Memory Management Introduction

Kenjiro Taura

2024/05/19

Contents

Introduction

What is memory management?

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
- desirably, programming languages should *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”

Memory management in programming languages

memory management is mainly about how to determine when the space (memory block) occupied by data can be safely reclaimed/ reused

Approaches covered

- manual … C, C++
- ***garbage collection (GC)***
  - ***traversing GC*** … Python, Java, Julia, Go, OCaml, etc.
  - ***reference counting*** … Python, etc.
- Rust ***ownership system*** … Rust

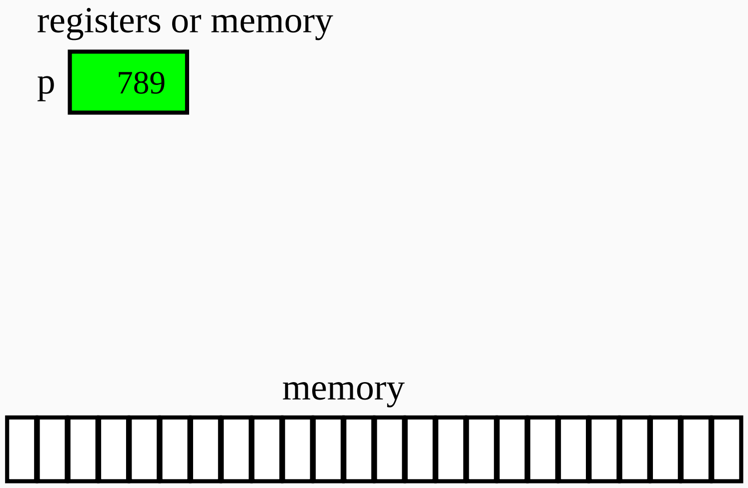
Data representation

Data representation

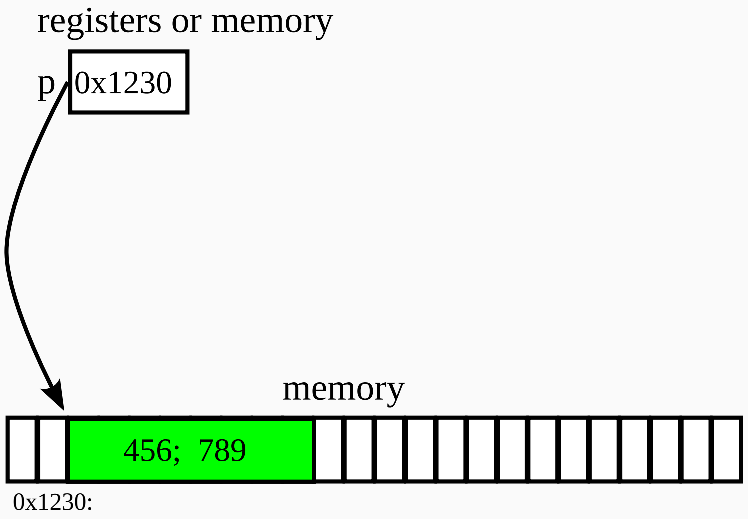
- data in your program must be somehow *represented* and laid out in registers and/or memory
  - primitive data (booleans, characters, integers, floating point numbers, …)
  - multiword data (structs),
  - dynamically-sized or large data (e.g., arrays and strings),
  - recursive data (lists, trees, graphs, etc.),
  - etc.

