Garbage Collection

Algorithms for automatic dynamic memory management

# Introduction

*“One of LISP’s most lasting contributions is a non-language feature namely the term and technique garbage collection, which refers to the system’s method of automatically dealing with storage.”*

**Jean E. Sammet**

**Programming Languages: History and Fundamentals, 1969**

Over the last dozen years, garbage collection has come of age. Whereas it was once confined to realm of Lisp and functional languages, today garbage collection is an important part of the memory management system of many modern programming languages, imperative as well as declarative. Although garbage collection has had a reputation for sloth and for disrupting interactive programs, modern implementation techniques have reduced its overheads substantially, to the point where garbage collected heaps are a realistic option even for traditional languages like C.

Despite the rapid growth in memory sizes of even the most modest computers, the supply of storage is not inexhaustible. Like all limited resources it requires careful conservation and recycling. Many programming languages tiday allow the programmer to allicate and reclaim memory for data whose lifetimes are not determined by lexical scope. Such data is said to be dynamically allocated. Dynamic memory may be managed explicity by the programmer through invocations of built-in or library prcedures that allocate storage and that dispose or free that storage when it is no longer needed.

Manual reclamation of dynamically managed storage is often unsatisfactory. The alternative is to devolve ressponsibility for dynamic memory management to the program’s runtime system. The programmer must still request dynamically allocated storage to be reserved but no longer needs to determine when that memory is no longer required: it is recycled automatically. Garbage collection is precisely this – the automatic management iof dynamically allocated storage. Some authors prefer to distinguish between direct techniques, such as reference counting, and indirect, tracing techniques. However the term garbage collection is widely used to refer to all forms of automatic management of dynamically allocated storage, and we shall use it to refer to both reference counting and tracing methods. We shall need to distinguish between the garbage collcetor and the part of the program that does ‘useful’ work. Following Dijkstra’s terminology, we shall call the user program the mutator since, as far as the collector is concerned, its sole role is to chanhe or mutate the connectivity of the graph of active data structures in the heap.

In this introduction we seel to answer three questions. What problem does garbage collection solve? How costly is garbage collection? By which parameters may different garbage collection algorithms ne compared? We also outline a taxonomy of garbage collection techniques and explain the notation used in the rest of the book. Let us first briefly review the history of programming languages, and in particular the implemetation of storage management, from the 1940s to the present day.

## 1.1 History of storage allocation

The history of the development of programming languages can be considered to be an account of the provision of greater support for abstraction and the automation of actions that were previously manual or explicit.

In the early days of computing all communication between programmer and machine was on a bit-by-bit basis, which simple switches for input. Shortly afterwards, the introduction of simple input and output devices made the exchange of hexadecimal values between operator and machine easier. The next step was to allow programmers to use mnemonic codes that were mechanically translated into binary notation. Nevertheless, users were responsible for every detail of their program’s execution. For example, special attention was needed to count the number of words in the program and find the absolute address of instructions in order to determine whether there was enough space available to load the program and in order to specify the destination of jumps.

By the late 1940s and early 1950s, this book-keeping burden had been transferred to macro codes and assembly languages [Metropolis et al, 1980]. Symbolic programs are easier to write and to understand than machine-language programs primarily symbolic codes. Nevertheless the programmers must still be intimately concerned with how a specific computer operates, and how and where data is represented within the machine. The large number of small machine-dependent details continues to make assembly language programming an exacting task.

To overcome these problems, ideas for high-level programming languages, intended to make the programming task simpler, appeared during the mid to late 1940s. By 1952 the first experimental compilers had appeared, and the first Fortran compiler was delivered in early 1957. A compiler for a high-level language must allocate resources of the target machine to represent the data objects manipulated by the user’s program. There are three ways in which storage can be allocated.

