Programming Languages (7) Garbage Collection (GC) : A Brief Introduction

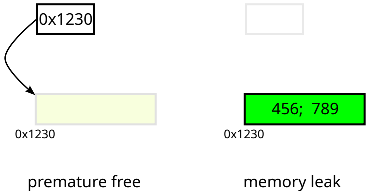
Kenjiro Taura

Contents

Contents

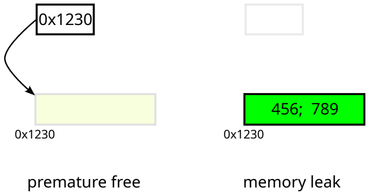
Garbage Collection (GC)

- the fundamental issue is the mismatch between
  - the period in which objects are accessed
  - the period in which the memory block for it is retained



Garbage Collection (GC)

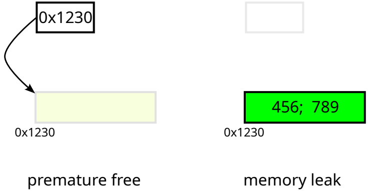
- the fundamental issue is the mismatch between
  - the period in which objects are accessed
  - the period in which the memory block for it is retained



- ⇒ Garbage collection (GC)
  - ratain memory block for objects if they could ever be accessed in future and reclaim otherwise
  - the system automatically does that
  - ⇒ eliminate memory leak and corruption

Garbage Collection (GC)

- the fundamental issue is the mismatch between
  - the period in which objects are accessed
  - the period in which the memory block for it is retained



- ⇒ Garbage collection (GC)
  - ratain memory block for objects if they could ever be accessed in future and reclaim otherwise
  - the system automatically does that
  - ⇒ eliminate memory leak and corruption
- the question: how does the system know *which objects may be accessed in future*?

Objects that may {ever/never} be accessed

- the precise judgment is undecidable
- (at the start of line 2) "the object pointed to by `p` will ever be accessed" $\iff$ "`f(x)` will terminate and return 0" $\rightarrow$ you need to be able to solve the halting problem...

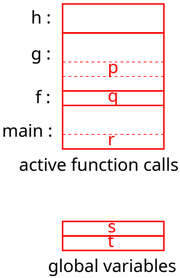
```
1 int main() {
2   if (f(x) == 0) {
3     printf("%d\n", p->f->x);
4   }
5 }
```

- $\rightarrow$ *conservatively* estimate objects that *may be* accessed in future
  - NEVER reclaim those that are accessed
  - OK not to reclaim those that are in fact never accessed
- in the above example, OK to retain objects pointed to by `p` when the line 2 is about to start

Objects that “may be” accessed

- global variables
- local variables of active function calls (calls that have started but have not finished)

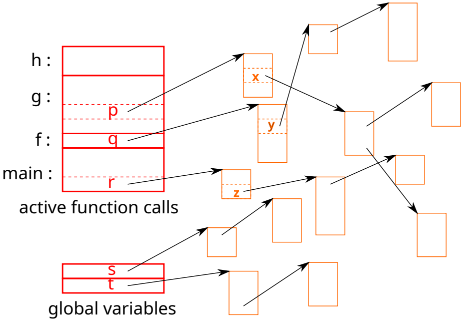
```
int * s, * t;
void h() { ... }
void g() {
  ...
  h();
  ... = p->x ...  }
void f() {
  ...
  g()
  ... = q->y ... }
int main() {
  ...
  f()
  ... = r->z ... }
```



Objects that "may be" accessed

- global variables
- local variables of active function calls (calls that have started but have not finished)
- objects reachable from them by traversing pointers

```
int * s, * t;
void h() { ... }
void g() {
  ...
  h();
  ... = p->x ...  }
void f() {
  ...
  g()
  ... = q->y ... }
int main() {
  ...
  f()
  ... = r->z ... }
```



Contents

The basic workings (and terminologies) of GC

- an object: the unit of automatic memory allocation/release (malloc in C; objects in Java; etc.)