Two strategies

immediate (unboxed) representation

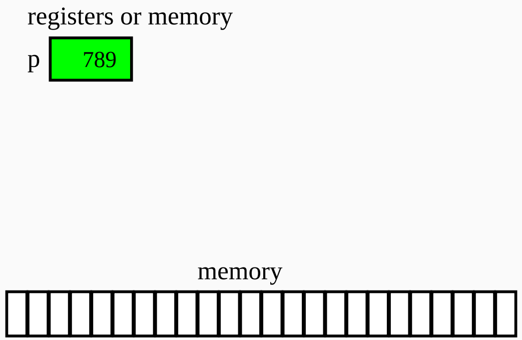


indirect (boxed) representation



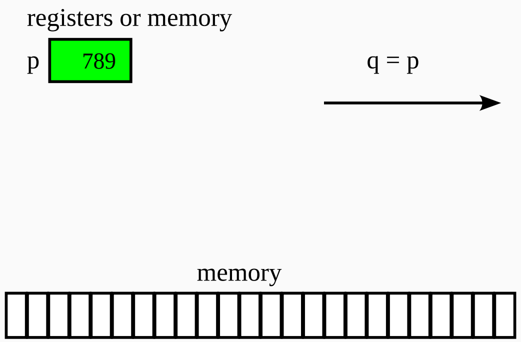
Immediate (unboxed) representation

- typically used for small data that fit one or a few machine words (integers, floats, characters, small structs, etc.),



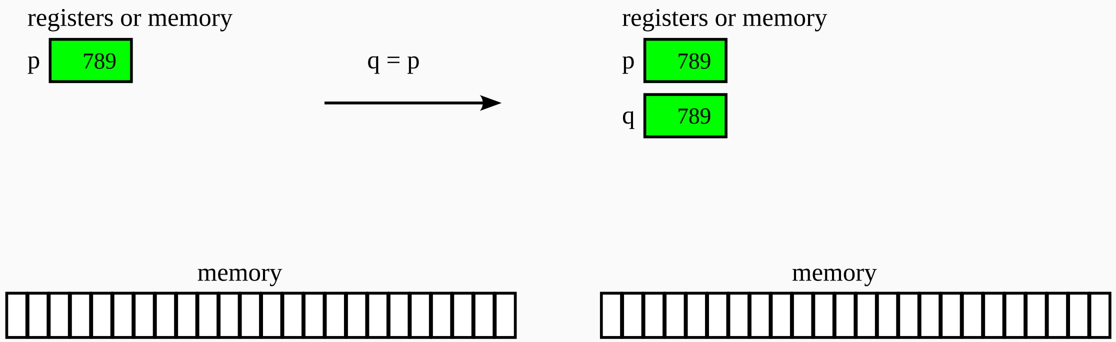
Immediate (unboxed) representation

- typically used for small data that fit one or a few machine words (integers, floats, characters, small structs, etc.),
- upon an assignment-like operation, the whole data gets copied (cheap as data are small)



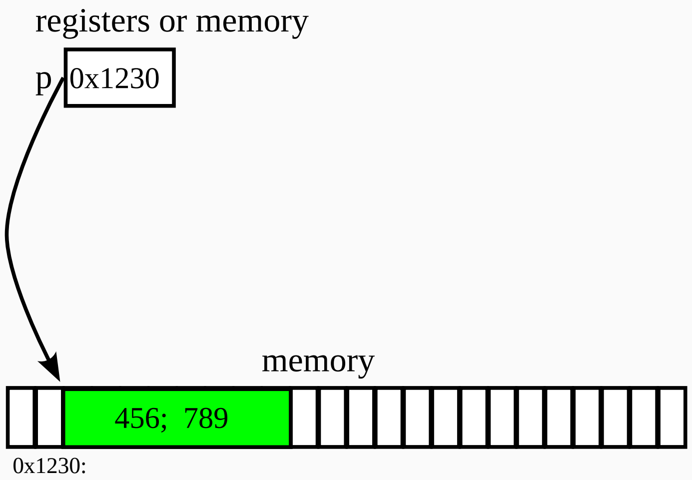
Immediate (unboxed) representation

- typically used for small data that fit one or a few machine words (integers, floats, characters, small structs, etc.),
- upon an assignment-like operation, the whole data gets copied (cheap as data are small)



Indirect (boxed) representation

- used for all other data
  - dynamically-indexed (string, arrays, etc.)
  - dynamically-sized (string, arrays, etc.)
  - recursive data (list node, tree node, graph node, etc.)
  - large data