### 1.1.1 Static allocation

The simplest allocation policy is static allocation. All names in the program are bound to storage locations at compile-time: these bindings do not change at run-time. This implies that the local variables of a procedure are bound to the same locations at every activation of the procedure. Static allocation was the origin implementation policy of Fortran, and it is still used by Fortran 77, for example, static allocation has three limitations.

* The size of each data structure must be known at compile-time.
* No procedure can be recursive since all its activations share the same locations for local names.
* Data structures cannot be created dynamically.

Nevertheless, static allocation does have two important benefits. Implementations of statically allocated languages are often fast since no data structures, such as stack frames, need to be created or destroyed during the program’s execution. Since the location of all data is known by the compiler, storage locations can be accessed directly rather than indirectly. Static allocation also offers a safety guarantee: the program cannot fail by running out of space at run-time since its memory requirements are known in advance.

### 1.1.2 Stack allocation

The first block-structured languages appeared in 1958 with Algol-58 and Atlas Autocode. Block-structured languages overcome some of the constraints of static allocation by allocating storage on a stack. An activation record or frame is pushed onto the system stack as each procedure is called, and popped when it returns. Stack organization has five implications.

* Different activations of a procedure do not share the same bindings for local variables. Recursive calls are possible, thereby greatly enhancing the expressivity of the language.
* The size of local data structured such as arrays may depend on a parameter passed to the procedure.
* The values of stack-allocated local names cannot persist from one activation to the next.
* A called activation record cannot outlive its caller.
* Only an object whose size is known at compile-time can be returned as the result of a procedure.

### 1.1.3 Heap allocation

Unlike the last-in, first-out discipline of a stack, data structures in a heap may be allocated and deallocated in any order. Thus activation records and dynamic data structures may outlive the procedure that created them. Heap allocation has a number of advantages.

* Design is about creating abstractions to model real-world problems and many of these are naturally hierarchical: the most common examples are lists and trees. Heap allocation allows the concrete representation of such abstractions to be recursive.
* The size of data structured is no longer fixed but can be varied dynamically. Exceeding built-in limits on the size of data structures, such as arrays, is one of the most common sources of program failure.
* Dynamically-sized objects can be returned as the result of a procedure.
* Many modern programming languages allow a procedure to be returned as the result of another procedure. Stack-allocated languages can do this if they prohibit nested procedures: the static address of the returned procedure is used. Functional and higher-order imperative languages may allow the result of a function to be a suspension or closure: a function paired with an environment of bindings of names to locations. These bindings will therefore outlive the activation of the function that created them.

## 1.2 State, liveness and pointer reachability

The values that a program can manipulate directly are those held in processor registers, those on the program stack (including local variables and temporaries), and those held in global variables. Such locations holding references to heap data form the roots of the computation. Automatic heap-memory management demands that certain rules be followed by the programmer, dynamically allocated data should only be accessible to the user program through the roots, or by following chains of pointers from these roots. In particular, the program should not access random locations in its address space, for example by picking an arbitrary offset from the base of the heap. This restriction is not unique to garbage collection. It is also enforced by strongly-typed languages such as Pascal. Safe use of C’s explicit *malloc/free* allocation mechanisms also demands that the user program does not access unallocated regions of memory.

An individually allocated piece of data in the heap will be called, interchangeably, a node, cell or object1. The rules above imply that the storage mechanism’s view of the liveness of the graph of objects in the heap is defined by pointer reachability. An object in the heap is live if its address is held in a root, or there is a pointer to it held in another live heap node. More formally, define → as the ‘points-to’ relation: for any node or root M and any heap node N, M→N if and only if M holds a reference to N. The set of live nodes in the heap is the transitive referential closure of the set of roots under this relation, i.e. the least set2 live where

1 It will be made clear where the latter term is meant in the object-oriented sense.

2 Mathematical note: such a least set exists by Tarski’s theorem, which states that any equation of the form where is a monotonic operation on sets, has a least fixed point.