The basic workings (and terminologies) of GC

- an object: the unit of automatic memory allocation/release (malloc in C; objects in Java; etc.)
- the root: objects accessible without traversing pointers, such as global variables and local variables of active function calls

The basic workings (and terminologies) of GC

- an object: the unit of automatic memory allocation/release (malloc in C; objects in Java; etc.)
- the root: objects accessible without traversing pointers, such as global variables and local variables of active function calls
- reachable objects: objects reachable from the root by traversing pointers

The basic workings (and terminologies) of GC

- an object: the unit of automatic memory allocation/release (malloc in C; objects in Java; etc.)
- the root: objects accessible without traversing pointers, such as global variables and local variables of active function calls
- reachable objects: objects reachable from the root by traversing pointers
- live / dead objects: objects that {may be / never be} accessed in future

The basic workings (and terminologies) of GC

- an object: the unit of automatic memory allocation/release (malloc in C; objects in Java; etc.)
- the root: objects accessible without traversing pointers, such as global variables and local variables of active function calls
- reachable objects: objects reachable from the root by traversing pointers
- live / dead objects: objects that {may be / never be} accessed in future
- garbage: dead objects

The basic workings (and terminologies) of GC

- an object: the unit of automatic memory allocation/release (malloc in C; objects in Java; etc.)
- the root: objects accessible without traversing pointers, such as global variables and local variables of active function calls
- reachable objects: objects reachable from the root by traversing pointers
- live / dead objects: objects that {may be / never be} accessed in future
- garbage: dead objects
- collector: the program (or the thread/process) doing GC

The basic workings (and terminologies) of GC

- an object: the unit of automatic memory allocation/release (malloc in C; objects in Java; etc.)
- the root: objects accessible without traversing pointers, such as global variables and local variables of active function calls
- reachable objects: objects reachable from the root by traversing pointers
- live / dead objects: objects that {may be / never be} accessed in future
- garbage: dead objects
- collector: the program (or the thread/process) doing GC
- mutator: the user program (vs. collector). very GC-centric terminology, viewing the user program as someone simply "mutating" the graph of objects

The basic workings (and terminologies) of GC

- an object: the unit of automatic memory allocation/release (malloc in C; objects in Java; etc.)
- the root: objects accessible without traversing pointers, such as global variables and local variables of active function calls
- reachable objects: objects reachable from the root by traversing pointers
- live / dead objects: objects that {may be / never be} accessed in future
- garbage: dead objects
- collector: the program (or the thread/process) doing GC
- mutator: the user program (vs. collector). very GC-centric terminology, viewing the user program as someone simply "mutating" the graph of objects

> the basic principle of GC:
> objects unreachable from the root are dead

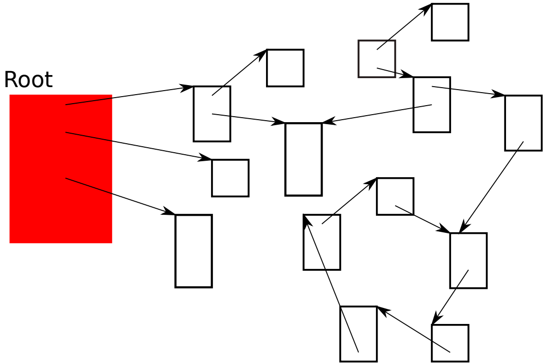
Contents

The two major GC methods

- traversing GC:
  - simply traverse pointers from the root, to find (or *visit*) objects reachable from the root
  - reclaim objects not visited
  - two basic traversing methods
    - mark&sweep GC
    - copying GC
- reference counting GC (or RC):
  - during execution, maintain the number of pointers (reference count) pointing to each object
  - reclaim an object when its reference count drops to zero
  - note: an object's reference count is zero → it's unreachable from the root
- remark: "GC" sometimes narrowly refers to traversing GC

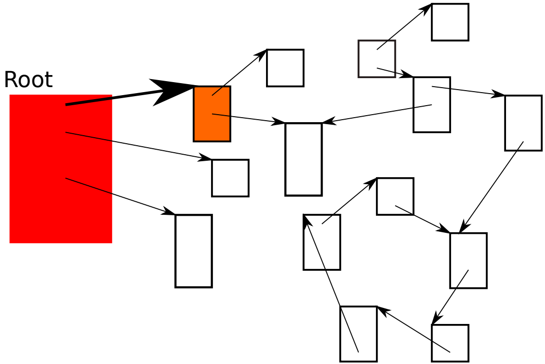
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



