CHAPTER 16
Memory

Introduction

This chapter explores the control of memory by Lisp programs. We'll discuss what you have to do to request the allocation of memory, how long your memory references are valid, and how to set about recycling memory when you're done with it.

Whatever your application, the correct handling of memory is critical to its success. Therefore this chapter is important reading. I assume you have a basic familiarity with the meaning of `eq` and with the Lisp data types presented in earlier chapters.

Memory defects can have disastrous consequences while being particularly elusive to locate. It's been said that dealing with memory management can consume 30% to 40% of the programmer resources in a typical project. Mercifully Lisp will spare you much of this: it manages memory automatically and makes many otherwise common memory-related defects impossible. Memory management then becomes transparent or—at worst—an optimization issue. All this is achieved with a built-in library universally known as the *garbage collector* (or *GC* for short).

The main message of this chapter is simply: *trust the GC*. We'll come back to this.

Garbage collection originated with Lisp and for many years was used as an "it's too inefficient!" stick with which to beat the language. Now that computers are larger and faster, and the technology has leaked into several more recent languages (Python and Java™, to name just two) it's become more respectable. So perhaps this is the time to give it some prominence in an introductory book on Lisp.

Every Lisp implementation has a GC. No exceptions. Pretty much everything said here applies to all of them. Indeed, much of it is relevant in some form or another to other language GCs as well. For a general reference, including a Beginner's Guide and an extensive glossary of memory management terms, take a look at the Memory Management Reference which is available online:

http://www.memorymanagement.org/

Everyday Use

How It Works, First Pass

Allocation is really simple and deallocation is trivial. You've been doing both of them all along. Whenever you create something new (a value not `eq` to anything which was there previously):

```
(cons 1 2)

(make-instance 'spong)

(1+ most-positive-fixnum)

(read-line)
```

you're asking the system to allocate memory. Other than in the exceptional situation that your computer has run out of memory altogether, the allocation will be made and a reference to it returned as a result of the function call. It'll be tagged internally (as a `cons`, a `spong`, a `bignum`, a `string`, whatever) and its slots will be correctly filled. That's all there is to it.

As long as your program *references* this value—as long as your code can access it by any means whatsoever—the reference remains valid. Once your program can no longer reach the value it becomes *potential garbage*: the garbage collector will come along soon, automatically reclaim the space and make it available again to the program when further allocation is requested.

So Did I Just Allocate?

Almost every time a function returns a value that wasn't there before, you're allocating. The notable exception to this is that in most implementations you can perform any arithmetic you like involving *fixnums*—integers between `most-negative-fixnum` and `most-positive-fixnum`—and so long as the intermediate and final results are all within this range you won't have allocated.

Arithmetic which you guarantee to be fixnum-only may be compiled very efficiently. (See Chapter 20.) Occasionally all this matters.

Also—again with the caveat that this applies to most but not all implementations—you can perform any operation which returns a character and that too won't allocate.

Exercise

Without—this once!—resorting to documentation, find out the fixnum range for the implementation you're using. Express both ends of the range in terms of a power of 2 or 16, whichever you think in most comfortably. How does this answer relate to your machine's word size? Either: how many bits are unavailable to fixnums in this implementation and what do you think happened to them? or: how can you tell from this that `=` fixnums are not `eq` in your implementation? In the former case, now turn to an authoritative source (documentation, direct question to implementors, ...): are your `=` fixnums necessarily `eq`?

Chapter 20 discusses techniques for reducing allocation when working with floats.

Let's go back to what we said before: almost whatever you're doing, you're allocating. You're probably allocating quite a lot. I just put this to the test by compiling a 20KB source file; the compilation allocated over 5MB. Does this matter?

Exercise

Find out whether your implementation's `cl:time` macro reports allocation. (Some do, some don't. Try it.) If it does, measure the allocation cost of `(read)`.

The main lesson to take away from this chapter is to *trust your garbage collector and the experts who wrote it*. For the most part, allocation is nothing to worry about. Don't fall into premature optimization traps; accept that your program's activities are going to allocate and that the GC will take care of this, unobtrusively. And while it isn't free of cost (so all things considered it's best not to stress your GC with massive and avoidable allocation inside your innermost loops) it's typically very efficient, often more so than manual management with `malloc()` and `free()`.

