

Programming Languages (6)

Memory Management

Kenjiro Taura

Contents

1 Introduction

2 Manual Memory Management in C/C++

Contents

1 Introduction

2 Manual Memory Management in C/C++

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

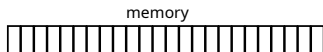
- data in your program must be somehow represented in the machine code
- some data (e.g., integers and floating point numbers) can be trivially mapped to machine representations
- less trivial is how to map
 - ▶ multiword data (structs),
 - ▶ unknown-size or large data (e.g., arrays and strings),
 -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

Two strategies

- immediate

registers or memory

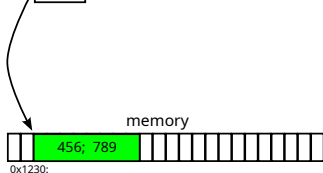
p 789



- indirect

registers or memory

p 0x1230



Immediate representation

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

registers or memory

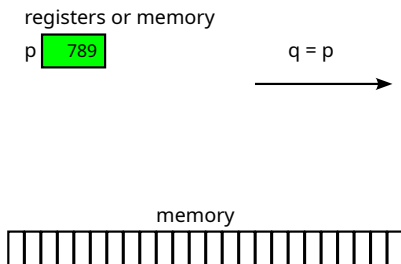
p 789

memory



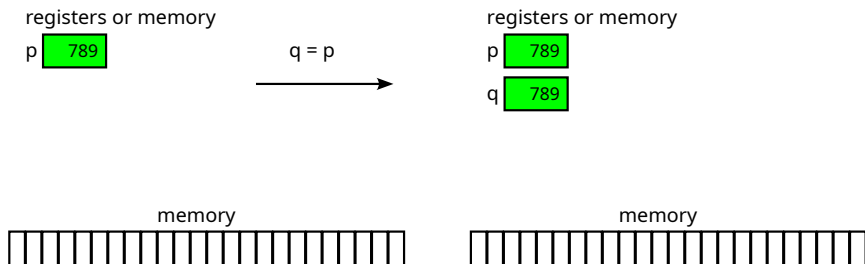
Immediate representation

- 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)



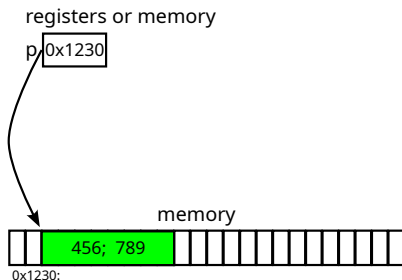
Immediate representation

- 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)