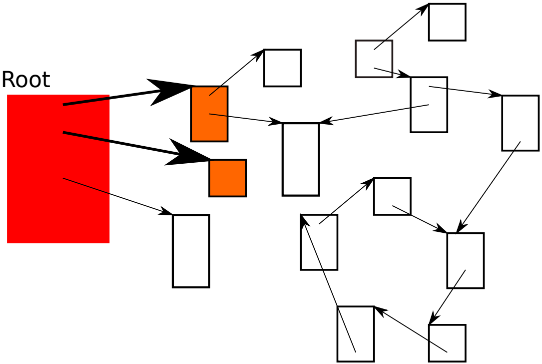
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



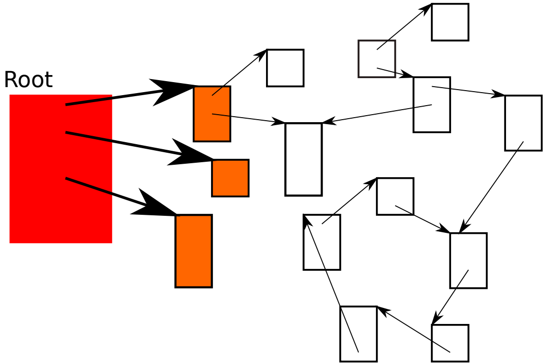
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



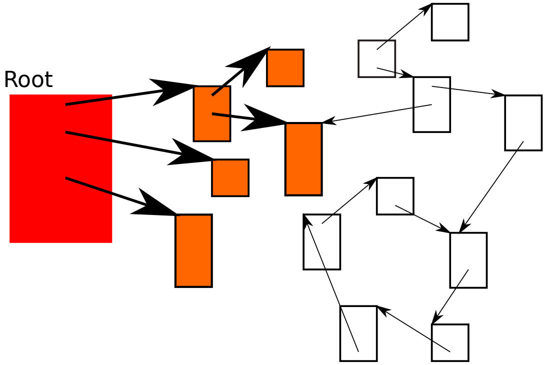
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



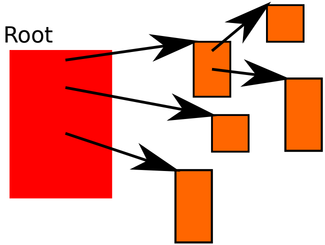
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



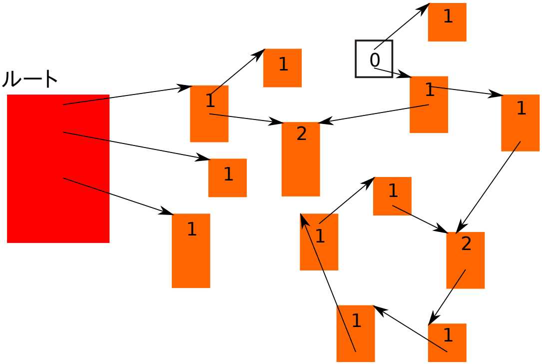
How traversing GC works

- traverse pointers from the root
- once all pointers have been traversed, objects that have not been visited are garbage
- the difference between mark&sweep and copying is covered later



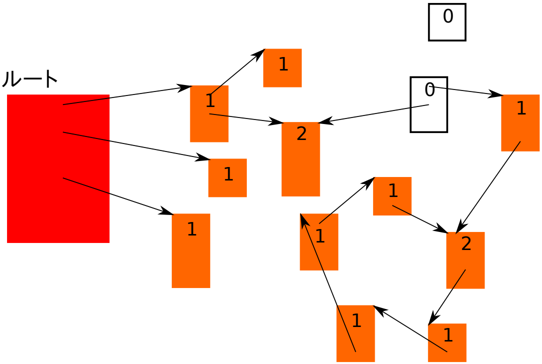
How reference counting works

- each object has a reference count (RC)
- update RCs during execution; e.g., upon `p = q;` →
  - the RC of the object `p` points to `-=` 1
  - the RC of the object `q` points to `+=` 1
- reclaim an object when its RC drops to zero → RCs of objects pointed to by the now reclaimed object decrease