The reasons for this are many and complex, but here's one that's worth hanging on to: in any complex application almost all allocation is *ephemeral*. In other words, almost all of the memory you allocate isn't going to be around for very long before you drop the reference and the GC comes along to sweep it up. Most modern GCs are built on this premise; they're tuned to make the arrival and departure of short-lived objects very efficient.

What Could Possibly Go Wrong?

A number of errors which are common to manual systems cannot happen under automatic memory management. In particular, GCed languages spare you from wasting energy or thought on either of the following:

Premature free
: Memory recycled before you really finished with it, thus making references which you still need invalid.

Double free
: Explicit attempt to free a memory reference for a second time, very unfortunate if that memory has been reallocated in advance of the second free.

On the other hand there's one error which can't go away and that's the *memory leak*. Under manual management you leak memory by failing to free it when you don't need it any more. In the GC world, you leak memory by retaining a pointer to it forever.

Here's a somewhat contrived example. I say "somewhat" because it's really very easy to make a mistake conceptually similar to this.

```
(defclass environment ()
  ((data     :reader environment-data                 :initarg :data)
   (previous :reader environment-previous-environment :initarg :previous)))

(defvar *current-environment* nil)

(defun make-environment (data)
  (let* ((previous *current-environment*)
         (new (make-instance 'environment
                             :data data
                             :previous previous)))
    (setf *current-environment* new)))
```

Exercise

Let's assume that your program needs access to the current environment and to the data from one previous environment (but no more than that). Explain why `make-environment` leaks memory. Fix the function so it doesn't do that any more.

Pause and Trust

Terminology: the *heap* is the name for the comparatively large block(s) of memory which the Lisp system obtains from the operating system and then manages on your behalf.

We haven't yet considered what you have to do to get the GC to run. The solution, almost always, is: absolutely nothing. The GC simply runs when it has to. Implementors are free to design the GC how they choose but a typical scenario goes as follows. When you allocate, the GC almost always has a bit of spare memory on the heap which it can hand over to you with a minimum of fuss. From time to time this stock of memory will turn out to be empty and that's the point at which a garbage collection will be triggered to reclaim dead objects. GCs are usually built to restrict most of their scavenging work to the *nursery*: a region of the heap where the most recent allocation took place (this makes sense because, as I've already said, most allocation is ephemeral). Occasionally the GC will need to examine more or even all of the heap; some systems may appear to pause briefly while this is going on.

Calling your memory manager a "garbage collector" is a bit of a misnomer. It's really an allocator which occasionally has to divert into maintenance tasks; certainly that's its most public face. Unfortunately, the name has stuck.

Some implementations support manual calls to the GC: functions whose side-effect is to trigger a garbage collection (see for example `excl:gc` in ACL and `hcl:gc-generation` in LW). Use of these from running applications is fairly specialized and you should read your documentation to see what's on offer and when you're advised to use it. To a first order of approximation: don't. Trust your GC.

Some implementations extend the concept of nursery to that of numbered *generations*. Recently allocated objects live in "low numbered" generations and these are collected frequently; if objects survive long enough they're *promoted* to "higher" generations which will be collected less often or not at all. Normally under these circumstances the only question for application developers is whether and when to explicitly collect higher generations. In LW for example that means when, if at all, to call `(hgl:gc-generation t)`. As always, read your documentation for guidance.

The movement of objects by the GC may adversely affect the performance of certain (`eq`, `eql`) hash tables.

While I'm on the subject: most GCs come with banks of dials and switches which allow you to tune their operation. Unless you really know what you're doing (for which I would read: unless you're acting under the close supervision of your implementation's support network) leave all this well alone.

Finally, don't be tempted into mass conversion to destructive versions of Lisp functions (e.g. `delete` instead of its copying counterpart `remove`) just so you can save on allocation and take load off the GC. You'll change your program's semantics (typically for the worse), you probably won't run noticeably faster, and you'll certainly have hurt your GC's feelings. Trust it.