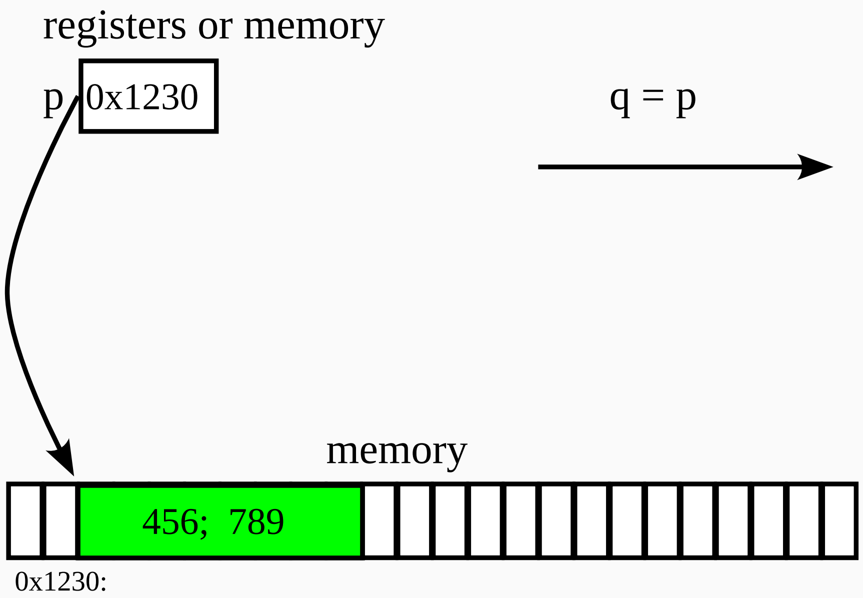


Assignment of indirectly represented data

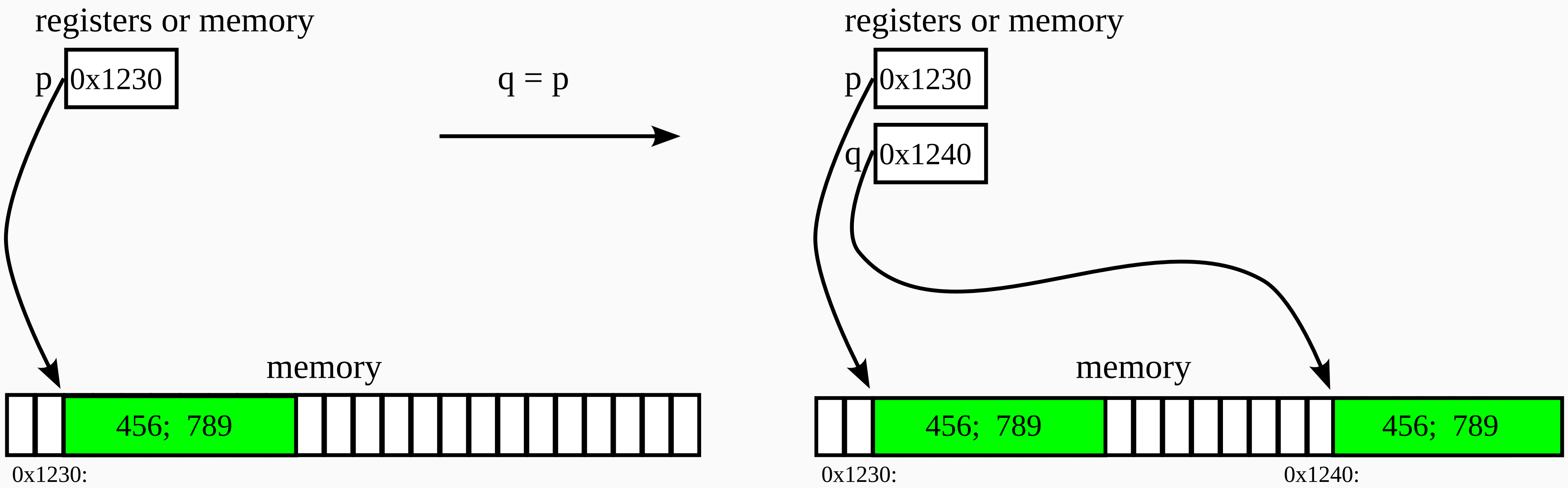
- upon an assignment-like operation of indirectly represented data, there are two choices:

1. ***copy-by-value:*** allocates memory and copies the data
2. ***copy-by-reference:*** copies the address ***(pointer)*** and *shares* data in memory

