Baby’s First Garbage Collector

↩ ↪

When I get stressed out and have too much to do, I have this paradoxical reaction where I escape from that by coming up with *another* thing to do. Usually it’s a tiny self-contained program that I can write and finish.

The other morning, I was freaking myself out about the book I’m working on and the stuff I have to do at work and a talk I’m preparing for Strange Loop, and all of the sudden, I thought, “I should write a garbage collector.”

Yes, I realize how crazy that paragraph makes me seem. But my faulty wiring is your free tutorial on a fundamental piece of programming language implementation! In about a hundred lines of vanilla C, I managed to whip up a basic mark-and-sweep collector that actually, you know, collects.

Garbage collection is considered one of the more shark-infested waters of programming, but in this post, I’ll give you a nice kiddie pool to paddle around in. (There may still be sharks in it, but at least it will be shallower.)

Reduce, reuse, recycle

The basic idea behind garbage collection is that the language (for the most part) appears to have access to infinite memory. The developer can just keep allocating and allocating and allocating and, as if by magic, never run out.

Of course, machines don't have infinite memory. So the way the implementation does this is that when it needs to allocate a bit of memory and it realizes it's running low, it *collects garbage*.

"Garbage" in this context means memory it previously allocated that is no longer being used. For the illusion of infinite memory to work, the language needs to be very safe about "no longer being used". It would be no fun if random objects just started getting reclaimed while your program was trying to access them.

In order to be collectible, the language has to ensure there's no way for the program to use that object again. If the program can't get a reference to the object, then it obviously can't use it again. So the definition of "in use" is actually pretty simple:

1. Any object being referenced by a variable still in scope is in use.

2. Any object referenced by another in-use object is in use.

The second rule is the recursive one. If object A is referenced by a variable, and it has some field that references object B, then B is in use since you can get to it through A.

The end result is a graph of *reachable* objects—all of the objects in the world that you can get to by

starting at a variable and traversing through objects. Any object *not* in that graph of reachable objects is dead to the program and its memory is ripe for a reaping.

Marking and sweeping

There are a bunch of different ways you can implement the process of finding and reclaiming all of the unused objects, but the simplest and first algorithm ever invented for it is called “mark-sweep”. It was invented by John McCarthy, the man who invented Lisp and beards, so implementing today is like communing with one of the Elder Gods, but hopefully not in some Lovecraftian way that ends with you having your mind and retinas blasted clean.

The algorithm works almost exactly like our definition of reachability:

1. Starting at the roots, traverse the entire object graph. Every time you reach an object, set a “mark” bit on it to true.

2. Once that’s done, find all of the objects whose mark bits are *not* set and delete them.

That’s it. I know, you could have come up with that, right? If you had, *you’d* be the author of a paper cited hundreds of times. The lesson here is that to be famous in CS, you don’t have to come up with really obscure stuff, you just have to come up with obvious stuff *first*.

A pair of objects

Before we can get to implementing those two steps, let's get a couple of preliminaries out of the way. We won't be actually implementing an interpreter for a language—no parser, bytecode, or any of that foolishness—but we do need some minimal amount of code to create some garbage to collect.

Let's play pretend that we're writing an interpreter for a little language. It's dynamically typed, and has two types of objects: ints and pairs. Here's an enum to identify an object's type:

```
typedef enum {
  OBJ_INT,
  OBJ_PAIR
} ObjectType;
```

A pair can be a pair of anything, two ints, an int and another pair, whatever. You can go surprisingly far with just that. Since an object in the VM can be either of these, the typical way in C to implement it is with a tagged union. We define it thusly:

```
typedef struct sObject {
  ObjectType type;

  union {
    /* OBJ_INT */
    int value;

    /* OBJ_PAIR */
    struct {
      struct sObject* head;
      struct sObject* tail;
    };
  };
} Object;
```

The main Object struct has a `type` field that identifies what kind of value it is—either an int or a pair. Then it has a union to hold the data for the int or pair. If your C is rusty, a union is a struct

where the fields overlap in memory. Since a given object can only be an int *or* a pair, there's no reason to have memory in a single object for all three fields at the same time. A union does that. Groovy.