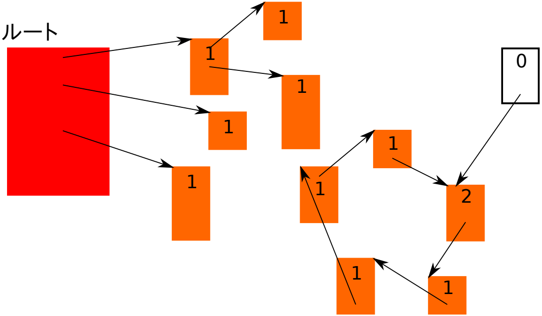
How reference counting works

- each object has a reference count (RC)
- update RCs during execution; e.g., upon `p = q;` →
  - the RC of the object `p` points to `-=` 1
  - the RC of the object `q` points to `+=` 1
- reclaim an object when its RC drops to zero → RCs of objects pointed to by the now reclaimed object decrease



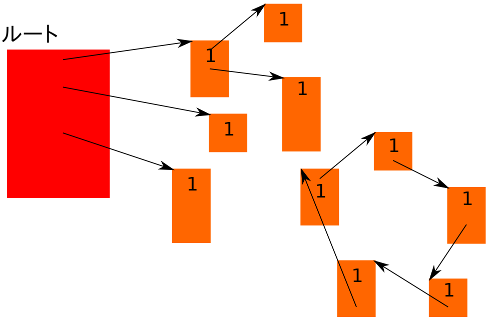
How reference counting works

- each object has a reference count (RC)
- update RCs during execution; e.g., upon `p = q;` →
  - the RC of the object `p` points to `-=` 1
  - the RC of the object `q` points to `+=` 1
- reclaim an object when its RC drops to zero → RCs of objects pointed to by the now reclaimed object decrease



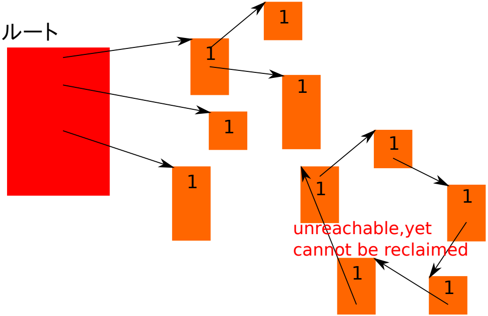
How reference counting works

- each object has a reference count (RC)
- update RCs during execution; e.g., upon `p = q;` →
  - the RC of the object `p` points to `-=` 1
  - the RC of the object `q` points to `+=` 1
- reclaim an object when its RC drops to zero → RCs of objects pointed to by the now reclaimed object decrease



How reference counting works

- each object has a reference count (RC)
- update RCs during execution; e.g., upon `p = q;` →
  - the RC of the object `p` points to `-=` 1
  - the RC of the object `q` points to `+=` 1
- reclaim an object when its RC drops to zero → RCs of objects pointed to by the now reclaimed object decrease



When an RC changes

- a pointer is updated `p = q; p->f = q;` etc.
- a function gets called

```
int main() {
  object * q = ...;
  f(q);
}
```

- a variable goes out of scope or a function returns

```
f(object * p) {
  ...
  {
    object * r = ...;

  } /* RC of r should decrease */
  ...
  return ...; /* RC of p should decrease */
}
```

- etc. any point pointer variables get copied / become no longer used

Shortcomings of GC

- may be costly
  - what if a traversing GC visits 10GB of reachable objects, to reclaim only 100MB of memory?
- may pause the user program (mutator) for a long time
  - a traversing GC does not want the mutator to modify the object graph while traversing it
- may slow the user program
  - esp. by reference counting

methods to overcome some of the issues will be covered in later weeks