On the Move

As already noted, most modern GCs will move objects which survive enough collections out of the nursery into more long-term accommodation. This housekeeping operation is transparent to your program. In particular things are always fixed up so that `eq` still works. However the printed representations generated by `(print-unreadable-object (... :identity t) ...)` include a permanent record of memory addresses and you can watch these change:

```
CL-USER(1): (setf x (make-instance 'standard-object))
#<STANDARD-OBJECT @ #x20f8c26a>

;; Allocate generously to trigger a collection

CL-USER(2): (dotimes (i 100000) (make-string 100))
NIL

;; The printed representation of X has changed

CL-USER(3): x
#<STANDARD-OBJECT @ #x20d6922a>

;; But it's still the same object as before

CL-USER(4): (eq * ***)
T
CL-USER(5):
```

Exercise

Try this in your own Lisp. Now crank down the allocation in line 2 and try to figure out (order of magnitude guess) how much memory you need to allocate before older objects start getting moved around. Why is this not a brilliant guess as to how many times the GC has run?

Monitoring

At the very least, your Lisp implements the function `cl:room` which prints information about current memory use. `(room t)` prints more information than plain `(room)` and `(room nil)` prints less. For an interpretation of the output you will have to read the relevant sections of your Lisp's documentation.

Exercise

Try all three invocations of room in your implementation.

If you care about the performance of your application you must get used to using `room` and any other help on offer (e.g. for checking fragmentation). Send regular output to the log files which your application surely maintains.

```
(let ((*standard-output* my-logging-stream))
  (room t)
```

By the way, in rough terms: a cons cell takes up a few words; a hash table, a few words per entry plus some fixed overhead; an array, one word per entry plus a fixed overhead.

Exercise

Using `(room)` or otherwise, how much memory is consumed in your implementation by an array of a million `double-float`s? What happens if you make the array with `:element-type 'double-float`?

As already noted, in some implementations the `time` macro reports allocation. There may also be extended functionality on offer, such as the LispWorks macro `hcl:extended-time` which includes a breakdown of GC activity in its report. If you can obtain figures for the proportion of time spent in GC you're on your way to knowing whether a performance problem is memory-bound.

If you intend to use `time` (or the clock on the wall) to compare running speeds for different variants of your code, be aware that performance can vary greatly according to the precise state that your GC was in when the test run started. Conduct the test three or four times and average the results. If your application supports a manual invocation of GC then use it to flush out all ephemeral garbage before each timed run. If not, then the best ways to level the playing field are to either run each experiment more times still or to restart Lisp after each run.

Advanced Use

The topics which follow are either more specialized or go into greater depth. Don't be put off if it all becomes heavy going: skimming this material now will be useful to you one day. You can come back and refer to it in detail later should you need it.

How It Works, Optional Second Pass

Consider this function which we defined in the last chapter (and recall that a call to `mp:process-wait` will only complete when the function passed to it returns true):

```
(defun handle-in-process (process function &rest arguments)
  (let* ((queue (getf (process-property-list process) 'queue))
         (no-result (cons nil nil))
         (result no-result))
```

```
(enqueue queue
         (lambda ()
           (setf result (apply function arguments))))
(process-wait (format nil "Waiting for response from ~a"
                      (process-name process))
              (lambda () (not (eq result no-result))))
result))
```

The call `(cons nil nil)` returns a new object of type `cons`. We know it's new because it's not `eq` to any cons which was in either your program or the underlying Lisp before. You've allocated a *cons cell*.

You might occasionally encounter the term *consing*. Lispers use this to refer to allocation of memory in general, not just the building of new cons cells. The reasons for this are buried in history.

The cons is bound by the `let*` into two lexical variables: `no-result` and `result`. While either of these variables points to this cons, the GC won't scavenge it. Also, although by inspection this won't be the case here, the same applies if anything else in the program (another variable, a member of some list, a slot in a CLOS instance, ...) can reach that cons.

We see that after the return from `process-wait`, `result` is no longer `eq` to `no-result`. We can also see that the value of `no-result` hasn't changed (because nothing in the code has changed it). So at this point, `result` is no longer bound to the cons and all that's keeping the cons alive is `no-result`.