For the moment, we note that this view of live cells in the heap is only a conservative estimate of the actual set of cells that are potentially accessible to the program. It may include cells that analysis of the program text or data flow analysis by an optimizing compiler would reveal to be dead. Typical examples include a local variable after its last use in a procedure, as yet uninitialized slots in a stack frame, or an obsolete pointer left in a register (to avoid the cost of clearing it). We shall return to this question later in this chapter and also when we consider techniques for conservative garbage collection in Chapter 9.

A node’s liveness may be determined either directly or indirectly. Direct methods require that a record be associated with each node in the heap, of all references to that node from other heap nodes or roots. The most common direct method is to store a count of the number of pointers to this cell, its reference count, in the cell itself. Direct algorithms for distributed systems may instead keep lists of the remote processors that contain references to each object. In either case, these records musts be kept up to date as the mutator alters the connectivity of the graph in the heap.

Indirect or tracing collectors typically regenerate the set of live nodes whenever a request by the user program for more memory fails. The collector starts from the roots and, by following pointers, visits all reachable nodes. These nodes are considered to be live and all memory occupied by other nodes is made available for recycling. If sufficient memory has been recovered, the user program’s request is satisfied and it is restarted.

## 1.3 Explicit allocation on the heap

### 1.3.1 A simple example

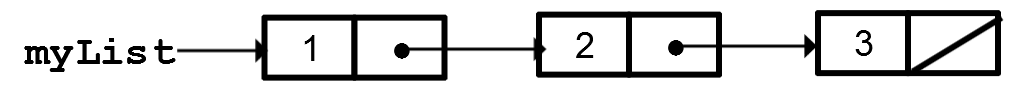
Traditionally, most imperative languages have placed the responsibility for the allocation and deallocation of objects on the heap with the programmer. In Pascal, memory is allocated in the heap by the new procedure. Given a pointer variable p, new(p) causes p to point to newly allocated storage for an object of the type to which p is declared to point. The object is deallocated or freed by calling dispose(p). The program fragment is Algorithm 1.1 on the following page creates a list [1,2,3].

### 1.3.2 Garbage

Dynamically allocated storage may become unreachable. Objects that are not live, but are not free either, are called garbage. With explicit deallocation, garbage cannot be reused; its space has leaked away. We could generate a space leak in the program in Algorithm 1.1 on the following page by adding a line

after the list is created(Diagram 1.2).

Now only the first element of list is accessible to the program; the memory containing items 2 and 3 is out of the program’s reach and can neither be used nor recovered. Automatic storage management can recover inaccessible memory: this is the subject of this book.



**Diagram 1.1 The list built by algorithm 1.1**

****

**Diagram 1.2 creates a space leak**

program pointer(input, output);

type ptr = ↑cell;

cell = record

value : integer;

next : ptr

end;

var myList : ptr;

function Insert (item : integer; list : ptr) : ptr;

var temp : ptr;

begin

new(temp);

temp↑.value := item;

temp↑.enxt := list;

Insert := temp;

end;

begin

myList := Insert(1, Insert(2, Insert(3, nil)))

end

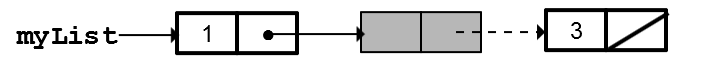
**Algorithm 1.1 Dynamic allocation of a list in Pascal**

### 1.3.3 Dangling references

Memory can also be deallocated while there are still references to it. Suppose we replace the new line in Algorithm 1.1 by

to return item 2 to the heap manager. Again, item 3 has become garbage: this small leak will not harm our tiny program (see Diagram 1.3 on the page). However, the next field of item 1 refers to memory that has been deallocated. A dangling reference has been created.

The program has no control over the use to which the disposed storage is put. It may be cleared, used to store book-keeping information or recycled by the heap manager. If the program follows the dangling reference, the best that can be hoped for is that it will crash immediately. If the heap manager had reallocated the disposed memory to another of the program’s data structures, a single location would represent two different objects. If we are lucky, the program will eventually crash at some future point. If we are unlucky, it will continue to run but produce incorrect results.

