a bunch of individually allocated objects), each garbage collection pass will be shorter, possibly becoming unnoticeable.[18] If I can allocate the chunk buffer at the beginning of the program & use the same one for the life of the program, the garbage collector will almost never run.

Considering that the garbage collector reduced my development time, there is no reason not to use a garbage collector for this type of program.

Another example of programs that run in interactive time are text editors, those programs we programmers know almost as well as our spouses. Gnu *emacs* runs on Lisp. Yep, it's a bunch of Lisp functions that run on a custom version of Lisp. It uses garbage collection. How often do you sit idly at your terminal while you wait for emacs to free memory? You notice a hiccup once in a while, less than a second, but even that is infrequent. Could emacs run noticeably faster if Richard Stalman were to rip-out the garbage collector & hard-code explicit memory deallocation function calls? I highly doubt it. And doesn't it run fast enough, anyway? That's interactive time, & garbage collection is fine for it.

8 Custom Garbage Collectors

Maybe a garbage collector could allow a program to tell it when not to run a sweep. Before a program entered a time-critical section, it might allocate whatever memory it needed for that section & then notify the garbage collector not to run until the program exited the section. You can accomplish the same thing with a more general garbage collector by allocating the memory before the

[18] My own experience is that the time required for a garbage collection pass increases as the number of allocated objects increases. If I've allocated ten objects, it takes the garbage collector a certain (unnoticeable) amount of time to examine them, regardless of their size, but if I have allocated 100,000 objects, it takes more time.

time-critical section & then being careful not to allocate memory until exiting the time-critical section. Isn't that a good technique for any time-critical section, garbage collection or not? As with choosing appropriate data structures, thoughts about alternatives to garbage collection have led full-circle to "If you write your program intelligently, garbage collection is purely beneficial".

Some programs might have specific times at which a garbage collection pass would go unnoticed: a program waiting for user input, or a service that has recently ended a client session. A garbage collector for that type of program might allow the program to cause a garbage collection pass to reduce the likelihood that the next allocation will pause for a collection pass. (Remember that most pauses are only a second or two on a modern computer. Probably not a problem.)

A garbage collection algorithm might be able to exploit memory usage patterns of some types of programs. Maybe old memory blocks rarely become collectable, so the garbage collector doesn't scan them every time. (This is the essence of generational garbage collection algorithms.) Maybe another garbage collector could be optimised for programs that release memory in a queue fashion: first allocated is generally first freed.

A garbage collector might do its collection work in a low-priority thread in the hope that memory management time goes unnoticed, interleaved with time spent on other computation. Java's garbage collector does this (most implementations of it, anyway). Another garbage collector might make a pass through memory only when absolutely necessary to minimize total time spent in garbage collection even though each pass through memory could require more time.

A garbage collector might allow the application program to provide hints about individual memory blocks. Maybe a program could tell the garbage collector that a memory block does not contain pointers to other memory or that there are no pointers to bytes within the block, only & always to the beginning of the block. The Boehm collector does both of these. A garbage collector might allow hints about when to free a block. I'm concerned that hint mechanisms would deteriorate in the hands of well-meaning programmers to an abundance of gratuitous, maybe erroneous, function calls into the garbage collection library, damaging its performance with a flood of data about each memory block & with laughably frequent & ironic "run the collector now" requests.

Also, garbage collectors can employ optimization techniques such as caching blocks of a certain size. Profiling programs could study an application to determine parameters that tune the garbage collector for the application.

In the presence of languages that are interpreted or that run on virtual machines, garbage collectors could use techniques that aren't available to the memory managers of C & C++ programs. One such technique is garbage collection by copying. I haven't used it, but it's simple, & I've heard it works well (though it doubles the amount of memory your program requires).

At this time, there aren't many (any?) specialized garbage collection libraries. That's because there isn't much demand. Maybe it's also because people think that garbage collection must be integrated with a language. I'm con-

vinced that if programmers learned to use garbage collection by default they'd see that it is great even for these special cases. Currently we have a chicken-&-egg problem. Because the existing garbage collection solutions are limited, programmers think that any special case precludes garbage collection. (And every programmer seems to think his is a special case.)

