Programming Languages (6)
Memory Management

Kenjiro Taura

Contents

Contents

Memory management in programming languages

- all data (integers, floating point numbers, strings, arrays, structs, ...) used in a program need a space (register or memory) to hold them
- ideally, programming languages *manage* them on behalf of the programmer; i.e.,
  - when creating a new data, find an available space for it
  - *retain* the space as long as the data is still "in use"
  - *reclaim/reuse* the space when the data is "no longer used"
- three approaches covered

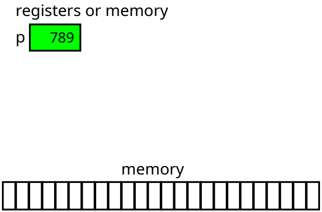
manual		C, C++
garbage collection	traversing reference counting	Python, Java, Julia, Go, OCaml, etc.
Rust ownership		Rust

Data representation

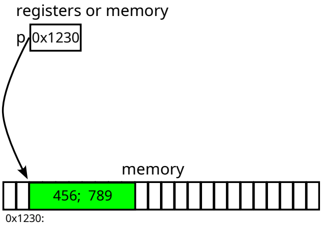
- data in your program must be somehow represented in the machine code
- some data (e.g., integers and floating point numbers) can be trivially mapped to machine representations
- less trivial is how to map
  - multiword data (structs),
  - unknown-size or large data (e.g., arrays and strings),
  - mutable data,
  - recursive data (lists),
  - etc.

Two strategies

- immediate

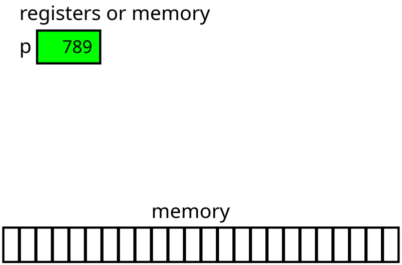


- indirect



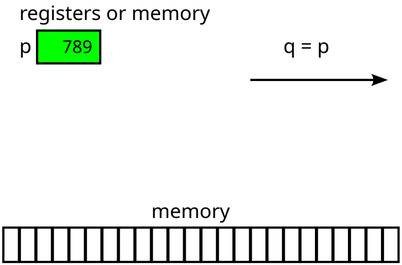
Immediate representation

- typically used for small data (integers, floating point numbers, characters, etc.) that fit on a single register (e.g., 64 bits)



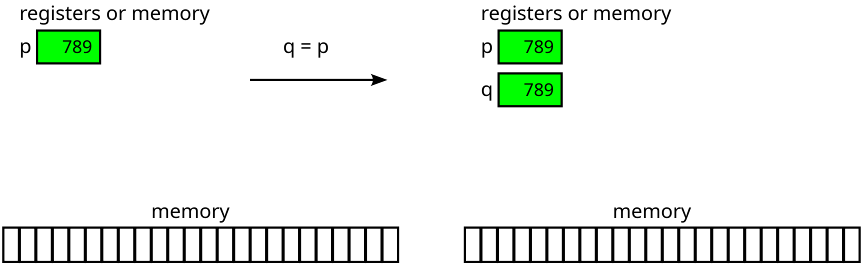
Immediate representation

- typically used for small data (integers, floating point numbers, characters, etc.) that fit on a single register (e.g., 64 bits)
- upon an assignment-like operation, the whole data gets copied (cheap as data are small)



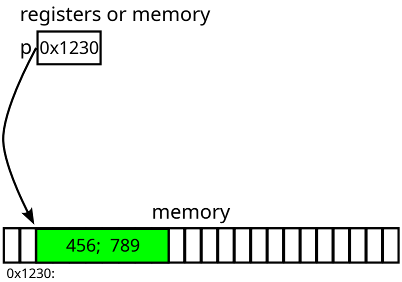
Immediate representation

- typically used for small data (integers, floating point numbers, characters, etc.) that fit on a single register (e.g., 64 bits)
- upon an assignment-like operation, the whole data gets copied (cheap as data are small)