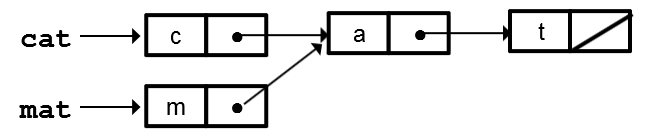
****

**Diagram 1.3 creates a space leak and a dangling pointer**

### 1.3.4 Sharing

Garbage and dangling references are the two sides of the same coin of explicit allocation. Garbage is created by destroying the last reference before an object is deallocated. Dangling references are created by deallocating an object while references to it remain. It might appear that the solution is that both actions --- destruction of the last reference and deallocation of its target --- should be co-ordinated, but this is not easy in the presence of sharing.

Suppose two lists share a common suffix (see Diagram 1.4 on the following page). A well-behaved list disposal routine will recursively deallocate each item of a list when he pointer to the head of the list is destroyed. However, if either cat or mat were destroyed in this way, the other would consist of a single item and a dangling pointer. This was the problem that led to interest in automatic storage reclamation techniques in the late 1950s [McCarthy, 1981].



**Diagram 1.4 Two lists may share a common suffix.**

### 1.3.5 Failures

Dynamic memory in complex programs is hard to manage correctly with explicit allocation and deallocation, and examples of failing programs are common. Programs crash unexpectedly and servers run out of memory for no apparent reason. The effect of such programming errors is indeterminate, particularly in multi-threaded environments. Dangling references may be benign if the hap manager does not reallocate that particular object. Space leaks may lie dormant under testing and even under normal conditions of use. Failures commonly only surface when the program is put under stress or left running for long periods. For example, the input to a complier may be machine generated and violate assumptions about the shape of code that a programmer might reasonably be expected to write. Space leaks may remain undiscovered when the code is run on the development machine. However, when executed on a machine with a smaller memory or on a long-running server, the leak may exhaust the memory. Debugging under these conditions is extremely difficult as failures are often unrepeatable.

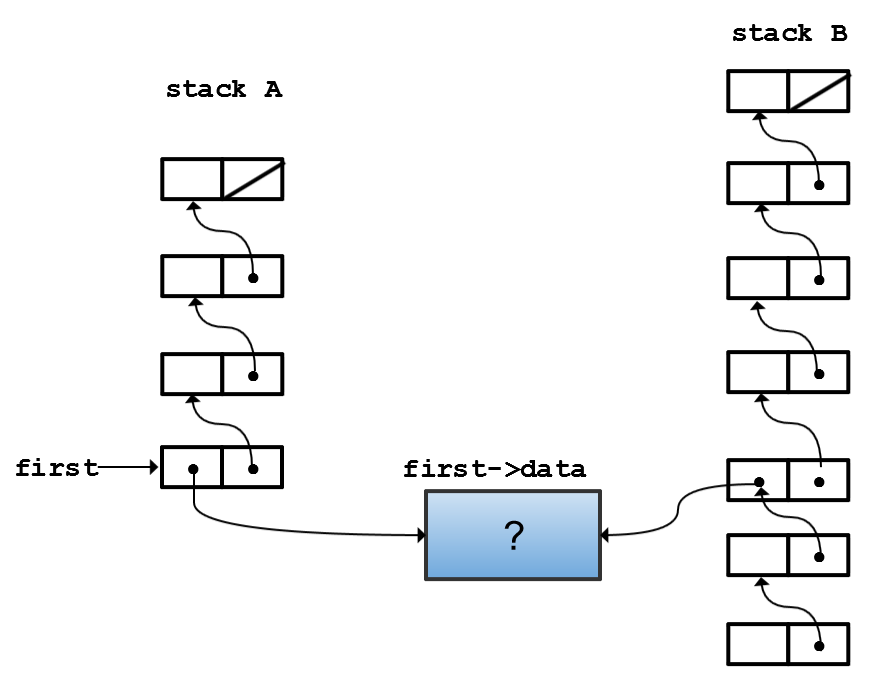
## 1.4 Why garbage collect?

