



Pointers are
used for lots of
things in C!



Pointers are used for ...



Pointing to a **location** in memory

Pointing to a contiguous series of
locations in memory (i.e. **arrays**)

(Memorable) data types, e.g., **char *** as
pointer representation for **strings**

Passing in **modifiable references** to an
object (*pointers to variables, double ptrs*)

Addresses of **dynamically allocated
memory** (pointers returned by **malloc**)

C is Pass by
Value

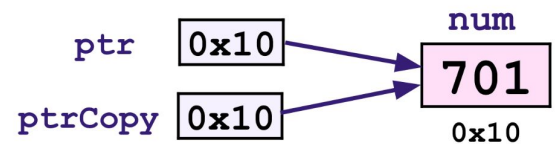
When passing parameters, **C passes a copy of whatever is passed.**

- ☐ To change values, and have those changes persist outside the function, **we must** **pass a pointer to the value** (i.e. its address in memory).
- ☐ For instance, to change an **int**, we have to pass in an **int *** (pointer to an int)
- ☐ To change a **char *** (string), we have to pass in a **char **** (pointer to string).

Short Example

```
function mischief(int* ptrCopy) {
    *ptrCopy = 701;
}

int main() {
    int num = 107;
    int* ptr = &num;
    mischief(ptr);
}
```



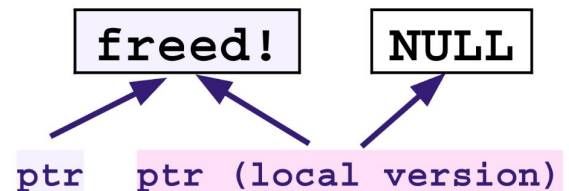
Though the pointer is itself duplicated, **it's still pointing to the same value** in memory, and so **we can directly modify that value!**

Passing in
Pointer
Exercise

Your ~~Sith Lord~~ coworker suggests that you use this function below as a **"safer" free function**, to prevent accidentally using **ptr** from that point forward (by also making it **NULL**).



```
void free_and_null(void *ptr){
    free(ptr);
    ptr = NULL;
}
```



- Will this function correctly **free** the client's pointer?

Yes, as our both pointers point to the same block of memory, so it'll be freed.

- Will correctly set it to **NULL**?

No; it only sets **free_and_null's own copy of the ptr** to **NULL**. You would have to pass a pointer to the client's pointer, to **NULL** it out.



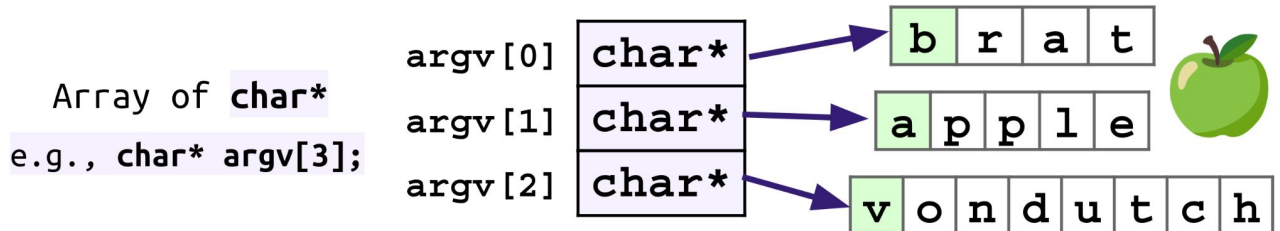
Arrays and Pointers



- When passing **arrays as parameters**, we always pass them as a **pointer to the first element**

`play(album);` `char album[5];`
b r a t \n

- Array of pointers** = array where each element is a pointer to some data.
- For example, an **array of strings** stores **pointers to their first characters**, NOT the characters themselves. Those can be stored elsewhere in memory

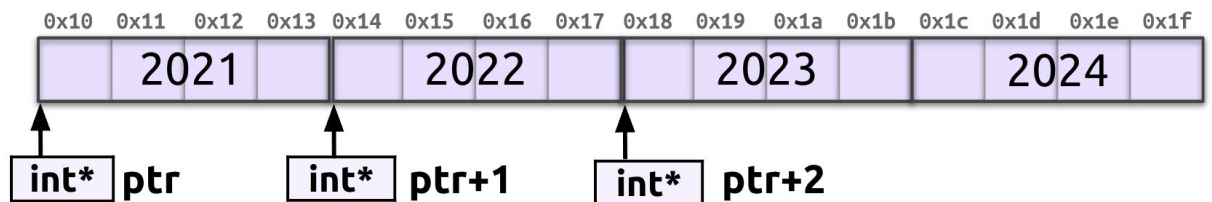


Pointer Arithmetic

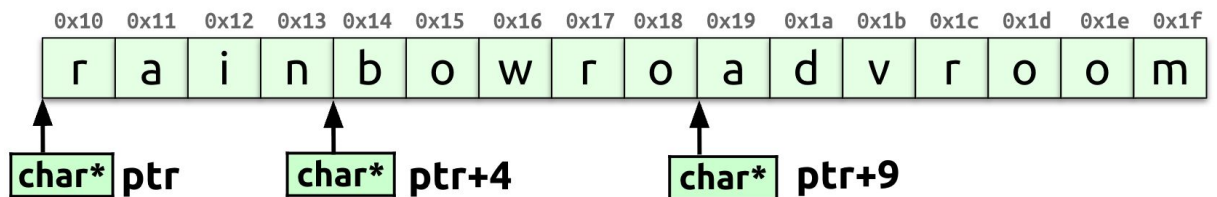


Pointer arithmetic works in terms of the **size of the data type being pointed to**

→ **+1** for an `int*` means advance one `int`, so **4 bytes**



→ **+1** for a `char*` means advance one `char`, so just **1 byte**



GDB Tip:

layout src



```
...ab/lab3 — ssh bbyan@myth.stanford.edu
(gdb) break main
Breakpoint 2 at 0x555555
(gdb) run
The program being debugged
Start it from the beginn
Starting program: /afs/.
[Thread debugging using
Using host libthread_db

Breakpoint 2, main (argc
52      int num = ar
(gdb) layout src
```



```
benjaminian — bbyan@myth65: ~/CS107_TA/lab3 — ssh bbyan@myth.stanford.edu — 63x20
buggy.c
47      // this location was already freed!
48      free(another);
49  }
50
51  int main(int argc, char *argv[]) {
B+>52      int num = argc > 1 ? atoi(argv[1]) : 0;
53      char *str = argc > 2 ? argv[2] : "Stanfor
54
55      if (num == 1) {
56          make_error_1(str);
57      } else if (num == 2) {
58          make_error_2();
59      } else if (num == 3) {

process 1100203 In: main      L52      PC: 0x555555555360
(gdb) info locals
num = 0
str = 0x7fffffff8b0 "\001"
(gdb)
```

As we're running GDB (i.e. stopped at a breakpoint), run **(gdb) layout src**

We have our **code** side-by-side with the debugger (with the **line we're about to run in GDB highlighted**) — neat right?