A minimal virtual machine

Now we can use that datatype in a little virtual machine. The VM's role in this story is to have a stack that stores the variables that are currently in scope. Most language VMs are either stack-based (like the JVM and CLR) or register-based (like Lua). In both cases, there is actually still a stack. It's used to store local variables and temporary variables needed in the middle of an expression. We model that explicitly and simply like so:

```
#define STACK_MAX 256

typedef struct {
  Object* stack[STACK_MAX];
  int stackSize;
} VM;
```

Now that we've got our basic data structures in place, let's slap together a bit of code to create some stuff. First, let's write a function that creates and initializes a VM:

```
VM* newVM() {
  VM* vm = malloc(sizeof(VM));
  vm->stackSize = 0;
  return vm;
}
```

Once we have a VM, we need to be able to manipulate its stack:

```
void push(VM* vm, Object* value) {
  assert(vm->stackSize < STACK_MAX, "Stack overflow!");
  vm->stack[vm->stackSize++] = value;
```

```
}

Object* pop(VM* vm) {
  assert(vm->stackSize > 0, "Stack underflow!");
  return vm->stack[--vm->stackSize];
}
```

Now that we can stick stuff in “variables”, we need to be able to actually create objects. First, a little helper function:

```
Object* newObject(VM* vm, ObjectType type) {
  Object* object = malloc(sizeof(Object));
  object->type = type;
  return object;
}
```

That does the actual memory allocation and sets the type tag. We’ll be revisiting this in a bit. Using that, we can write functions to push each kind of object onto the VM’s stack:

```
void pushInt(VM* vm, int intValue) {
  Object* object = newObject(vm, OBJ_INT);
  object->value = intValue;
  push(vm, object);
}

Object* pushPair(VM* vm) {
  Object* object = newObject(vm, OBJ_PAIR);
  object->tail = pop(vm);
  object->head = pop(vm);

  push(vm, object);
  return object;
}
```

And that’s it for our little VM. If we had a parser and an interpreter that called those functions, we’d have an honest to God language on our hands. And, if we had infinite memory, it would even be able to run real programs. Since we don’t, let’s start collecting some garbage.

Marky mark

The first phase is *marking*. We need to walk all of the reachable objects and set their mark bit. The first thing we need then is to add a mark bit to `Object`:

```
typedef struct sObject {
  unsigned char marked;
  /* Previous stuff... */
} Object;
```

When we create a new object, we modify `newObject()` to initialize `marked` to zero.

To mark all of the reachable objects, we start with the variables that are in memory, so that means walking the stack. That looks like this:

```
void markAll(VM* vm)
{
  for (int i = 0; i < vm->stackSize; i++) {
    mark(vm->stack[i]);
  }
}
```

That in turn calls `mark`. We'll build that in phases. First:

```
void mark(Object* object) {
  object->marked = 1;
}
```

This is the most important bit, literally. We've marked the object itself as reachable. But remember we also need to handle references in objects—reachability is *recursive*. If the object is a pair, its two fields are reachable too. Handling that is simple:

```
void mark(Object* object) {
  object->marked = 1;

  if (object->type == OBJ_PAIR) {
```

```
    mark(object->head);
    mark(object->tail);
  }
}
```

There's a bug here. Do you see it? We're recursing now, but we aren't checking for *cycles*. If you have a bunch of pairs that point to each other in a loop, this will overflow the C callstack and crash.

To handle that, we simply need to bail out if we get to an object that we've already processed. So the complete `mark()` function is:

```
void mark(Object* object) {
  /* If already marked, we're done. Check this first
     to avoid recursing on cycles in the object graph. */
  if (object->marked) return;

  object->marked = 1;

  if (object->type == OBJ_PAIR) {
    mark(object->head);
    mark(object->tail);
  }
}
```

Now we can call `markAll()` and it will correctly mark every reachable object in memory. We're halfway done!

Sweepy sweep

The next phase is to sweep through all of the objects we've allocated and free any of them that aren't marked. But there's a problem here: all of the unmarked objects are, by definition, unreachable! *We* can't get to them!

The VM has implemented the *language's* semantics for object references, so we're only storing pointers to

objects in variables and the pair fields. As soon as an object is no longer pointed to by one of those, the VM has lost it entirely and actually leaked memory.

The trick to solve this is that the VM can have its *own* references to objects that are distinct from the semantics that are visible to the language *user*. In other words, we can keep track of them ourselves.