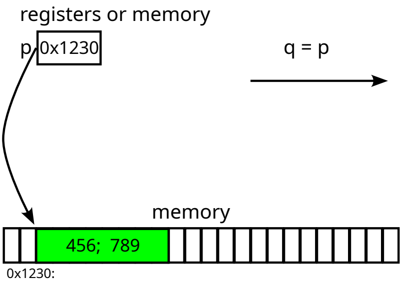
Indirect representation

- typically used for multi-word data



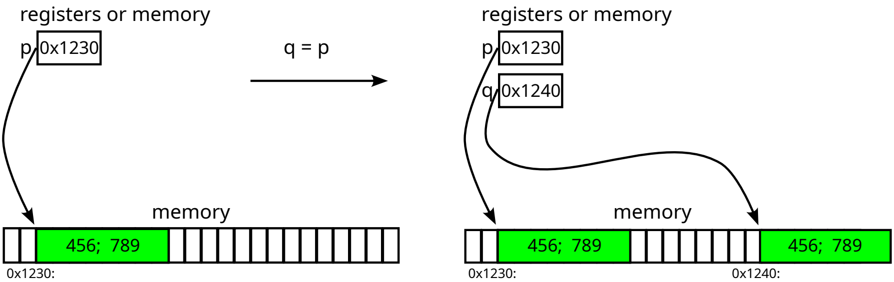
Indirect representation

- typically used for multi-word data
- upon an assignment-like operation, there are two choices



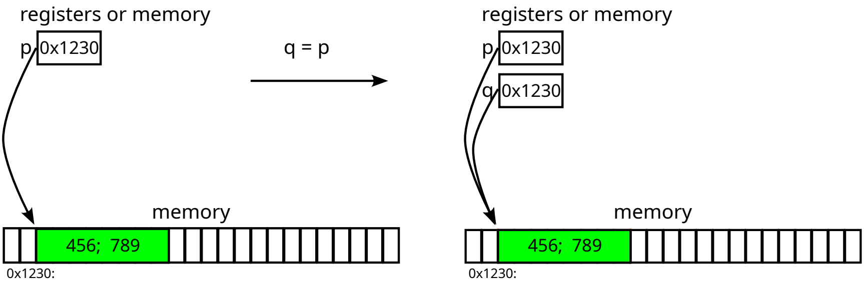
Indirect representation

- typically used for multi-word data
- upon an assignment-like operation, there are two choices
    1. (by-value) copies the whole data, or



Indirect representation

- typically used for multi-word data
- upon an assignment-like operation, there are two choices
  1. (by-value) copies the whole data, or
  2. (by-reference) copies only the address (*pointer*) and *share* data in memory

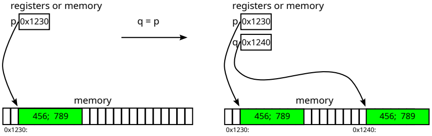


By-value vs. by-reference?

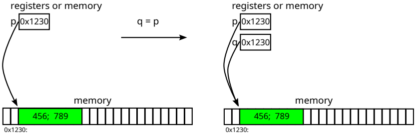
- it affects behavior (semantics) of *mutable* data; e.g.,

```
p = Point{x=456, y=789};
q = p;              // by-value or by-reference?
p.x = 1000;
print(q.x)          // 456 or 1000?
```

- therefore, for *mutable data, by-reference* is the only choice
- the choice does not affect the semantics of *immutable data*, so it is up to implementation



by-value



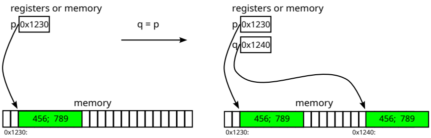
by-reference

Other data implemented typically passed-by-references

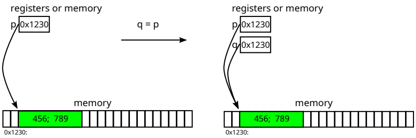
- besides mutable data, other data types whose assignment-like operations we want to implement by reference include
  - large data
  - recursive data
  - unknown-size data
- why? $\Rightarrow$ we don't want to impose large copying overhead whenever such values go through assignment-like operations
- for examples, strings, arrays, trees, graphs, etc.