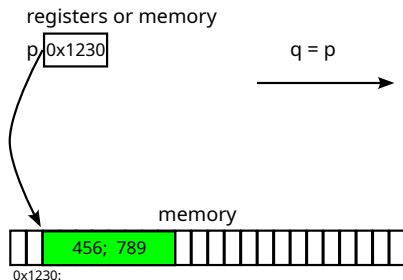
Indirect representation

- typically used for multi-word data



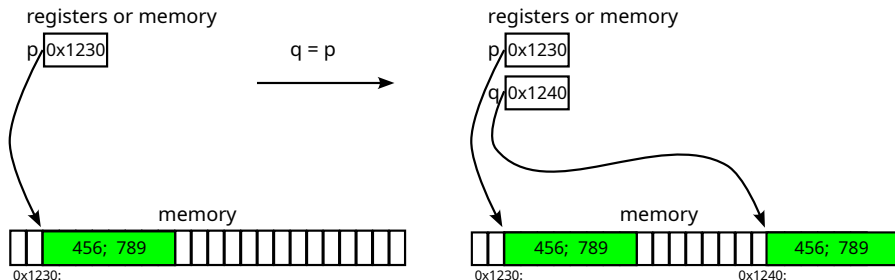
Indirect representation

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



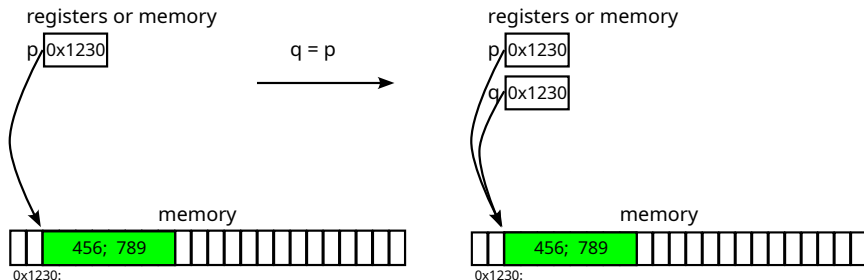
Indirect representation

- 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

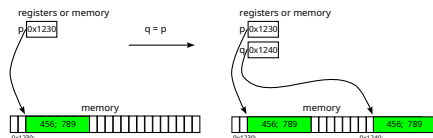


By-value vs. by-reference?

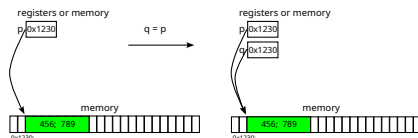
- it affects behavior (semantics) of *mutable* data; e.g.,

```
1 p = Point{x=456, y=789};  
2 q = p;           // by-value or by-reference?  
3 p.x = 1000;  
4 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



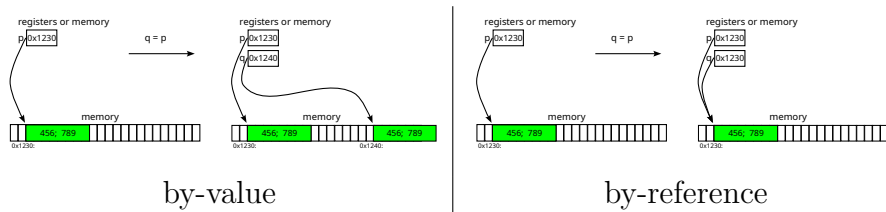
by-reference

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.

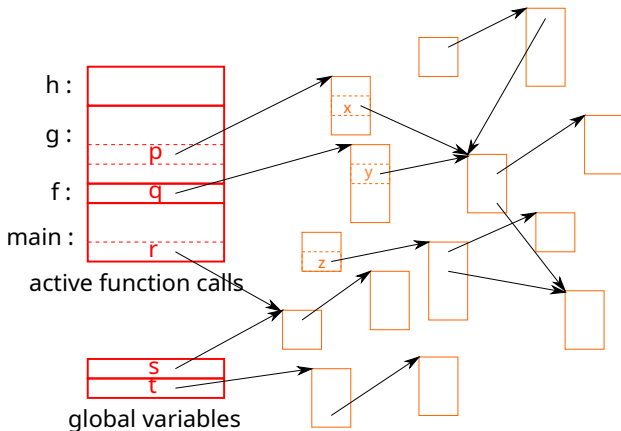
The root of the problem

- 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



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 ($\text{Box}::\langle T \rangle$) and reference ($\&T$)
- 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

1 Introduction

2 Manual Memory Management in C/C++

Memory allocation in C/C++

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

```
1 int g; int ga[10];  
2 int foo() {  
3     int l; int la[10];  
4     int * a = &g;  
5     int * b = ga;  
6     int * c = &l;  
7     int * d = la;  
8     int * e = malloc(sizeof(int));  
9 }
```

- 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

```
1 node * foo() {  
2     node * m = new node("Mimura");  
3     node * o = m;  
4     delete m;  
5     ... o->name ...  
6 }
```

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

```
1 node * foo() {  
2     node * m = new node("Mimura");  
3     node * o = new node("Ohtake");  
4     return o;  
5 }
```

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

```
1 node * foo() {  
2     node m = node("Mimura");  
3     node o = node("Ohtake");  
4     return &o;  
5 }
```

```
1 node * foo() {  
2     node    m =     node("Mimura");  
3     node * o = new node("Ohtake");  
4     o->friend = &m;  
5     return o;  
6 }
```

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 notoriously 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