This is  Section.

Week 4: Memory

Gabe LeBlanc

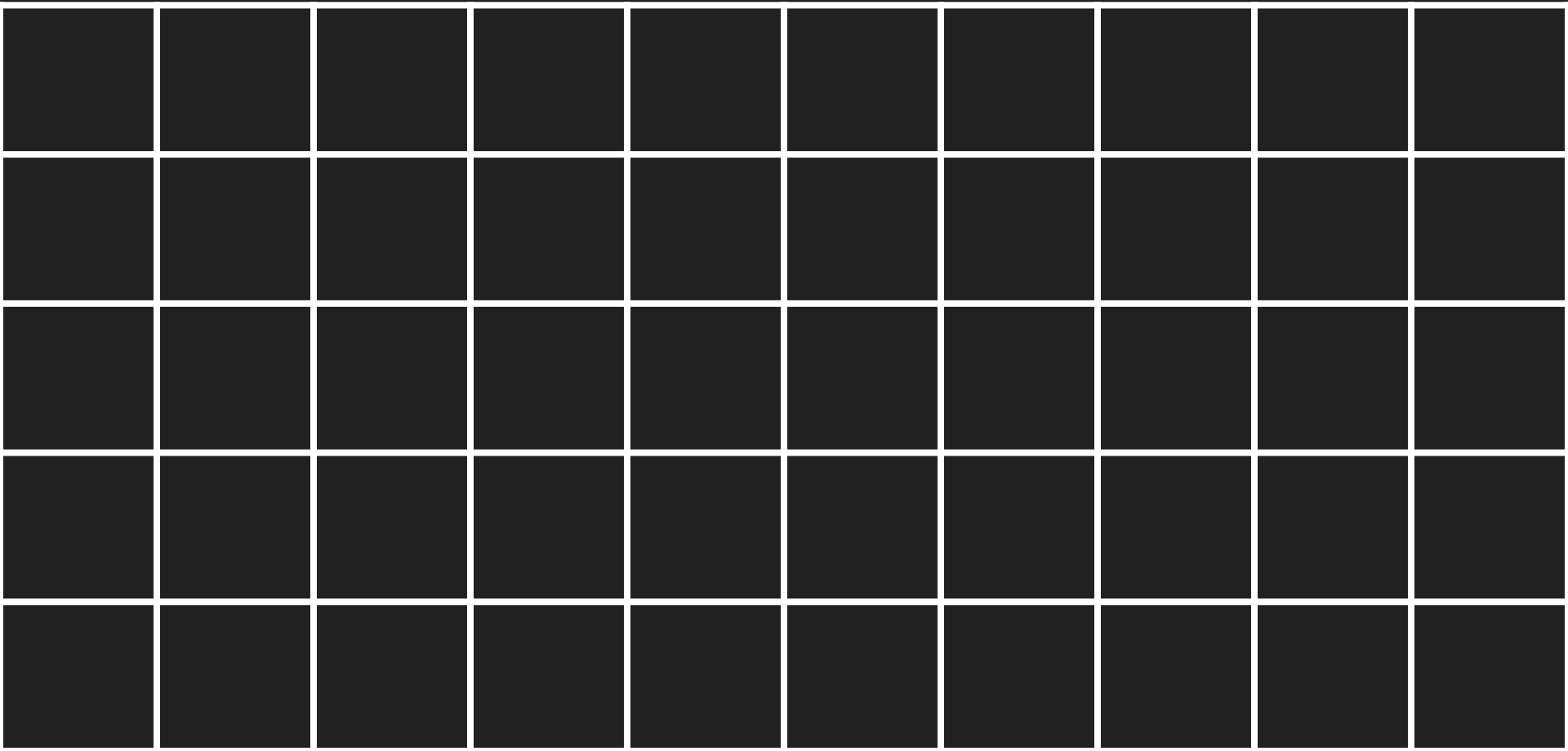
Attendance Form: tinyurl.com/gabesection4

`type *` is a pointer that stores the address of a **`type`**.

