Baby's First Garbage Collector

DECEMBER 08, 2013 C CODE LANGUAGE

↩ ↪

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

Hi! I'm **Bob Nystrom**, the one on the left.

I wrote a book called **Game Programming Patterns**.

You can email me at `robert` at this site or follow me on twitter at `@munificentbob`.

ELSEWHERE

- Code at github
- Code at bitbucket
- Tweets at twitter
- Photos at flickr
- Video at vimeo
- Posts at google+

CATEGORIES

- code 66
- language 41
- magpie 24
- c-sharp 13
- dart 12
- game-dev 12
- java 10
- cpp 8
- game-patterns 6
- parsing 6
- roguelike 6
- design 5
- go 5

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, it never fails.

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 it 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 that's being referenced by a variable that's still in scope is in use.

- js 4
- book 3
- c 3
- finch 3
- python 3
- ruby 3
- blog 2
- f-sharp 2
- lua 2
- ai 1
- beta 1
- blogofile 1
- game 1
- jasic 1
- javascript 1
- music 1
- oop 1
- optimization 1
- oscon 1
- politics 1
- scheme 1
- typescript 1
- visualization 1

All 72 articles…

2. Any object that's referenced by another object that's in use 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's 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 you implementing it now 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.

It 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 smart stuff, you just have to come up with dumb 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](http://www.flickr.com/photos/raganwald/212588975/) 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'll 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 wrap that in a little virtual machine structure. Its 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'll 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've got a VM, we need to be able to manipulate its stack:

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

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

OK, 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 t
  Object* object = malloc(sizeof(Objec
```

```
  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_I
  object->value = intValue;
  push(vm, object);
}

Object* pushPair(VM* vm) {
  Object* object = newObject(vm, OBJ_P
  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'll 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);
  }
}
```

But 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 stack and crash.

To handle that, we just 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. Ch
     to avoid recursing on cycles in t
  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 elements. As soon as an object is no longer pointed to by one of those, we've 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'll extend `Object` itself to be a node in that list:

```
typedef struct sObject {
  /* The next object in the list of al
  struct sObject* next;

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

The VM will keep track of the head of that list:

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

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

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

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

  /* Insert it into the list of alloca
```

```
  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 just need to traverse the list:

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

      *object = unreached->next;
      free(unreached);
    } else {
      /* This object was reached, so u
         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'll extend `VM` to track how many we've created:

```
typedef struct {
  /* The total number of currently all
  int numObjects;

  /* The number of objects required to
  int maxObjects;

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

And then initialize them:

```
VM* newVM() {
```

```
  /* Previous stuff... */

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

The `INITIAL_GC_THRESHOLD` will be the number of objects at which you 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 t
  if (vm->numObjects == vm->maxObjects

  /* 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 (and in things like GC 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!

Andrew Berls • 2 years ago

Fantastic post! One note: in your 7th

code block (where `pushInt` and 'pushPair' are defined), it should read `newObject(vm, type)` instead of `allocate(vm, type)`, no?

12 • Reply • Share ›

munificent Mod
Andrew Berls • 2 years ago

Sharp eye! I renamed that function but missed a few spots. Fixed now. Thanks!

3 • Reply • Share ›

Jon Topper • 2 years ago

It's worth noting that your decision to store the mark bit in the object itself is what original versions of Ruby did - leaving them inefficient with RAM on fork() operations (because the mark process dirtied all the copy-on-write pages next time around). More detail at http://www.rubyenterpriseediti...

10 • Reply • Share ›

Dimitris Zorbas • 2 years ago

"It was invented by John McCarthy, the man who invented Lisp and beards", lol

8 • Reply • Share ›

esteban • 2 years ago

This is one of my favorite tech post titles of all time. I'm so sick of seeing titles like "You're doing X wrong" or "Why X sucks" or "Why X is better than Y", etc, etc. This one is simple and funny in a not-trying-so-hard type of way.

5 • Reply • Share ›

gasche • 2 years ago

A new trick: in the sweep phase, you don't need to reset the mark. You can just use another integer as the "traversed" mark for the next traversal. You can increment the mark at each traversal, and if you're not comfortable with overflows wrap it around a fixed bound that must be higher than the total number of colors you need, plus one. This saves you one write per non-freed object, which is nice.

