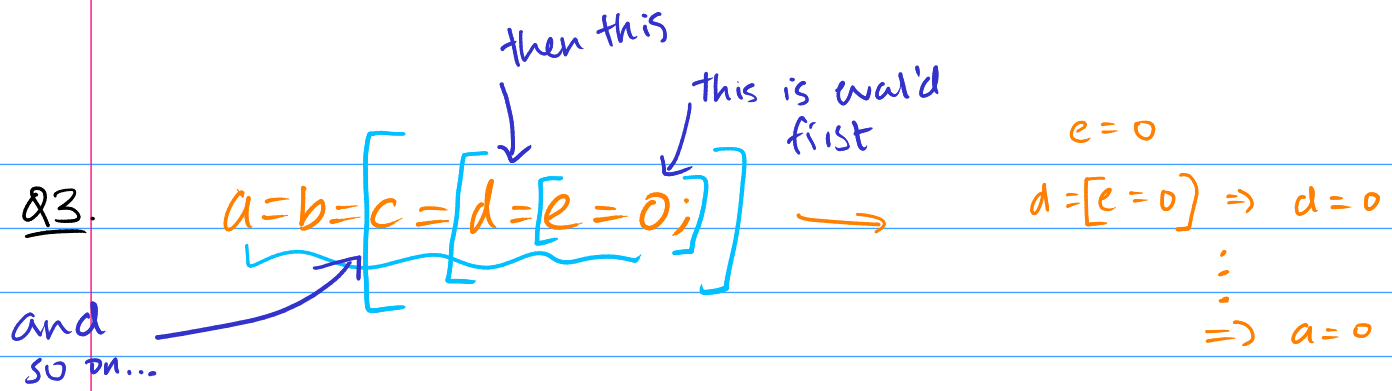


Each assignment expression evaluates to 0, so the next variable in the chain going from right to left also gets set to 0.
In the end, all are set to 0.

In general, the assignment

`e=v;`

evaluates to

`v`.

Just be careful with pointers.

A student asked...

Could we use an array instead?

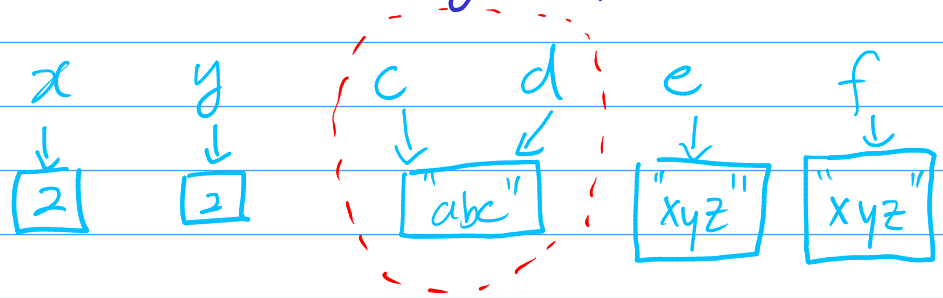
Yes, but we would lose semantic meaning conveyed by variable names!

We could do `int vars[5] = {0}` to create 5 zero initialised "variables", but

Compare { count, sum, foo } with { vars[0], vars[1], vars[2] }

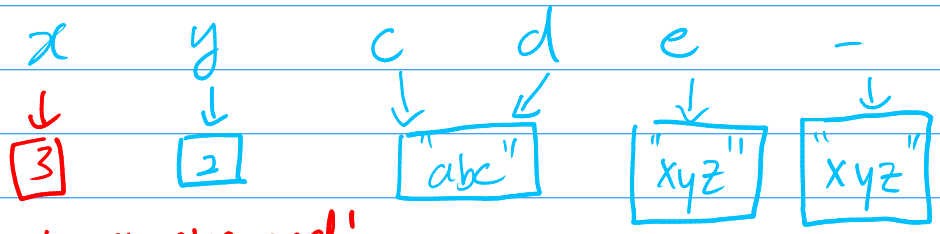
At first glance, what does `vars[0]` mean? We don't know unless we wrote the code! On the other hand, `count` is more readable.

Q4. State of memory after the first block:



Note that c and d point to the same string in memory!

Now, after the operation x++ it looks like this:



only x changed!
x=y=2 does not mean x and y refer to the same 2 in memory.

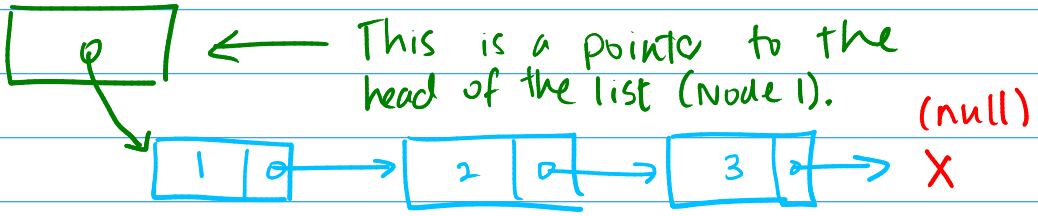
Everything else is the same.

But when we do `*c = 'A';` the program crashes?
Why? The string "abc" lives in read-only memory, since it is a pointer string (or string literal).

[Remark: If we had `char c[3] = {'a', 'b', 'c'}` instead, then `*c = 'A'` would work as it is a character array instead of a pointer string.]

Q13. Representation 1 looks like this:

Representation 2 looks like this:



Why is rep. 2 better in some cases?

- It makes functions that operate on lists simpler. If we want to implement an insertion function with rep 1 we have to return a list, because the head of the list might change. But in rep 2, we just need to update the list's head pointer internally - the list pointer itself is unchanged!

- It also allows us to add other useful info; consider also things like list size and a tail pointer! They can be helpful to have access to straight up (as we will soon see).

There is a disadvantage though - the more extra things we add, the more things we have to make sure are still correct.