### 1.4.1 Language requirements

Garbage collection may be essential or merely highly desirable. It may be a language requirement: heap allocation is required for data structures that may survive the procedure that created them. If these data structures re then passed to further procedures or functions, it may be impossible for the programmer or compiler to determine the point at which it is safe to deallocate them. The prevalence of sharing and delayed execution of suspensions means that functional languages often have particularly unpredictable execution orders. Garbage collection is mandatory.

### 1.4.2 Problem requirements

Garbage collection may be a problem requirement. Boehm and Chase offer a helpful illustration [Boehm and Chase, 1992]. Suppose a general stack data type is to be implemented is C as A linked list. Each node on the stack contains two pointer fields: data and next. The Pop operation is to deallocate the top of the stack, call it first, and return a pointer to the remainder of the stack. Should Pop deallocate the data referenced from the top element, first->data? If the data is statically allocated, the answer is ‘no’. Otherwise, if this is the last reference to the data, the answer is ‘yes’. If the data may be pushed on to more than one stack --- it is in Diagram 1.5 on the preceding page --- the answer is ‘maybe’. Some convention is required for deallocation even for such a simple abstraction. This will either complicate the interface to the stack, reduce its applicability or force unnecessary copying (so that deallocation decisions can be made locally).



**Diagram 1.5 Should the data object be deallocated when the stack is popped?**

### 1.4.3 Software engineering issues

Software engineering is most succinctly described as the management of complexity in large scale software systems. Two of the most powerful tools available to the software engineer are abstraction and modularity. We strongly believe that explicit memory management cuts against these principles. Automatic memory management gives increased abstraction to the programmer. The model of memory allocation is less low-level and programmers are relieved of the burden of book-keeping detail: their time is better spent on higher-level details of the design, and implementation of the programming problem at hand. Memory management by the run- time system is adopted by all high-level programming languages for static and stack-allocated data. Abstracting away from such low-level issues is universally recognized by designers of high-level programing languages to be essential for global and lexically-scoped data. Programmers do not have to worry where to place global data, or how to set up or take down procedure activation frames on the stack. We believe that the case for abstraction applies equally strongly to heap-allocated data in complex programs.

Reliable code is understandable code. At the level of the module, this means that a programmer should be able to understand its behavior from the module itself, or, in the worst case, a few neighbouring modules. It should not be necessary to understand an entire program before being able to develop a single module. This is clearly essential for large-scale projects involving teams of developers. In contrast, explicit allocation can allow one module to cause the failure of another through space leaks or premature reclamation of storage. The behavior of the module is no longer independent from the context in which it is used.

The oft-cited goal of allowing software components to be combined in the same way as hardware components requires that interfaces should be simple and well-defined. Modules that are extensible may be composed more easily with other modules: the module is reusable in different contexts. Increasing module cohesion also makes programs easier to maintain. Meyer suggests that every module should communicate with as few others as possible, and if any two modules do communicate, they should exchange as little information as possible [Meyer, 1988]. Wilson correctly observes that ‘liveness is a global property’ [Wilson, 1994]. Adding book-keeping detail to module interfaces weakens abstractions and reduces the extensibility of modules. Modifications to the functionality of a module might entail alteration of its memory management code. Since liveness is a non-local matter, changes to book-keeping code might radiate beyond the module being developed.