3 • Reply • Share ›

munificent Mod gasche
• 2 years ago

Great call! I was going for maximum clarity here, but that's a good first optimization to do in a real implementation.

• Reply • Share ›

Erin Keenan gasche
• 2 years ago

This is a good trick.

It's also generally applicable anytime you want to remember what you've seen for one round, and then reset, w/o having to pay the cost of resetting.

Example: I once wrote a program to find high-scoring boggle boards. To score a board, you need to track which words you've seen (so you don't double-count words). Then, when you're scoring a new board, you need to reset which words you've seen. Using a simple incrementing counter to "reset" seen is much faster (and easier!) than going back through and explicitly clearing a seen boolean flag.

• Reply • Share ›

imicro • 2 years ago

GCObject *luaC_newobj (lua_State *L, int tt, size_t sz, GCObject **list, int offset) {
global_State *g = G(L);
GCObject *o = obj2gco(cast(char *, luaM_newobject(L, tt, sz)) + offset);
if (list == NULL)
list = &g->allgc; /* standard list for collectable objects */
gch(o)->marked = luaC_white(g);
gch(o)->tt = tt;
gch(o)->next = *list;
*list = o;
return o;
}

I was able to figure out the above from Lua source because it is same

see more

3 • Reply • Share ›

kirbyfan64sos • a year ago

+1! I finally get why mark-and-sweep collectors don't have to worry about circular references!

2 • Reply • Share ›

Chris Clarke • 2 years ago

This is a great post, as it actually gives you realistic code. Most language development/low level stuff gives you a 'toy' program, which is pointless because you must optimize a more efficient algorithm afterward.

Some still say that the program is 'doing *XYZ* wrong', but realistically unless there's an error in the code itself I know the GC isn't made to be super efficient. If your goal in the post was to make a very efficient GC you would've probably used tri-color marking.

Thank you for making this. I don't know if it's a coincidence that Ruby got a new 2.1 GC recently. ;)

2 • Reply • Share ›

eriksvedang • 2 years ago

You should write a book. Oh yeah, you mentioned that you were... Anyway, this was some seriously awesome writing (and code). Funny, clear and of very practical value. Thanks for making it!

2 • Reply • Share ›

Marcos Savoury • 2 years ago

I've always wondered how the VM managed to find the 'unreachable' objects and you did a great job explaining it. Thanks for the quality post.

2 • Reply • Share ›

Matt Greer • 2 years ago

Great post! Question: Why does sweep need to use a pointer to the pointer? It seems like it'd work just fine with the pointer as is.

2 • Reply • Share ›

Felix Jodoin → Matt Greer • 2 years ago

Because the list isn't just being traversed from the perspective of "some pointer"; we want to make changes to the next fields (or the vm->firstObject field), so we're actually taking the address of those fields instead of using a bunch of "previous object" pointers.

*object = unreached->next;

is the pointer-to-pointer way of saying

previous->next = unreached->next;

.. where dereferencing "object" is giving us the memory location of the previous object's next field (object = &(*object)->next;).

8 • Reply • Share ›

munificent Mod → Matt Greer • 2 years ago

It lets us handle freeing the first node without having to special case it. When you free, you need to fix the pointer to the node you're deleting. In most cases, that pointer will be the `next` pointer of the previous node. For the first node, though, it will be the VM's `firstObject` pointer. Using a pointer to a pointer lets you handle both of those generically.

1 • Reply • Share ›

Wolf → Matt Greer • 2 years ago

I *think* it's to balance out the address (&) operator to the virtual machine. Then again, I'm not very experienced with C...

• Reply • Share ›

Matteo Poropat • 6 months ago

The whole article is very detailed and inspiring, but the part that made me crazy is the reference to lovecraftian elder gods.
You gain a reader (by the way I'm reding the blog from few days, attracted by the posts about rouguelikes) and you sell me a copy of your book (paper version, waiting for Amazon droids to knock on my window with the package)

1 • Reply • Share ›

Kaidul Islam Sazal • a year ago

Great Article! Thanks man! Your presentation was very nice

1 • Reply • Share ›

Alan • 2 years ago

Great work, thank you for sharing!

