# Programming Languages (5)
# Memory Management Introduction

Kenjiro Taura

# Contents

# Contents

# Memory management in programming languages

- all values (integers, floating point numbers, strings, arrays, structs, . . . ) need memory to hold them
- ideally, programming languages manage them on behalf of the programmer
- three approaches covered

| | | |
|---|---|---|
| manual | | C, C++ |
| garbage collection | traversing<br>reference counting | Python, Java, Julia, Go, OCaml, etc. |
| Rust ownership | | Rust |

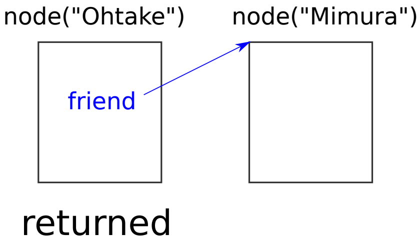
# Memory Management Quiz

take the quiz via any of the following

- direct link
- go menti.com and enter code 4574 1905
- use this QR Code

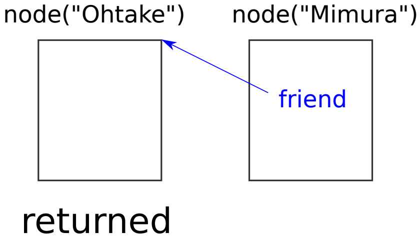
# Illustration (Q2)

```
def foo():
    m = node("Mimura")
    o = node("Ohtake")
    o.friend = m
    return o
```