But the only reference to `no-result` is the closure `(lambda () (not (eq result no-result)))`. After `process-wait` returns this closure is no longer referenced by your code and therefore neither is `no-result` and therefore neither is the value of `no-result`. The cons cell is now *garbage*; the garbage collector is free to come along and recycle the memory which held it.

Exercise

Without splitting hairs you should be able to find three more points in `handle-in-process` where allocation clearly occurs. Do all of these new values become garbage? What assumptions do you have to make about `process-wait` and about the behavior of the Lisp process `process`?

It's not often that you need to think about this particular level of detail while you're coding. The GC and general design of the language cooperate to just make things work.

Stacks and Roots

As we've seen, the call `(cons nil nil)` allocates heap memory. The result of the call is the address of this memory (albeit dolled up as a Lisp object, a cons cell).

The bindings of `no-result` and `result` to this cons:

```
(let* ((no-result (cons nil nil))
       (result no-result))
  ...)
```

consume memory too. The implementation has no choice about this: the two lexical variables could later take on values other than our cons cell and their current values have to be stored somewhere at all times. But the compiler is smart and knows that once we've left the `let*` form these bindings will no longer be operative. So these variable values needn't live on the heap: most (one would hope: all) Lisp compilers will store them on the *stack*.

The stack is a data structure which, like the GCed heap, was originally invented for Lisp back in the 1950s. It's sufficiently common to the internals of more or less any computer system that your hardware undoubtedly has specific instructions for managing it. It's used for nested temporary storage: at the "top" of the stack, the local variables and return addresses for the current function call; "below" that the variables and return addresses for the function which called this one; and so on all the way back to `main()` or whatever local equivalent started the application running in the first place. On entry to any block of code, the top of the stack can be *extended* very cheaply thus giving that code as much temporary storage as it needs; on exit the *stack pointer* is reverted to its previous value and the temporary storage is no longer visible. No GC is required for running the stack; this form of memory management is more or less orthogonal to heap storage (whether manual or automatic) and indeed over the years some languages have got by perfectly well with no concept of heap at all.

There are three reasons for going into this detail, and here's the first. If you know hand-on-heart that the cons cell returned by `(cons nil nil)` is definitely not going to be accessible once your function exits, you can tell the compiler to *stack allocate* it:

```
(let* ((no-result (cons nil nil))
       (result no-result))
  (declare (dynamic-extent no-result))
  ...)
```

The `dynamic-extent` *declaration* is a guarantee on your part. "I promise never to do anything that would result in this variable's value living on beyond the lifetime of the variable itself." But it's nothing more than this and the Lisp system is completely free to ignore it. (This is explained further in Chapter 20 which discusses optimizations.) Your documentation should list the data types for which the implementation supports *stack consing*. With ACL, for example, you can stack allocate "`&rest` arguments, closures, and some lists and vectors".

The declaration shown above is fairly pointless, as it exposes your program to potential defects (if you break your promise) in exchange for an immeasurably small gain. So by way of illustration consider instead this definition lifted from one of my applications:

```
(defun apply-predicate (element match environment)
  (let* ((predicate (first match))
         ;; Application needs a fresh copy of this list for each call to
         ;; execute-predicate...
         (match (copy-list (rest match))))
    (execute-predicate element predicate match environment)))
```

Although the lists concerned aren't very long, the function `apply-predicate` is called very many times. Now, I wouldn't want to suggest for a moment that the GC concerned couldn't handle ephemeral lists with blinding efficiency. However it turned out that just declaring the variable `match` to be `dynamic-extent` made the whole application run 4% faster.

```
(defun apply-predicate (element match environment)
  (let* ((predicate (first match))
         (match (copy-list (rest match))))
    (declare (dynamic-extent match))
    (execute-predicate element predicate match environment)))
```

That's not bad for a one-line change. We'll come back to performance issues in Chapter 20; for further details about the `dynamic-extent` declaration see:

http://www.lispworks.com/documentation/HyperSpec/Body/d_dynami.htm

Exercise

Take at least a quick look at that.

