Memory management within a program is broken into two main categories:

1. Stack Management
2. Heap Management

Stack Management

Each function has an activation record called a frame that records the instance of the call.
Each frame shares the code, but the values are different.
A frame includes:

- return address
- arguments
- local variables
- temporary instances

Frames developed in a few stages:

1. FORTRAN (1958)
   Frames were static rather than dynamic; thus FORTAN had no stack or recursion
2. C (1972)
   Frames statically defined frame layout, but the instantiation is left to runtime.
   This allows recursion!
   This does not allow for a heap, or global variables.

```
int f(int n) {
  ...
  char[n] a;
  ...
} // NOT ALLOWED
```

3. Algol (1960)
   Frames are stored in a stack which can grow as the program runs.
   Data is accessed by two pointers (%rbp and %rsp) to the base and top of the stack, respectively.

```
int f(int n) {
  ...
  char[n] a;
  ...
} // OKAY
```

   We can thus introduce the idea of **nested functions**

```
int f(void) {
  int x;
  int g(void) {
    int y;
    ...
    return x + y;
  }
  g();
}
f();
```

   We store two linked lists of addresses.

   (a) **The Static Chain**
   This forms the connection of definer frames.
   The depth is the nest level.
   This tends to be pretty short.

   (b) **The Dynamic Chain**
   This forms the connection of caller frames.
   The depth is the call level.
   This can be relatively long.

   To make this work, functions are represented by a pair of pointers

   (a) a pointer to the code (ip)

(b) a pointer to the defining frame (op)

A function is thus an (ip, op) pair.
These pointers are called FAT, since they hold two words as opposed to thin C pointers.
This is how continuations work!

4. ML A nesting function can return whilst the nested is still valid (currying).

```
(fun x -> fun y -> x + y) 3;;
-: 'a -> 'a = <fun>
(* this returns a function that remembers the 3! *)
```

The stack can be expanded or contracted with a single pointer addition instruction.
Thus heap management is far more expensive, so function calls are thus not as efficient.

Heap Management

How do we arrange to free unused storage?
The simplest approach: explicit free operation
Problems:

- It is a hassle for the programmers.
- A programmer may forget to free. (memory leak!)
- A programmer may preemptively free. (dangling pointer → undefined access!)

We can prevent this with **garbage collection**
≡ there is no 'free' — the underlying memory management system handles it.
Problems:

- How do we kept track of pointers into the heap (roots)?
  A static table is kept with a frame template and indirect pointers to the heap.
- How do we free indirect pointers to the heap?
  mark & sweep; recursively mark all reachable elements, then free all unmarked.
- How does the heap managed keep track of used/unused space?
  We keep a free list in unused block 1 for fast allocation

The heap can become very large as the program's data gets very large.
This can lead to delays, as memory allocation can take up to linear time.
Additionally, blocks near the beginning of the heap will tend to be more fragmented.
If they fragment too much, they chew up CPU time as runts.
Thus we use a **roving pointer/next-fit list**
≡ leave the free pointer at the location of the last successful allocation

We must be sure to make a distinction between using a free/del operator & a garbage collector.
If we are using free/del, keeping track of roots is delegated to the program.
The program is then responsible for not misusing the primitive (arbitrary free, double free, etc.).
This can be more efficient if the program is well written, BUT it is less reliable, since:

- Dangling Pointers (usage of freed storage)
- Memory Leaks (allocated and never freed memory)
- Imposter Pointers (free (void *) 2323; — luckily 2323 is not an address on SEASNet)

The garbage collector is by far the most common choice in modern languages.
Thus garbage collectors avoid the above problems; how?

- Storage is not reclaimed if it is reachable from roots.
- We want the GC to be as accurate as possible in determining which storage is reachable
  BUT reclaiming immediately can be expensive, so we sometimes wait.
- Languages do not allow forging of pointers (in at least Java and OCaml)

Mark and Sweep is generally a pretty good algorithm, but it makes malloc O(object count).
How can we speed it up?