The root of the problem

- were there no data implemented by reference, memory management problem would be largely non-existent
  - if a variable is gone, the data it points to is gone, too
- the difficulty arises as soon as data are *shared* (i.e., whose address may be held at multiple locations)
  - yet it is essential/unavoidable to implement mutable and/or implement large data efficiently, among others



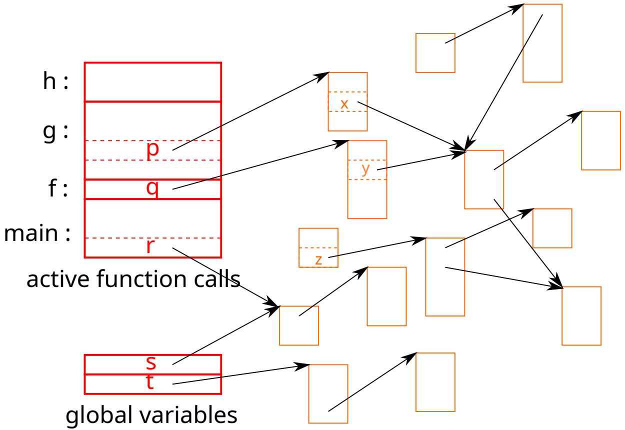
by-value



by-reference

The fundamental problem

- the problem is how to know which memory block can be safely reclaimed/reused when
  - there may be multiple pointers to a single memory block,
  - which allow arbitrary graph of memory blocks



A few remarks on "by-reference" vs. "by-value"

- some languages distinguish a data type (T) from a reference (pointer) to T
  - C/C++ : pointer (T`*`)
  - Go : pointer (`*`T)
  - Rust : box (`Box::<`T`>`) and refernce (`&`T)
- in other languages, there are no such distinction
  - OCaml, Julia, Python, etc.
- no matter what the language looks like from the programmer's perspective, the fundamental problem is the same
  - many (mutable, recursive, or large) data structures are passed *by reference*, leading to multiple references to a memory block

Contents

Memory allocation in C/C++

1. Global variables/arrays
2. Local variables/arrays
3. Heap

```
int g; int ga[10];
int foo() {
  int l; int la[10];
  int * a = &g;
  int * b = ga;
  int * c = &l;
  int * d = la;
  int * e = malloc(sizeof(int));
}
```

- lifetime

	starts	ends
global	when the program starts	when program ends
local	when a block starts	when a block ends
heap	malloc, new	free, delete

- note: the following discussion calls all of them *objects*

What could go wrong in manual memory management (e.g., C/C++)?

- heap-allocated (i.e., `new`/`malloc`'ed) memory must be `delete`/`freed` at the right spot
  - *premature free* = using it after `delete/free` → memory corruption

```
node * foo() {
  node * m = new node("Mimura");
  node * o = m;
  delete m;
  ... o->name ...
}
```

  - *memory leak* = not `delete/free`ing no-longer-used memory → (eventually) out of memory

```
node * foo() {
  node * m = new node("Mimura");
  node * o = new node("Ohtake");
  return o;
}
```

What could go wrong in manual memory management (e.g., C/C++)?

- stack-allocated memory are automatically reclaimed when it goes out of scope
  - using it afterwards $\equiv$ premature delete

```
node * foo() {
  node m = node("Mimura");
  node o = node("Ohtake");
  return &o;
}
```

```
node * foo() {
  node   m =     node("Mimura");
  node * o = new node("Ohtake");
  o->friend = &m;
  return o;
}
```

Tools to make C/C++ memory management safer

- `valgrind` (memory checker)
  - detect memory-related errors (use after free, memory leak, out of bound accesses, etc.)
- Boehm garbage collection library for C/C++
  - automatically garbage-collect memory blocks allocated by malloc/new

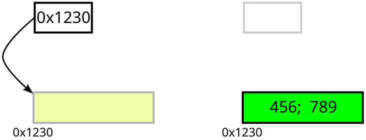
Note : it is not a *pointer* that is to blame

- C/C++ are notriously unsafe languages
- a common misconception is they are unsafe *because they expose pointers* to the programmer
- sure, many features that make C/C++ unsafe are related to pointers in one way or another,
- yet this is a misconception because
  - eliminating pointers from the surface of a language does not solve the memory management problem, and
  - languages exposing pointers can be made safe

Contents

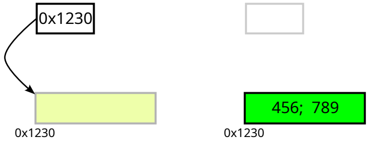
Garbage Collection (GC)

- the fundamental problem 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)

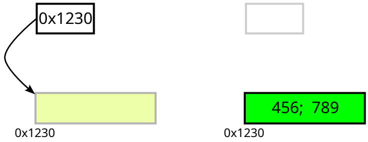
- the fundamental problem issue is the mismatch between
  - the period in which objects are accessed
  - the period in which the memory block for it is retained



- ⇒ 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 problem 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...