While global explicit dynamic memory management may be efficient and appropriate for monolithic systems built from hierarchical designs by stepwise refinement, this approach to design seems at odds with the philosophy of object-orientation. It conflicts with the principle of minimal communication and clutters interfaces. If objects are to be reused in different contexts, the new context must understand these rules of engagement, but this reduces the freedom of composition of objects. One author has suggested that the problem of memory management in complex systems may only be solvable without garbage collection if programs are designed with correct memory management as their prime goal [Nagle, 1995]. Garbage collection, on the other hand, uncouples the problem of memory management from class interfaces, rather than dispersing it throughout the code. This is why it has been a fundamental component of many object-oriented languages.

A further indication of the extent of this problem is the range of tools available to assist with checking correct usage of heap memory: the best-known examples include CenterLine [CenterLine, 1992] and Purify [Purify, 1992]. The very existence of tools of this kind reveals the importance of correct memory management and the difficulty of getting it right. However, such tools are only practically useful as debugging aids since they impose a considerable run-time overhead on programs (the CenterLine interpreter by a factor of fifty, the Puriy link-time library by a factor of two to four [Ellis, 1993]).

Although these tools are often very useful for tracking down programming errors, they do not address the heart of the problem. Debugging tools do nothing to simplify the interfaces of complicated systems, nor do they enhance the reusability of software components. Considerable effort still must be devoted to correcting an implementation or, even worse, a design after a leak or a dangling reference is discovered. Debugging tools tackle the symptoms rather than the disease itself. Garbage collection, on the other hand, is an effective software engineering tool because it relieves the programmer from the burden of discovering memory management errors by ensuring that they cannot arise.

Work by Rovner suggests that a considerable proportion of development time may be spent on memory management bugs [Rovner, 1985]. He estimated that forty percent of the time developing the Mesa system was spent on memory management3. Today, object-oriented programming languages are increasingly commonly used. Programs written in these languages typically allocate a greater proportion of their data on the heap than their conventional procedural counterparts. The data structures generated, and the problems tackled, by object-oriented programs are often more complex. These factors can only increase the intricacy of explicit storage management.

Designers and programmers are tempted to be over-defensive in order to overcome the complexities of explicit dynamic memory management. Data is allocated statically or copied between modules rather than being shared: each module is then free to destroy its copy of the object at will --- the global liveness decision is transformed into a local one. Unnecessary copying and static allocation are, at best, wasteful of space since cautions overestimates of memory requirements must be made. If used on large problems, however, static limits may prove inadequate and the programs will fail.

A commonly used alternative is to build a domain-specific garbage collector. Domain-specific collection often fails to take advantage of advances in garbage collection techniques. Because their applicability is by definition limited, the costs of development of such collectors cannot be amortized over a wide set applications. This means that testing is likely to be less thorough. Wilson notes that the very existence of such weakly engineered collectors is testimony to the importance of garbage collection [Wilson, 1994]. The solution is to make garbage collection part of system rather than a ‘bolt-on’ extra.

3 There is a real need for more research to be published on the cost of memory management bugs to development time.

### 1.4.4 No silver bullet

We do not argue that garbage collection is a mandatory requirement for the solution of every problem in every language. Programs with straightforward dynamic memory requirements may be supported at lower run-time cost by explicit deallocation4. However, beware solutions to simple problems that are reused in more complicated programs: the short-term gain may have a longer-term cost. Problem specifications may make demands that garbage collection may not be able to satisfy. Hard real-time systems demand guarantees that memory requests be small. The problem of garbage collection for hard real-time programming has yet to be solved without the use of special hardware.

Nor do we argue that garbage collection is a panacea for all memory management problems. Garbage collection has its own costs, in terms of both time and space, and we introduce these in the next two sections. Furthermore, although garbage collection removes the two classic bugs of explicit storage management – dangling pointers and space leaks – it is still vulnerable to other errors, and moreover raises debugging problems of its own.

Garbage collection has no solution for the problem of data structures that grow without bound. Detlefs and Kalsow report that such data structures are ‘surprising common’, with one example being the caching of intermediate results to avoid recomputation [Detlefs and Kalsow, 1995].

4 But note that this is not necessarily always true.