Now let's turn to the other reasons for considering stacks. I've said that GC works by examining whether Lisp values are reachable. The question is, "reachable from what?" and the answer is more or less this. When the GC runs it starts by crawling over the stack, assuming any values found there are *live*. If any of these refer to other Lisp values (in the way that a cons refers to its `car` and `cdr`, and a `boundp` symbol refers to its `symbol-value`) then these too are by definition live, and so on. Somewhere on the stack (or maybe hardwired into the search some other way) there'll be a reference to some symbol such as `nil` which via its `symbol-package` points into the package system from which we can reach every other symbol and all your global values and function definitions. That's how come your program—and the implementation itself—don't get swept away by the GC. The starting points for the GC's walk over the heap are known as its *roots*.

Finally: each process operates in its own stack, remembering its execution strand: its function call history, lexical variables, catch tags and so forth. In many implementations the stack also holds per-process bindings for global variables. When the process switches, the stack gets switched with it. So actually the GC's roots must somehow

include all stacks, not just the current one. Oh, and where are the stacks themselves allocated? On the heap.

Exercise

When a process run-function completes, what do we hope will happen to its stack?

We'll come back to roots and what they can reach when we discuss *treeshaking* in Chapter 28.

Weak References

The premise of garbage collection is that if you can find a path of references to an object from any of the system's roots, then that object won't be collected. Although this is almost always just what we wanted, there's a useful paradigm which it doesn't support. Suppose you need to store some objects in a cache (perhaps so you can perform periodic housekeeping tasks on them). Just keeping an object in this cache would ordinarily prevent it from ever being garbage collected; the cache becomes a memory leak. You'd like to arrange that an object in the cache can be collected when there are no other references to it, in other words when the cache itself is the only remaining reference to the object. You can do this by making the references from the cache to its members *weak*. If you do this then the GC won't follow them. An object which can be reached non-weakly by some other route will stay in the cache. If the only references to the object in the image are weak, they'll be replaced with `nil`s.

All this sounds very theoretical; an example is in order. We'll use ACL.

```
CL-USER(1): (setf unweak (make-instance 'standard-object)
                  cache (let ((weak (make-instance 'standard-object))
                              (c (make-array 4 :weak t)))
                          (setf (aref c 1) weak
                                (aref c 3) unweak)
                          c))
#(NIL #<STANDARD-OBJECT @ #x20fab4f2> NIL #<STANDARD-OBJECT @ #x20faafca>)
CL-USER(2): (gc)
CL-USER(3): cache
#(NIL NIL NIL #<STANDARD-OBJECT @ #x20e7715a>)
CL-USER(4):
```

Immediately after line 1 there are two objects in the cache. One of them can also be reached by the symbol-value of `unweak` and so it remains in the cache after a garbage collection. The other object is only reachable via the weak array and so the GC can carry it off, leaving a `nil` in its place.

Most implementations support weak hash tables and weak arrays. As ever the details will vary, so check your documentation. SBCL has an interesting variation on this: the *weak pointer*. You create one of these with the call `(sb-ext:make-weak-pointer`

`object)` and interrogate it with `(sb-ext:weak-pointer-value weak-pointer)` which returns two (i.e. multiple) values: either the original object and `t` or—if the reference is no longer valid—`nil` and `nil`.

Finalizers

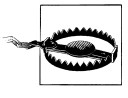
These are functions which can be run, when an object is collected, to reclaim external resources associated with it. A classic example of this is an open `file-stream`: when this is GCed it's best that the system should release its underlying filesystem handle. If you're tempted to suggest that the Lisp programmer should have to call `close` on every file-stream they open (or else leak the underlying handle), consider the problem of determining whether a file-stream is still live. The need for finalizers to reclaim non-memory resources is parallel to that of the application to reclaim heap space in general.

Exercise

Macroexpand a call to `with-open-file` to confirm that `close` is always called. Imagine an application where `with-open-file` wouldn't be appropriate and in which finalizers could be useful.