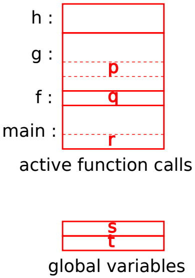
```
1 int main() {
2   if (f(x) == 0) {
3     printf("%d\n", p->f->x);
4   }
5 }
```

- $\rightarrow$ *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)

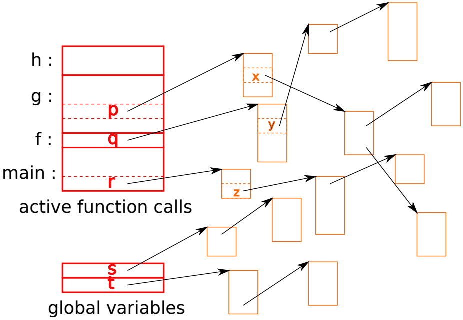
```
int * s, * t;
void h() { ... }
void g() {
  ...
  h();
  ... = p->x ...  }
void f() {
  ...
  g()
  ... = q->y ... }
int main() {
  ...
  f()
  ... = r->z ... }
```



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 ... }
```



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

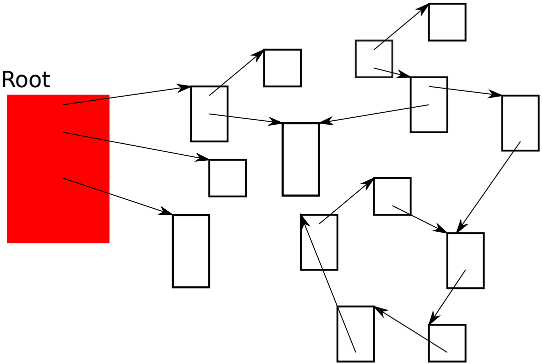
> the basic principle of GC:
> objects unreachable from the root are dead

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 $\rightarrow$ it's unreachable from the root
- remark: "GC" sometimes narrowly refers to traversing GC

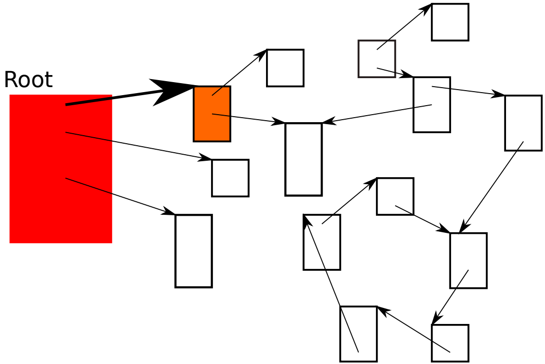
How traversing GC works

- 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



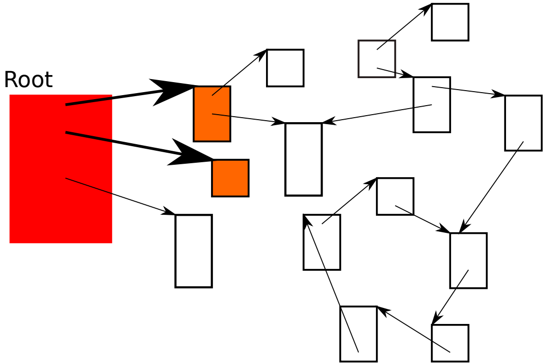
How traversing GC works

- 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