1 • Reply • Share ›

grobie • 2 years ago

Thanks, that was a really interesting post! Very well written.

1 • Reply • Share ›

José Manuel • 2 years ago

Awesome post! Shouldn't the newVM function set `firstObject` to NULL? Otherwise the sweep phase might never end.

1 • Reply • Share ›

munificent Mod → José Manuel • 2 years ago

Yup! Fixed now.

• Reply • Share ›

david karapetyan • 2 years ago

Cool. Looking forward to more stuff like this.

1 • Reply • Share ›

Christian Friedl • 9 months ago

v <--- jaw (mine)
.
.
. <------- dropping move
___ <--- floor

Everything I read on such "low level" subjects made them out to be extremely complicated.

Care to write a book on compiler construction? :-)

BTW, hauberk rulez!

• Reply • Share ›

allinlabs • a year ago

I encountered a nasty bug resulting in potential huge memory leak.
When you push objects in the stack *n* times and the *vm->maxObjects* == *n*, if you pop the stack *n* times, *vm->maxObjects* become equal to zero because *vm->maxObjects* = *vm->numObjects * 2* but *vm->numObjects* is actually equal to zero now that the stack got emptied.

Here's a example to make it clearer:

```
VM* vm = newVM();

for (int i = 0; i < 100000; i+
  for (int j = 0; j < 256; j++
    pushInt(vm, i);
```

see more

• Reply • Share ›

Paul Six • a year ago

I am currently working on an interpreter for Scheme, but a lot of the objects in scheme such as

symbols are static, so shouldn't there be a function such as:
Object* newObject_nogc such that it doesn't increment the object count and is freed at the accord of the caller?

• Reply • Share ›

munificent Mod → Paul Six • a year ago

You can do that. You can also just have the mark pass treat all of those objects as roots which will prevent them from being freed. Either way works.

• Reply • Share ›

Zach Reedy • 2 years ago

Awesome tutorial but I think I may have found a shark in the pool.
This implementation expects that the stack is used in a very specific way, if you were to push 2 integers and then do an operation (such as adding them together) you'd end up popping both off and then pushing on the result. This is fine and expected behavior regarding the stack, but since the VM keeps track of all objects that have been allocated between cleaning, if the GC hasn't ran before the result is pushed onto the stack then you actually end up leaking memory by overwriting the object that's still in existence in the stack's path.

This can be simulated in the test2() function if you add a pushInt(vm, 3) after the pops. The assertion fails as you would expect because the newly pushed value has overwritten the object at the bottom of the stack.

You could run the GC every time you

You could run the GC every time you push/pop something onto the stack but this is a pretty obvious (and serious) hit to performance.
I'm not too sure there's any way to avoid this. Maybe I'm misunderstanding something here but it seems like a really serious issue with the GC's implementation.
Any idea how to fix this?
Thanks.

• Reply • Share ›

munificent Mod → Zach Reedy • 2 years ago

> you actually end up leaking memory by overwriting the object that's still in existence in the stack's path.

Nope, there's no leak here. Yes, we overwrote the *stack's* pointer to the allocated object. But we are free to do that. In fact, doing that is exactly what allows the previous object referenced in that slot to be collected later.

When the GC is marking, it walks the stack to see what's still in use. But when it's *collecting*, it doesn't walk the stack, it walks the linked list that contains every allocated object. We didn't overwrite that link, so it can find the number and delete it.

> This can be simulated in the test2() function if you add a pushInt(vm, 3) after the pops. The assertion fails as you would expect because the newly pushed value has

overwritten the object at the bottom of the stack.

In this case, it's failing because the 3 we just pushed on the stack is still in memory. And that's correct! It's still on the stack and available for use.

If you add another pop() after that to discard the 3, it runs without error.

1 • Reply • Share ›

Zach Reedy munificent • 2 years ago

Ah, thank you! I see