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

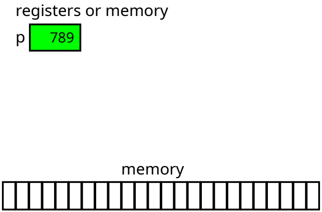
manual		C, C++
garbage collection	traversing reference counting	Python, Java, Julia, Go, OCaml, etc.
Rust ownership		Rust

Data representation

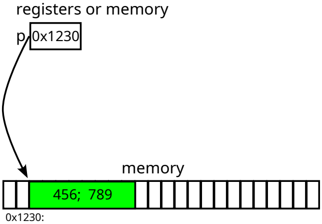
- data in your program must be somehow represented in the machine code
- some data (e.g., integers and floating point numbers) can be trivially mapped to machine representations
- less trivial is how to map
  - multiword data (structs),
  - unknown-size or large data (e.g., arrays and strings),
  - mutable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

Two strategies

- immediate

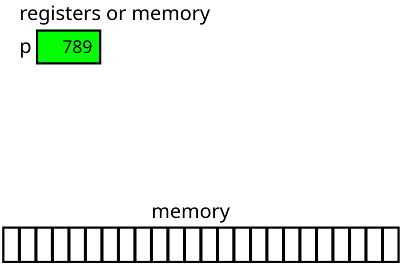


- indirect



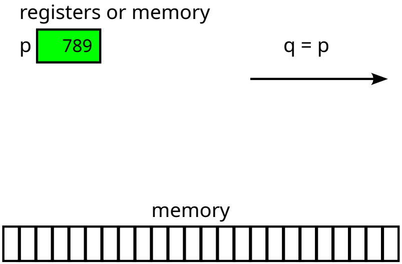
Immediate representation

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



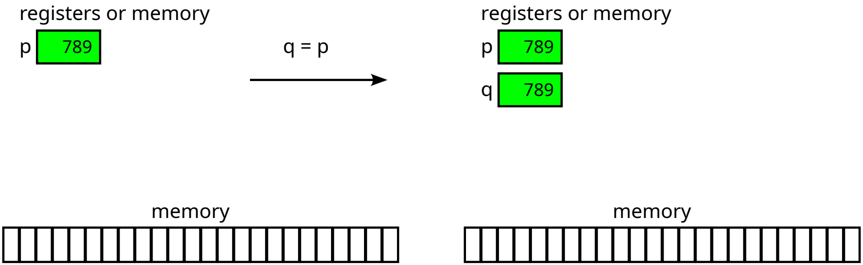
Immediate representation

- 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)



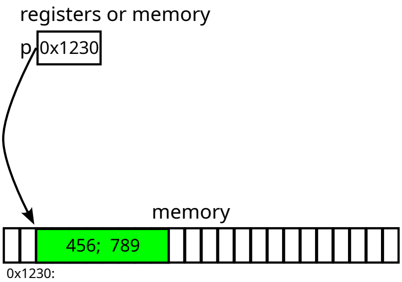
Immediate representation

- 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)



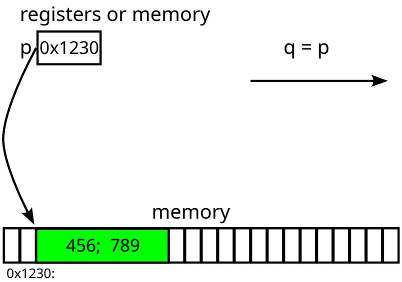
Indirect representation

- typically used for multi-word data



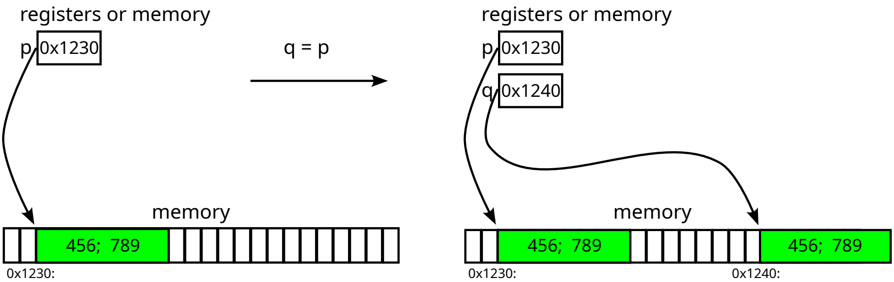
Indirect representation

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



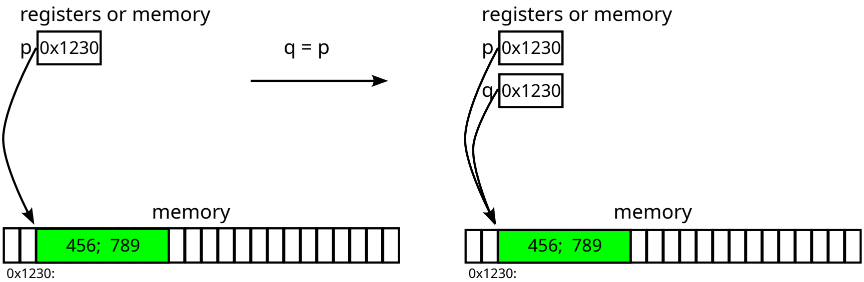
Indirect representation

- 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

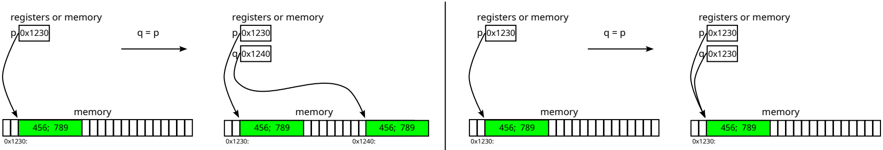


By-value vs. by-reference?

- 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



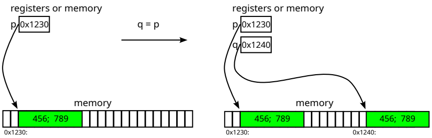
by-value | by-reference

Other data implemented typically passed-by-references

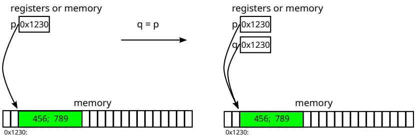
- besides mutable data, other data types whose assignment-like operations we want to implement by reference include
  - large data
  - recursive data
  - unknown-size data
- why? $\Rightarrow$ we don't want to impose large copying overhead whenever such values go through assignment-like operations
-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



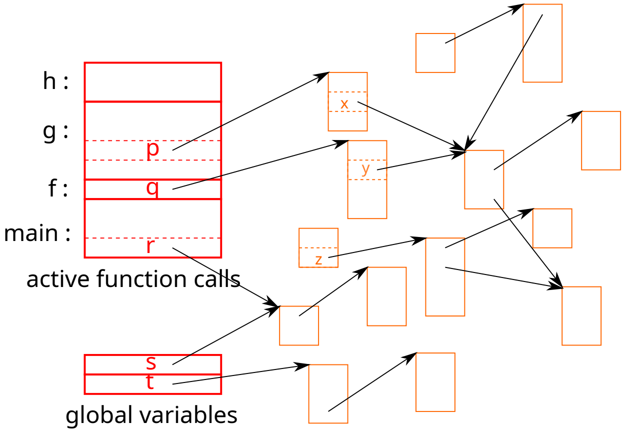
by-value



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`/`free`d 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 ≡ 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