Levels of support for finalizers differ between implementations (as do the details for invoking them). In ACL it's very simple indeed: you just pass the object to be finalized along with a function of one argument to `excl:schedule-finalization`. Instead of scavenging that object the GC will pass it to the finalizing function. Provided that function hasn't *resurrected* your object—done anything to retain it on the heap (such as pushing it onto a list, or rescheduling it for another finalization)—it'll be scavenged during a later collection.

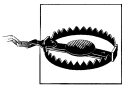
```
CL-USER(1): (schedule-finalization (make-instance 'standard-object)
                                   'print)
#(#<STANDARD-OBJECT @ #x20e762a2> PRINT NIL)              ; a weak array!
CL-USER(2): (gc)

#<STANDARD-OBJECT @ #x20e762a2>
CL-USER(3):
```



AllegroCache connections are not automatically closed. The result of calling `db.ac:open-network-database` is stored in `db.ac:*allegrocache*` thus overwriting any previous value. If you haven't stored that somewhere else, it'll *leak*: it's consuming resources and you can't get them back. (Also, the Express Edition only supports three open connections at a time so this is one leak you can't afford.)

Exercise

Whether or not you have access to ACL, write a finalizer for AC connections which guarantees that `ac.db:close-database` will be called. (See "Connecting to the Database" in Chapter 13 for details of this function.)



There is absolutely no promise about when, or in what order, or indeed even whether finalizers will be run. Do not ever count on finalizers for anything other than the husbanding of resources.

Avoiding Stack Overflow

Stacks are designed for efficiency. There'll be a block of contiguous memory; the *stack pointer* tells the system how much of that block is currently in use; the stack pointer is moved one way when values are pushed onto the stack and the other way when they're popped off it.

If the pointer hits the far end of the stack, the stack is said to have *overflowed*. It's extremely unlikely that you'll ever see this happen unless your code is either broken or deliberately deeply recursive.

Stack overflow may be implemented as a `storage-condition`. This is not a subtype of `error`. If you wish to handle both conditions together, use their common supertype `serious-condition`.

With one exception over the last 20 years, every time this happened to me it was because my code was broken: the recursion I'd written simply wasn't working properly. Just once however, it happened that the story wasn't quite that clear cut.

The following recursive code has been adapted from an old version of one of my applications. Appropriately for an example in a chapter about garbage collection, it's a simple *tri-coloring algorithm*. It starts from some frob and accumulates into a hash table that frob's "neighbors" and recursively all their neighbors, and so on. A GC might use something similar for walking the heap and recording which objects are reachable from the roots.

```
(defun walk-frobs-recurse (starting-frob)
  (let ((frobs (make-hash-table)))
    (labels ((walk (frob)
               (setf (gethash frob frobs) t)
               (dolist (neighbor (frob-neighbors frob))
                 (unless (gethash neighbor frobs)
                   (walk neighbor)))))
      (walk starting-frob)
      frobs)))
```

Exercise

Look up "tri-color algorithm", for example at

http://www.memorymanagement.org/glossary/t.html#tri-color.marking

and identify the three colors in `walk-frobs-recurse`.

Exercise

Why would a GC not use a hash table for this?

My QA discovered that if you were to link several thousand frobs together in a simple chain, neighbor to neighbor, my equivalent to this code would overflow the stack.

Exercise

Although the above code has been heavily redacted you should be able to get it to break. Define a suitable class of frobs and try it out in more than one Lisp if you can. How do they compare?

Although implementations usually document ways of controlling how large stacks will be and might also support some notion of *extending* them (growing stacks on-the-fly), this solution wasn't really satisfactory to us because we wanted to be able to handle any pattern of data without breaking anything. "Only several thousand frobs" sounded pitiful. So the code was rewritten to be non-recursive, implementing its own stack on the heap.

```
(defun walk-frobs-iterate (starting-frob)
  (let ((frobs (make-hash-table))
        (stack (list starting-frob)))
    (loop while stack do
          (let ((frob (pop stack)))
            (setf (gethash frob frobs) t)
            (dolist (neighbor (frob-neighbors frob))
              (unless (gethash neighbor frobs)
                (push neighbor stack)))))
    frobs))
```