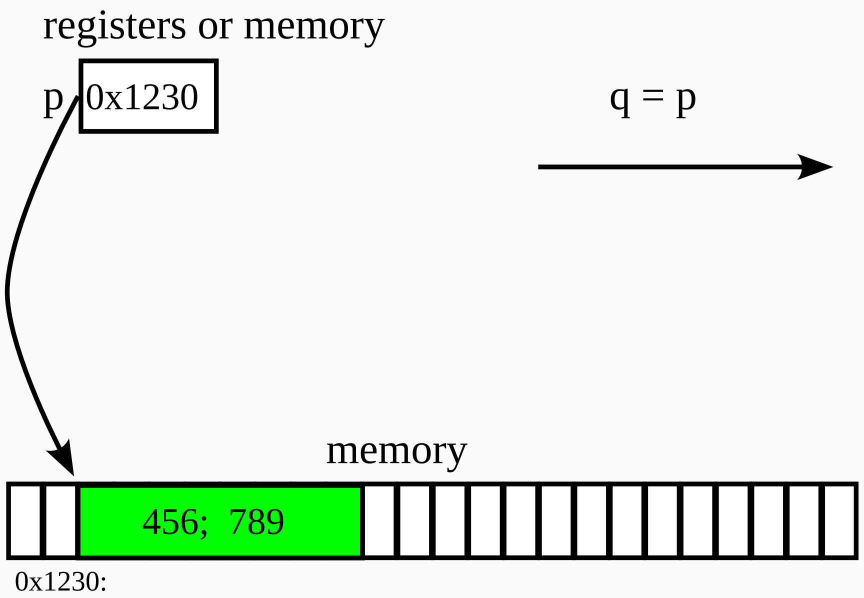
Copy-by-value



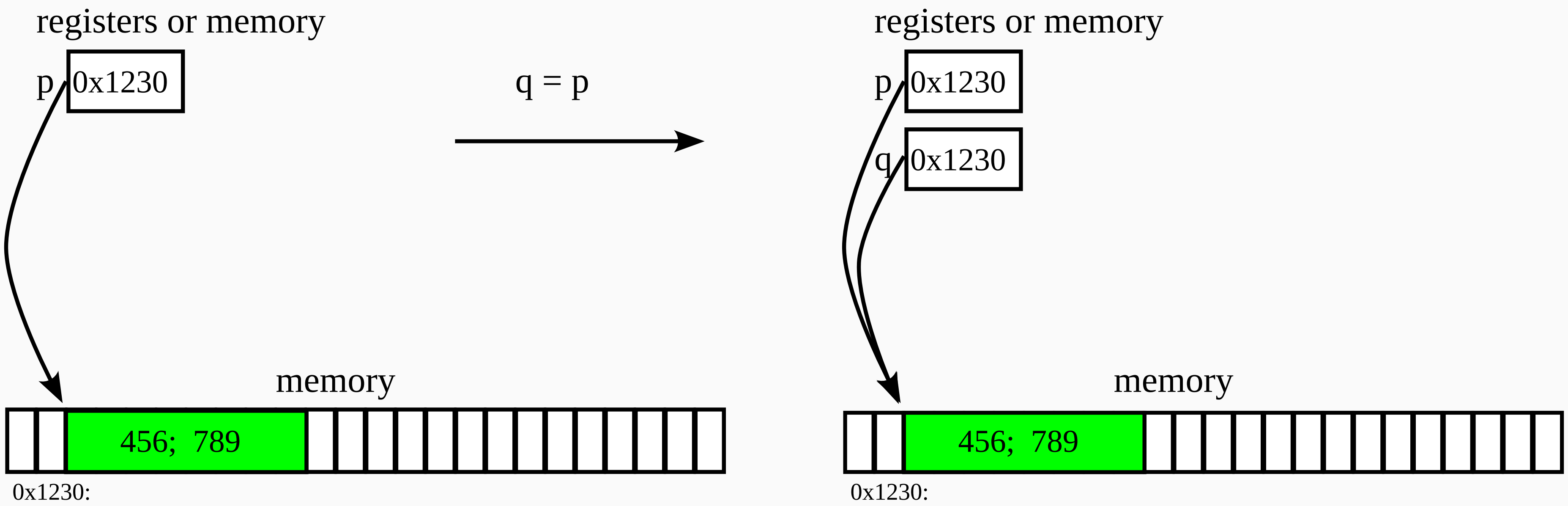
Copy-by-value



Copy-by-reference



Copy-by-reference



Copy by-value vs. by-reference

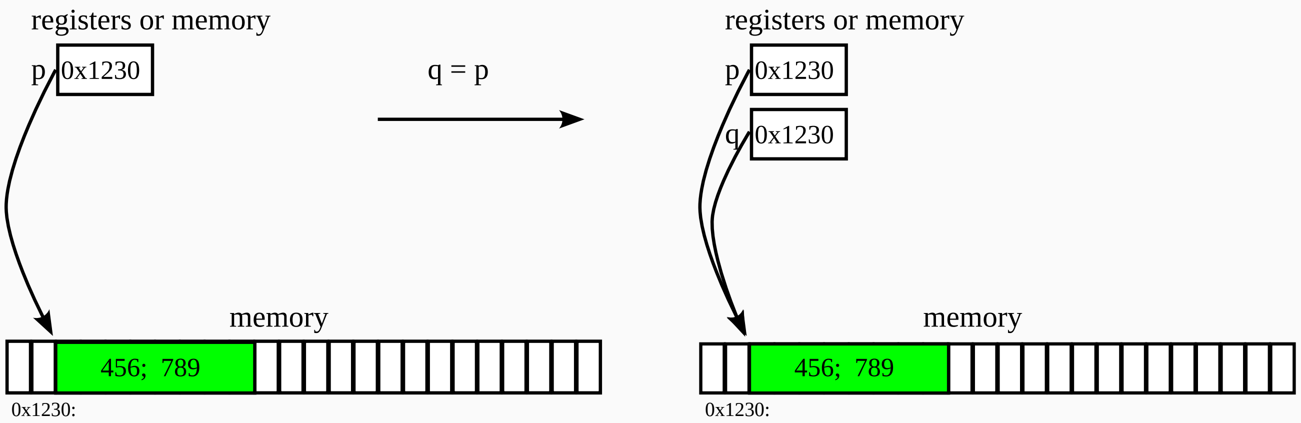
- besides the cost of copy, it affects *behavior (semantics)* of *mutable* data

```
a = point(x: 10, y: 20)
b = a          # copy-by-value? or reference?
b.x = 100
print(a.x)     # 10 if by-value, 100 if by-reference
```

- if the language spec says it should print 100 in this program, `point` objects should be copied by reference

A terminology note

- many programming languages employ this semantics for all mutable data and therefore implement them by copy-by-reference
- we casually say such data is implemented by ***pointer***



Why memory management is difficult at all?

- were there no data implemented by copy-by-reference, memory management problem would be largely non-existent
- $\because$ if all data were immediate or copied upon assignment
  - $\Rightarrow$ two pointers never point to the same memory block
  - $\Rightarrow$ if a pointer is gone (e.g., a pointer variable goes out of scope), the memory block it points to (and all data reachable from it) can be safely reclaimed

Why memory management is difficult at all?

- it is difficult precisely because some data are (*and must be*) implemented by copy-by-reference
  - ⇒ the same memory block may be pointed to by multiple references
  - ⇒ even if a pointer is gone, other pointers may still exist and data may still be used

Memory management in C/C++

Three types of memory in C/C++

- global variables/arrays (defined at the toplevel)
- local variables/arrays (define inside a function)
- heap (malloc, new)

```
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

- ***lifetime*** of a memory block (variable, array, heap-allocated block) refers to the period in which it is valid (i.e., remembers the last-written data)

	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 discussion below calls memory blocks *objects*

What can go wrong in C/C++ (stack-allocated objects)

- unconditionally reclaimed when it goes out of scope
- yet there may be a pointer still pointing to it

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

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

What can go wrong in C/C++ (heap-allocated objects)

- lifetime ends with and only with free/delete by the programmer

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

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