# Illustration (Q3)

```
def foo():
    m = node("Mimura")
    o = node("Ohtake")
    m.friend = o
    return o
```



# Contents

# Memory allocation in C/C++

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

```c
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`/`free`d at the right spot
  - *premature free* = using it after `delete/free` → memory corruption
  - *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 ≡ 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->frien = &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

# Contents

# Garbage Collection (GC)

- the fundamental problem of manual memory management is the mismatch between the actual "lifetime" of objects and "the period in which they are accessed"
  - you may access an object after its lifetime
  - you may not free an object despite you no longer access it

# Garbage Collection (GC)

- the fundamental problem of manual memory management is the mismatch between the actual “lifetime” of objects and “the period in which they are accessed”
  - you may access an object after its lifetime
  - you may not free an object despite you no longer access it
- $\Rightarrow$ Garbage collection (GC)
  - keep objects alive if they could ever be accessed in future and reclaim otherwise
  - the system automatically does that
  - $\Rightarrow$ eliminate memory leak and corruption

# Garbage Collection (GC)

- the fundamental problem of manual memory management is the mismatch between the actual "lifetime" of objects and "the period in which they are accessed"
  - you may access an object after its lifetime
  - you may not free an object despite you no longer access it
- ⇒ Garbage collection (GC)
  - keep objects alive 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" → you need to be able to solve the halting problem...

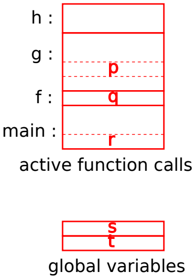
```
1 int main() {
2   if (f(x) == 0) {
3     printf("%d\n", p->f->x);
4   }
5 }
```

- → *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)

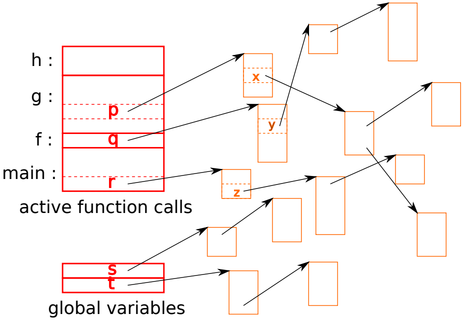
```
int * s, * t;
void h() { ... }
void g() {
  ...
  h();
  ... = p->x ...  }
void f() {
  ...
  g()
  ... = q->y ... }
int main() {
  ...
  f()
  ... = r->z ... }
```



# 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

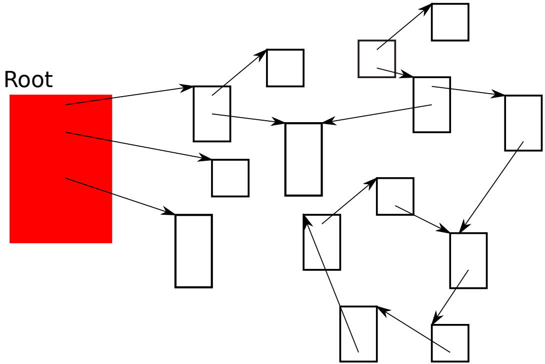
the basic principle of GC:
objects unreachable from the root are dead

# 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 → it's unreachable from the root
- remark: "GC" sometimes narrowly refers to traversing GC

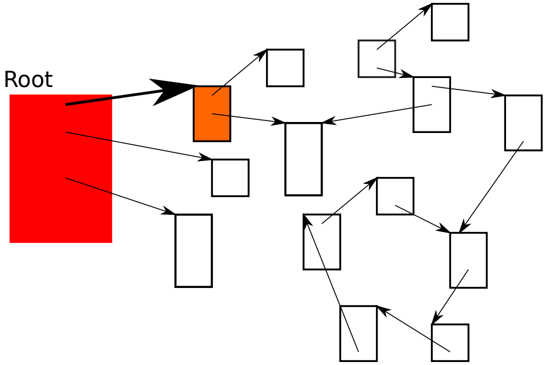
# How traversing GC works

- 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



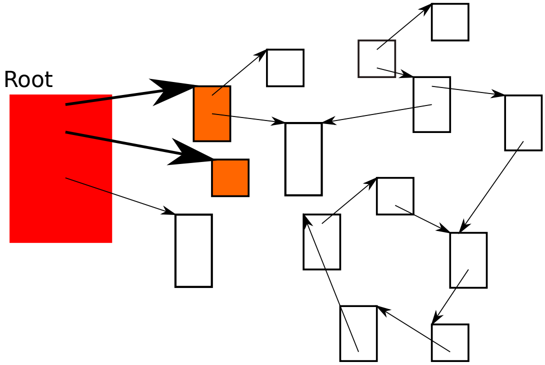
# How traversing GC works

- 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



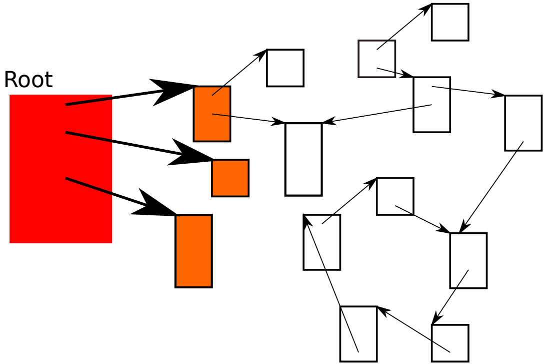
# How traversing GC works

- 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



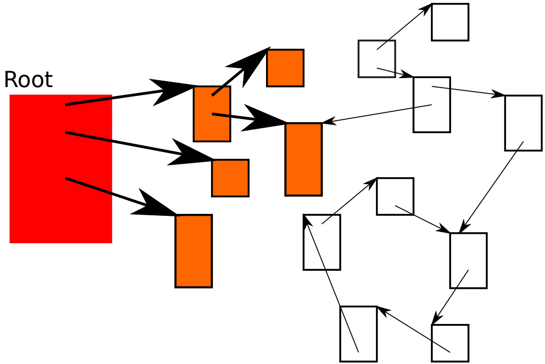
# How traversing GC works

- 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



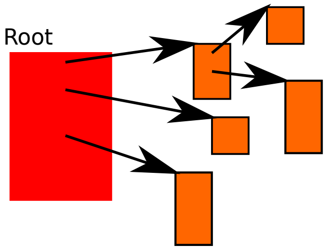
# How traversing GC works

- 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



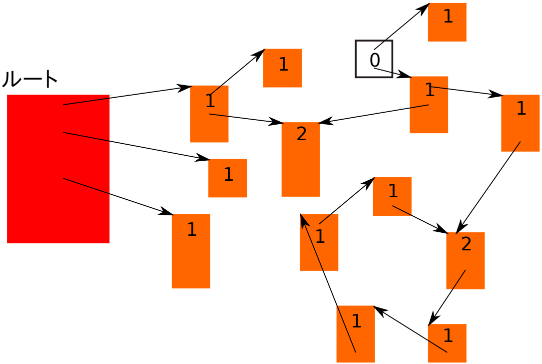
# How traversing GC works

- 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



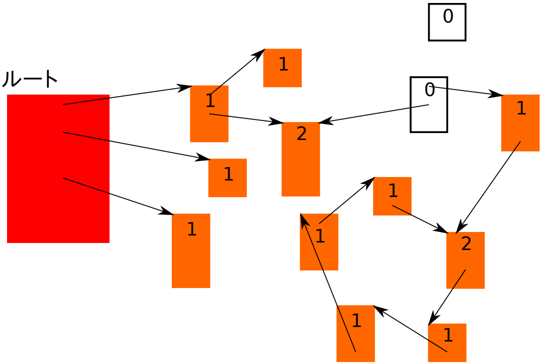
# How reference counting works

- 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



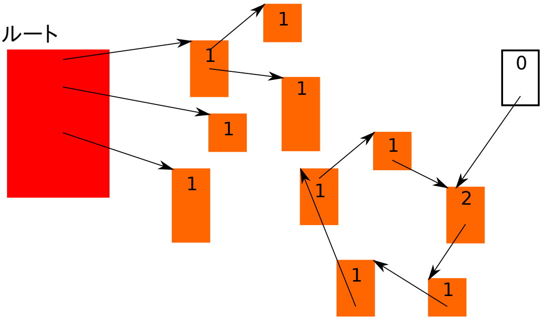
# How reference counting works

- 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



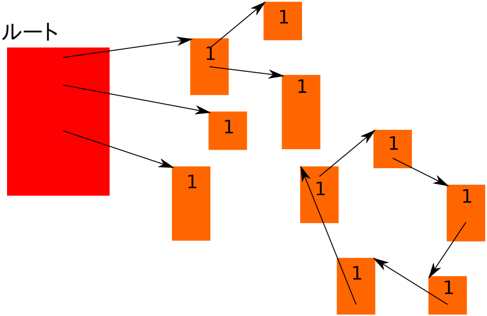
# How reference counting works

- 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



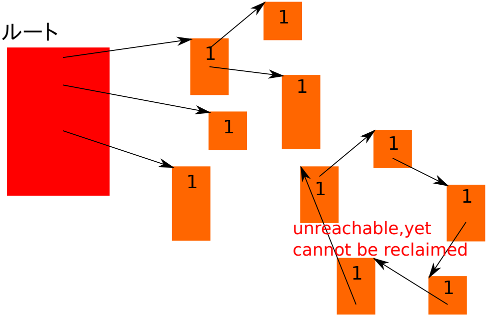
# How reference counting works

- 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