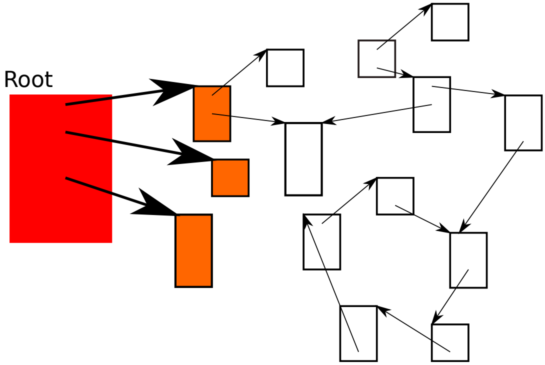
How traversing GC works

- 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



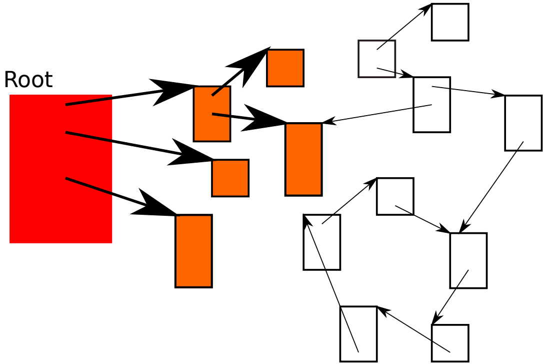
How traversing GC works

- 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



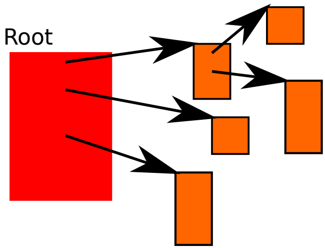
How traversing GC works

- 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



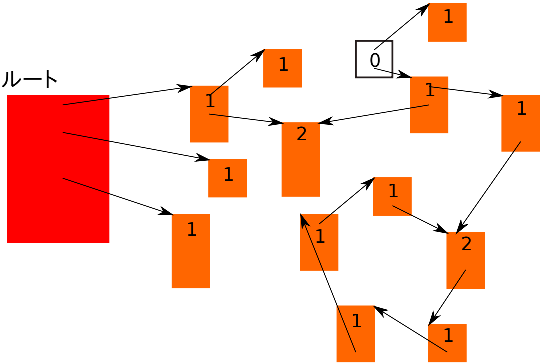
How traversing GC works

- 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



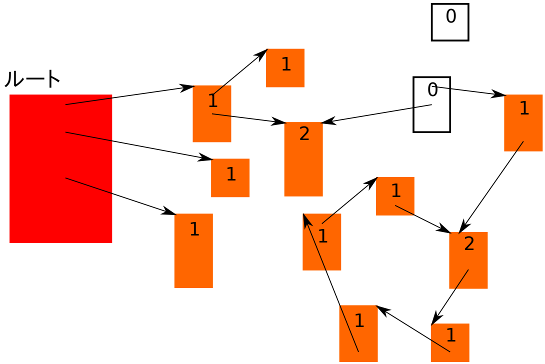
How reference counting works

- 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



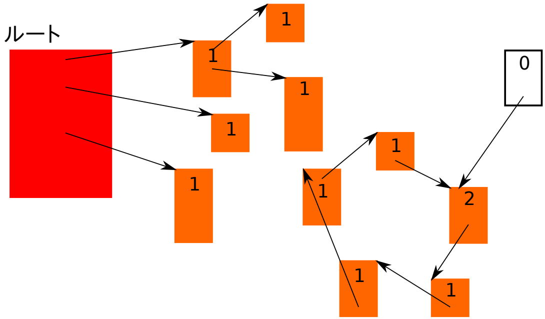
How reference counting works

- 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



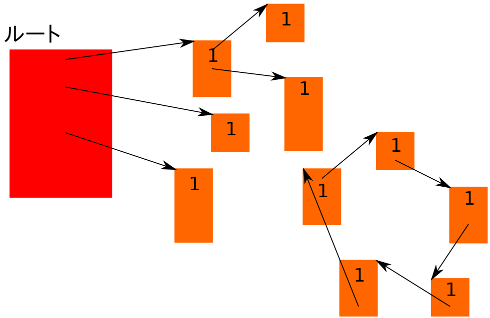
How reference counting works

- 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



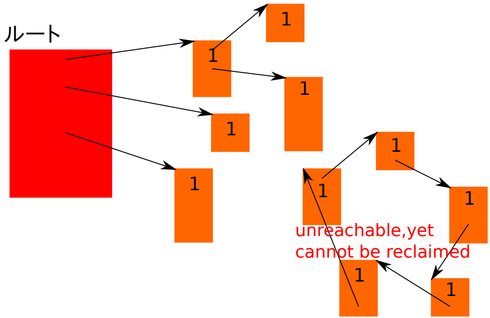
How reference counting works

- 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

GC will be covered more deeply in later weeks