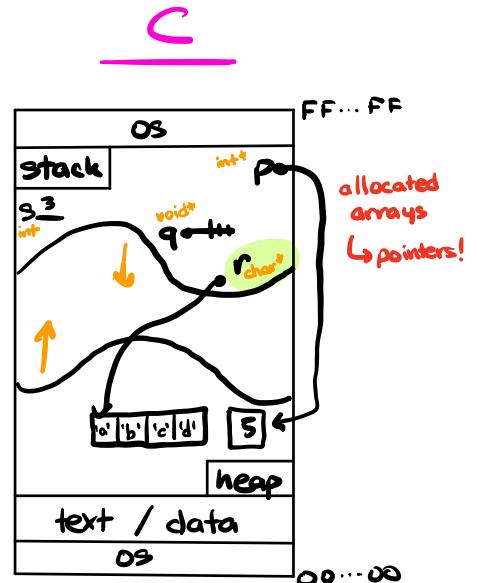
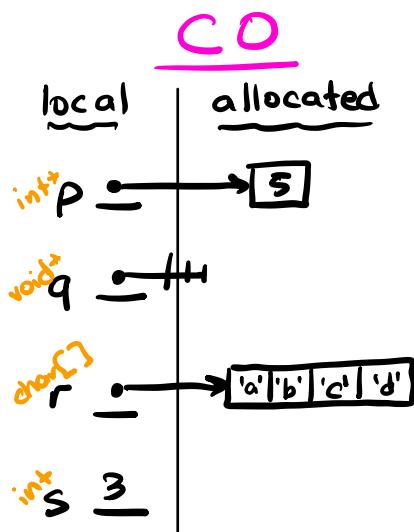


# C Memory Model



readonly! →  
data = string literals

text = program code

## allocating memory

### allocating a pointer ...

... in CD: `type* p = alloc(type);` (initializes to default value)  
 ... in C: `type* p = malloc(sizeof(type));` (not initialized!)  
or `type* p = calloc(sizeof(type), 1);` (initialized to 0)

### allocating an array ...

... in CD: `type[] a = alloc_array(type, n);` (initialized)  
 ... in C: `type* a = malloc(sizeof(type)*n);` (not initialized!)  
or `type* a = calloc(sizeof(type), n);` (initialized)

### Keep in mind:

- `sizeof(type)` gives the number of bytes needed to store type
- `malloc/calloc` can return `NULL`; use `xmalloc/xcalloc` or do `NULL` checks
- `malloc` is faster than `calloc` since it doesn't initialize the bytes
- values in a `malloced` block are undefined; overwrite it before use
- **freeing!**

## freesing memory

if you allocate it, you free it (mostly\*)

- no garbage collection in C!
- don't double free the same memory location
- good practice to not free NULL
- if there is some kind of nested structure, free the inside first!



what happens when we call `free(p)`?

it frees node 1, but nodes 2 and 3 remain unfreed and lost since they cannot be accessed :c

- use data structure free functions if available

now why the (mostly\*) up there?

- sometimes, we hand off ownership of memory to the client (like in task 5 of COVM!) so may not want to free
- or we may want to keep using the memory

## Stack stuffs

### structs

- can have structs on the stack in C

heap: `struct smth* s = malloc(sizeof(struct smth));`

stack: `struct smth s; (not initialized!)` → same as `(*s).x`

- on heap, access fields with → (like `s->x`)

- on stack, access fields with • (like `s.x`)

### arrays

- can have stack arrays in C

stack: `int a[4] = {1, 2, 3, 4};`

or `int a[] = {1, 2, 3, 4, 5, 6};` (length is inferred)

`int a[6]; (not initialized!)`

### strings ...

## String stuffs

strings are just null-terminated char arrays!

### stack

char str[] = "honk"; (auto null-terminated)  
or char str[] = {'c', 'h', 'o', 'n', 'k', '\0'} (not auto null-term)

### heap

char\* str = xcalloc(sizeof(char), n);  
... char assignments...

ex) what is the max length string we can safely put in str?  
n-1, save space for null-terminator!!

### data

char\* str = "122"; (auto null-term)  
↑ this is read-only!

## Casting + pointer arithmetic

- different from integer casting:  
does not actually change any data ever!
- just looks at memory differently

tip: casting to  
char\* is good  
for stepping  
through bytes

ex)  $\text{char } *A = 0x70;$

how many bytes does \*A read?

1, since char

$*(\text{short } *)A$ ?    $*(\text{int } *)A$ ?    $*(\text{void } *)A$ ?  
2                  4                  error! can't deref void\* :)



- type also affects pointer arithmetic

ex)  $\text{char } *A = 0x70;$

what is A+1?   A-3?   A+2?    $(\text{int } *)A + 2$ ?    $((\text{long } *)A) + 2$ ?  
0x71            0x6C            0x72            0x78            0x80

- $A[i] = *(A+i)$

## address - of

- for any variable  $x$ ,  $\&x$  is the memory address  $x$  is located at even for pointers! this is the location of the pointer, NOT the location it points to !!

- if  $x$  has type  $t$ ,  $\&x$  has type  $t^*$
- useful to allow functions to change stack/local memory

ex) void fun (int<sup>\*</sup> x) {

    \*x = 122;

}

what is the value of  $x$  after this?

int x = 150;

122

fun (&x);

be careful of scope !!

do NOT do this:

```
int *terrible () {  
    int horrible = 92;  
    return &horrible;  
}
```

horrible goes out of scope  
at the end of the function,  
so the return value is basically  
useless