1. Mark and Sweep in a separate thread? NO — the locks will almost always be a bottleneck.
2. Combine GC with C/C++?
   Our reflex is to say no, since
   (a) C lets us theoretically access any address.
   (b) The compiler doesn't share roots or object layout with the program

BUT it is possible with **conservative garbage collection**.
The garbage collector doesn't know which data is a pointer, but it does have memory bounds.
The objects that the heap manager knows about are:

- in registers (16 in x86-64)
- on the stack (%rsp in x86-64)
- in static variables (*.o are static files)

We search in these locations and assume any pointers within range are valid
This can lead to issues:

```
char msg[] = "\300\147...";
long long int gates_balance = 347140743208473;
// might happen to be a pointer to the heap.
malloc(1); // returned (char *)347140743208473 a while ago.
// The conservative GC will think that 1-byte object is still in use.
```

BUT if we are careful, wrong-assumptions should be rare:
We place our heap is in obscure locations, with addresses not likely to look like actual numbers.
If the wrong assumptions are rare, then we won't have a lot of leaks.

Who actually does the garbage collection?
Typically garbage collection is done in user-mode code without much formal help.
This is done for performance reasons.

The heap manager knows where objects are but does not show the OS.
For instance, one could find the source code for malloc() and free() written in C.
(Mostly, except malloc could request a huge chunk of RAM)
This is a reasonably popular option and is used in the C++ gcc code and C emacs code.
The idea of this is that we never need to call free!

```
#define free(p) ((void*) p)
```

There are two major garbage collection methods:

1. **Generation-Based Copying Collector (Java)**
   Older objects appear farther to the left; new allocations are taken from the nursery.
   Objects tend to point to older objects that rarely mutate, so garbage is in newer generations.
   In actuality, there is some overlap, since there are exceptions for backward pointers We thus only need to garbage collect the nursery, and the roots to a degree.
   The generation-based keyword refers to the the separately collected generations.

   The copying keyword comes from the merging to remove fragmentation.
   Upsides:

   (a) Runtime is O(bytes used).
   (b) We need not access free areas.
   This might seem small, but avoiding a free list means we don't have to fill the L1 & L2 cache.
   This is huge if the objects are small.

   Downsides:

   (a) We have to do a LOT of copying.
   (b) We may need to free data not directly controlled by the GC.
   We need to change any pointer referring to these objects.
   We call Object.finalize() to collect garbage from a given object.
   This is a method that by default does nothing, and is for cleanup.
   mark and sweep can solve the finalize() issue.
   Upon sweep, unmarked objects have finalize() called.

   Since this collector doesn't sweep, Java has to blend the two approaches:
   Java does generation-based garbage collection for standard objects.
   It does mark and sweep for objects with a user-defined finalize() method.

   ISSUE — Multithreading.
   The naive multithreaded garbage collector requires synchronization, and thus is slow.
   Java avoids a global lock by giving each nursery its own lock Typically, threads share only old objects, so in practice this works well In the worst case, this reduces to the global lock case

ISSUE — Real-Time.
In this case, we don't care particularly about the speed of 'new', but predictability.
Thus we do an incremental garbage collection with some marking or sweeping each call.
This can be tricky, but is often used in Java applications.

2. **Reference-Counting Garbage Collector (C-Python)**
Each object keeps track of the number of references to it.

Positives:

(a) quick garbage discovery/reclamation

Negatives:

(a) cyclic object patterns are not reclaimed
(b) Lost references

Not having cycles isn't inherently terrible, but people laughed off Python's method.
This is because they either needed avoid cycles or remember to break them.
Nowadays, Python integrates an extra phase of mark & sweep.
The timing of the mark and sweep is dependent on multiple heuristically tuned parameters.
Therefore, we need to optimize these when building a large application.

A trick that sometimes works and sometimes fails miserably is called **object polling**.
Say we know we are going to allocate a bunch of 'struct cons' objects. We then do

```
struct cons { struct cons * car; struct cons *cdr; };
// We thus allocate en masse and keep our own free list
struct cons *free_conses;
struct cons *next_cons(void) {
  if (free_conses) {
    struct cons * r = free_conses;
    free_conses = r->car;
    return r;
  }
  return malloc(sizeof(struct cons)); // expensive
}
void free_cons(struct cons *p) {
  p->car = free_conses;
  free_conses = p;
}
```

This works with mark and sweep, is but terrible for generation, since it copies the free list.