9 Dead Ends

Here are some ideas about garbage collection that I have considered at one time or another before concluding they were ill-conceived. Maybe this section is a log of my own well-meaning foolishness.

9.1 Static Garbage Collection

What if you had a program that studied your application & told you where to insert calls to free memory (or delete objects, whatever)? It's probably technically impossible at this time (the year 2003), but let's forget about that & imagine that one exists.

You'd feed your program to this program. It'd analyze your program, probably while running it like you would run your program in a profiler. Then the analysis program would insert `free` calls or `delete` statements so that your program had no memory leaks, no dangling pointers, & no multiple deallocations. (Yes, yes, a tall order, but for now, just pretend.)

What kinds of results would it produce?

It'd be great for lexically structured deallocations. By "lexically structured", I mean those very common cases in which you allocate a block, do something with it, & then free it. In C & C++, all variables on the stack are lexically structured allocations, but they are handled automatically by the compiler. I'm talking about those cases that are conceptually like variables on the stack, but because the compiler demands that the sizes of items on the stack be known at run-time, you must call `malloc` & `free` explicitly. These cases can be difficult to detect if you had to separate the allocation & deallocation into different functions.[19] Wouldn't it be cool if the mythical memory analysis program detected these cases & inserted the `free` or the `delete` where appropriate?

In fact, there's no need for this. In C++, you can put an object on the stack that is in charge of allocating the memory when it's constructed & for deallocating the memory when it's destroyed. You don't even need to write a new class for this purpose; just use class template `std::auto_ptr`. In many implementations of C, there is a little-known function called `alloc`, which is like `malloc` except that it allocates memory on the stack, & the memory doesn't need an explicit clean-up.

So the mythical analysis program doesn't need to exist to find block-structured allocation patterns. How about more general usage patterns?

[19] They should be in the same function whenever possible – which is nearly every time, but people don't always do that.

I suspect that if a block of memory is not used in a way that can be determined to be lexically structured at compile-time, then you need a run-time system to determine when to deallocate it. I can't prove that, but I suspect it's true. Whether you can do a quick reference-count check or need a more general garbage collection pass, you have to do some amount of run-time computation. That means that the mythical static analyzer would need to insert conditional code at strategic points. Maybe the code could be a quick check of the reference count followed by a deallocation call if the reference count is zero, or maybe the code would need to be a call to a full-blown mark-&-sweep algorithm. In any case, the so-called static deallocation idea has become statically & strategically placed calls to run the garbage collector.

Where would most such calls go? The time you need to run the garbage collector is right before you allocate memory. The garbage collector probably does this anyway, whenever you call its function that allocates memory. In that function, the garbage collector almost certainly checks to see if it can allocate the memory. If it can't, it does a collection pass. Most of the calls the mythical static analyzer inserted would be right before an allocation request. Heck, it might want to do that in front of every allocation request, but the garbage collector does it already. So we lose another need for the mythical static analyzer.

Maybe the mythical static analyzer could look for places in the program to insert strategic suggestions for the garbage collector to run so that, when the program tried to allocate memory for real, a delay would be unlikely. There might be a use for this, but I'll bet anyone that in most cases, the performance improvement will be negligible. Also, if the program has situations in which it needs memory but cannot afford a garbage collection sweep at those times, the programmer can & should pre-allocate memory so that the program doesn't need to call the general purpose memory allocator at these critical times. This applies whether or not the program uses garbage collection; it's good programming practice.

I still think a profiler could analyze memory use patterns & produce parameters to tune a garbage collector to the application, but I don't think there's any point to a program that inserted calls to run the garbage collector.

9.2 Smart Pointers

In C++, you can implement a garbage collection library that uses smart pointers instead of a library that operates "under" `new` & `delete`. In fact, I've done it in a library called Giggle ([Sto00]).

That library works fine. The main reason for it was to have a garbage collection library that allowed me to plug-in different garbage collection algorithms at run-time so I could compare their performances. Giggle does that very well, but what I learned after I had finished & used it is that programming with smart pointers is an error-prone pain in the ass because, no matter how smart your pointer, you have to get a nude pointer sooner or later so you can do something with it. A form of pointer leakage occurs. What's more, either you extract a nude pointer at nearly every function call, or you customize your functions to

allow smart pointers. If the only concern were syntax, function templates would work in most cases, but many compilers don't support method templates. And do you really want every damned function in your program to be a template?