The simplest way to do this is to just maintain a linked list of every object we've ever allocated. We extend `Object` itself to be a node in that list:

```
typedef struct sObject {
  /* The next object in the list of all objects. */
  struct sObject* next;

  /* Previous stuff... */
} Object;
```

The VM keeps track of the head of that list:

```
typedef struct {
  /* The first object in the list of all objects. */
  Object* firstObject;

  /* Previous stuff... */
} VM;
```

In `newVM()` we make sure to initialize `firstObject` to `NULL`. Whenever we create an object, we add it to the list:

```
Object* newObject(VM* vm, ObjectType type) {
  Object* object = malloc(sizeof(Object));
  object->type = type;
  object->marked = 0;

  /* Insert it into the list of allocated objects. */
  object->next = vm->firstObject;
  vm->firstObject = object;
```

```
  return object;
}
```

This way, even if the *language* can't find an object, the language *implementation* still can. To sweep through and delete the unmarked objects, we traverse the list:

```
void sweep(VM* vm)
{
  Object** object = &vm->firstObject;
  while (*object) {
    if (!(*object)->marked) {
      /* This object wasn't reached, so remove it from the list
         and free it. */
      Object* unreached = *object;

      *object = unreached->next;
      free(unreached);
    } else {
      /* This object was reached, so unmark it (for the next GC)
         and move on to the next. */
      (*object)->marked = 0;
      object = &(*object)->next;
    }
  }
}
```

That code is a bit tricky to read because of that pointer to a pointer, but if you work through it, you can see it's pretty straightforward. It just walks the entire linked list. Whenever it hits an object that isn't marked, it frees its memory and removes it from the list. When this is done, we will have deleted every unreachable object.

Congratulations! We have a garbage collector! There's just one missing piece: actually calling it. First let's wrap the two phases together:

```
void gc(VM* vm) {
  markAll(vm);
  sweep(vm);
}
```

You couldn’t ask for a more obvious mark-sweep implementation. The trickiest part is figuring out when to actually call this. What does “low on memory” even mean, especially on modern computers with near-infinite virtual memory?

It turns out there’s no precise right or wrong answer here. It really depends on what you’re using your VM for and what kind of hardware it runs on. To keep this example simple, we’ll just collect after a certain number of allocations. That’s actually how some language implementations work, and it’s easy to implement.

We extend `VM` to track how many we’ve created:

```
typedef struct {
  /* The total number of currently allocated objects. */
  int numObjects;

  /* The number of objects required to trigger a GC. */
  int maxObjects;

  /* Previous stuff... */
} VM;
```

And then initialize them:

```
VM* newVM() {
  /* Previous stuff... */

  vm->numObjects = 0;
  vm->maxObjects = INITIAL_GC_THRESHOLD;
  return vm;
}
```

The `INITIAL_GC_THRESHOLD` will be the number of objects at which we kick off the *first* GC. A smaller number is more conservative with memory, a larger number spends less time on garbage collection. Adjust to taste.

Whenever we create an object, we increment `numObjects` and run a collection if it reaches the max:

```
Object* newObject(VM* vm, ObjectType type) {
  if (vm->numObjects == vm->maxObjects) gc(vm);

  /* Create object... */

  vm->numObjects++;
  return object;
}
```

I won't bother showing it, but we'll also tweak `sweep()` to *decrement* `numObjects` every time it frees one. Finally, we modify `gc()` to update the max:

```
void gc(VM* vm) {
  int numObjects = vm->numObjects;

  markAll(vm);
  sweep(vm);

  vm->maxObjects = vm->numObjects * 2;
}
```

After every collection, we update `maxObjects` based on the number of *live* objects left after the collection. The multiplier there lets our heap grow as the number of living objects increases. Likewise, it will shrink automatically if a bunch of objects end up being freed.

Simple

You made it! If you followed all of this, you've now got a handle on a simple garbage collection algorithm. If you want to see it all together, here's the full code. Let me stress here that while this collector is *simple*, it isn't a *toy*.

There are a ton of optimizations you can build on top of this—in GCs and programming languages, optimization is 90% of the effort—but the core code here is a legitimate *real* GC. It's very similar to the collectors that were in Ruby and Lua until recently. You can ship production code that uses something exactly like this. Now go build something awesome!