Basically, smart pointers are clunky, at least when they are applied to garbage collection. They aren't necessary, anyway, because the Boehm collector demonstrates that a garbage collector can operate "under" the level of a raw memory allocator.

A less biased & far more thorough discussion of the problems with smart pointers is [Ede92].

9.3 The Ultimate Lazy Garbage Collector

One of the many algorithms I implemented for Giggle was the ultimate lazy garbage collector. It didn't collect anything at all until the end of the program. Sounds silly at first, but what if you know the maximum amount of memory your program will use, & you know it'll fit into memory? If that's your case, why have the cost of a garbage collection sweep (or even a `free` or `delete`) at run-time? Save it all until the program ends.

I also expected that an ultimate lazy collector might make some optimizations when it came to calling the destructors on all the objects.

It's an interesting idea, but what happens is that the program runs like blazes & then locks-up for a while as it exits because the garbage collector has to free everything then. The computer isn't dead, even the program isn't dead; it'll exit eventually. Why go to the trouble, though, when the operating system would have freed all memory in one, super-fast operation as soon as the program existed, anyway? The only reason for such an ultimate lazy allocator was to call destructors on all objects, but look at the cost.

The ultimate lazy constructor helped me realize two things about garbage collectors. The first was why most garbage collectors just exit when the program exists; they don't bother to find & free memory at that time. The second was that destructors & garbage collection are separate issues. In fact, "garbage collection" is a bad, bad term. The term should be "automatic memory management".

10 Conclusion

Garbage collection is of huge benefit & negligible cost to programmers everywhere – except C & C++ programmers. While programmers who use other languages wouldn't consider living without it, C & C++ programmers consider garbage collection a new, untested, & questionable technology. Here's a bit of history: The first reference to garbage collection I have read was in John McCarthy's essay in which he introduced the world to Lisp ([McC60]). That was 1960. Garbage collection goes back to 1960, maybe farther. Since then, it has been used in innumerable programs, & there has been much research in it. In the modern world (2003), more languages use garbage collection than not: Java,

Perl, Python, C-shlep, Visual Basic, Lisp, Scheme, & Java Script. The Unix file system has reference counting[20], so it's not a stretch to say that the Unix shells (Bourne, Korn, bash, & others) use garbage collection. Ada doesn't specify an exact memory management strategy, but allows an implementation to do its own memory management, & garbage collection is a possibility.[21] Contrast this with the manual memory-management holdouts: C, C++, Fortran, & COBOL.

To all the C & C++ programmers (including some really good friends) who keep telling me that garbage collection just can't work for their high-performance special case, How can you be sure if you don't try it? If you are right, you're passing up an opportunity to rub my nose in it. If you don't try it, you're the only one who will never know for sure.

There's no reason not to use an automatic memory manager. It decreases production costs at a slight increase in run-time costs. A *slight* increase, not even noticeable in most programs. Garbage collection is already widely used, & as with function calls, high-level languages, & other improvements in the practice of making software, programmers will rely on it more as the cost of run-time decreases & the cost of programmer time increases.

References

[Ede92] Daniel Edelson. Smart pointers: They're smart, but they're not pointers. UCSC-CRL-92-27. `http://tinyurl.com/c5wjthb`, Jun 1992.

[McC60] John McCarthy. Recursive functions symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[Sto00] Gene Michael Stover. Giggle. `http://giggle.sourceforge.net/`, 2000. C++ garbage collection library.

[Sto03a] Gene Michael Stover. Boehm collector for c and c++, the. *C/C++ Users Journal*, March 2003.

[Sto03b] Gene Michael Stover. Why you should use garbage collection in your program. *personal web site*, April 2003. `http://cybertiggyr.com/gene/htdocs/wgc/`.

[20] Think of hard-links & i-nodes.

[21] I'm not an Ada programmer, so take that with a grain of salt.