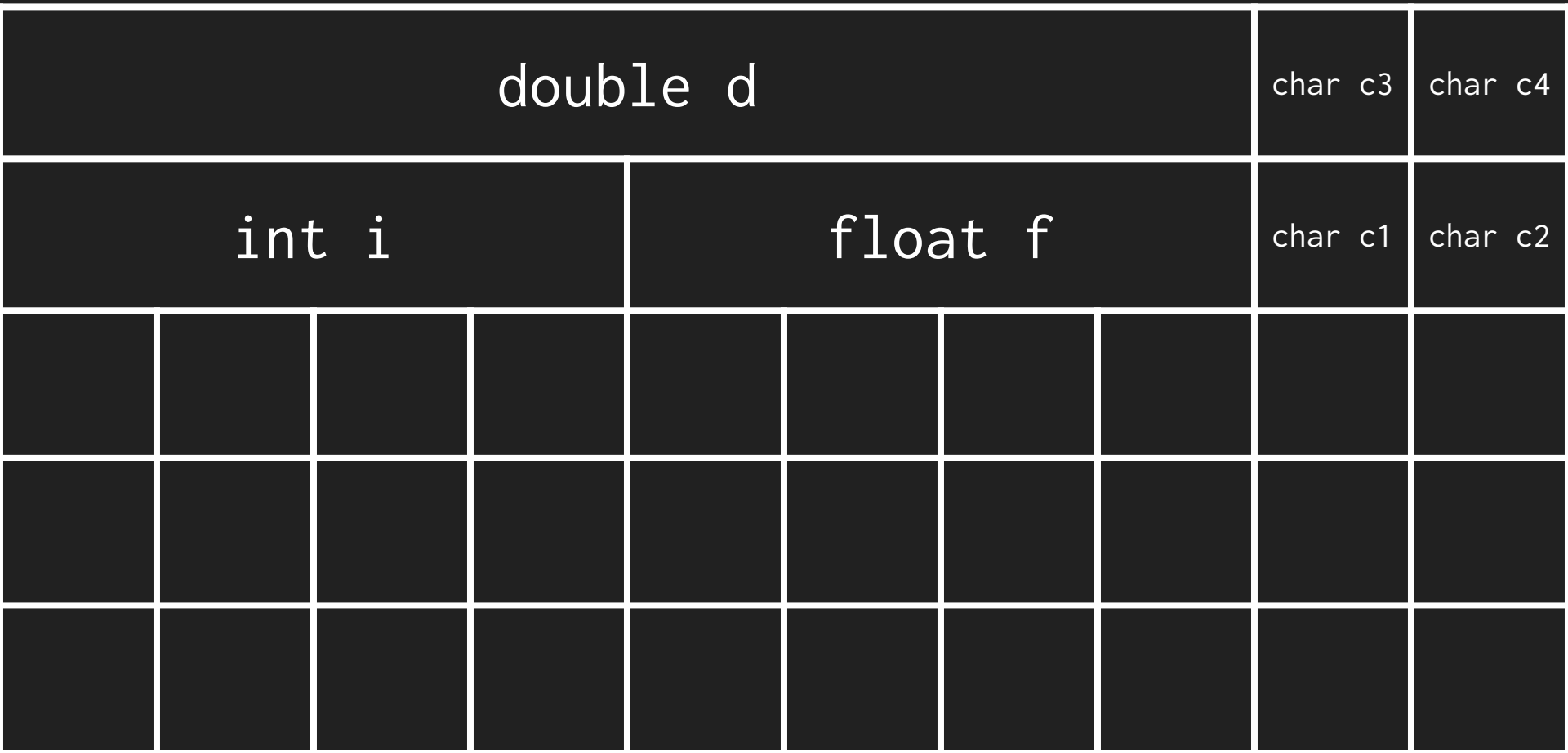
`*x` takes a pointer **`x`** and goes to the address stored at that pointer.

`&x` takes **`x`** and gets its address.

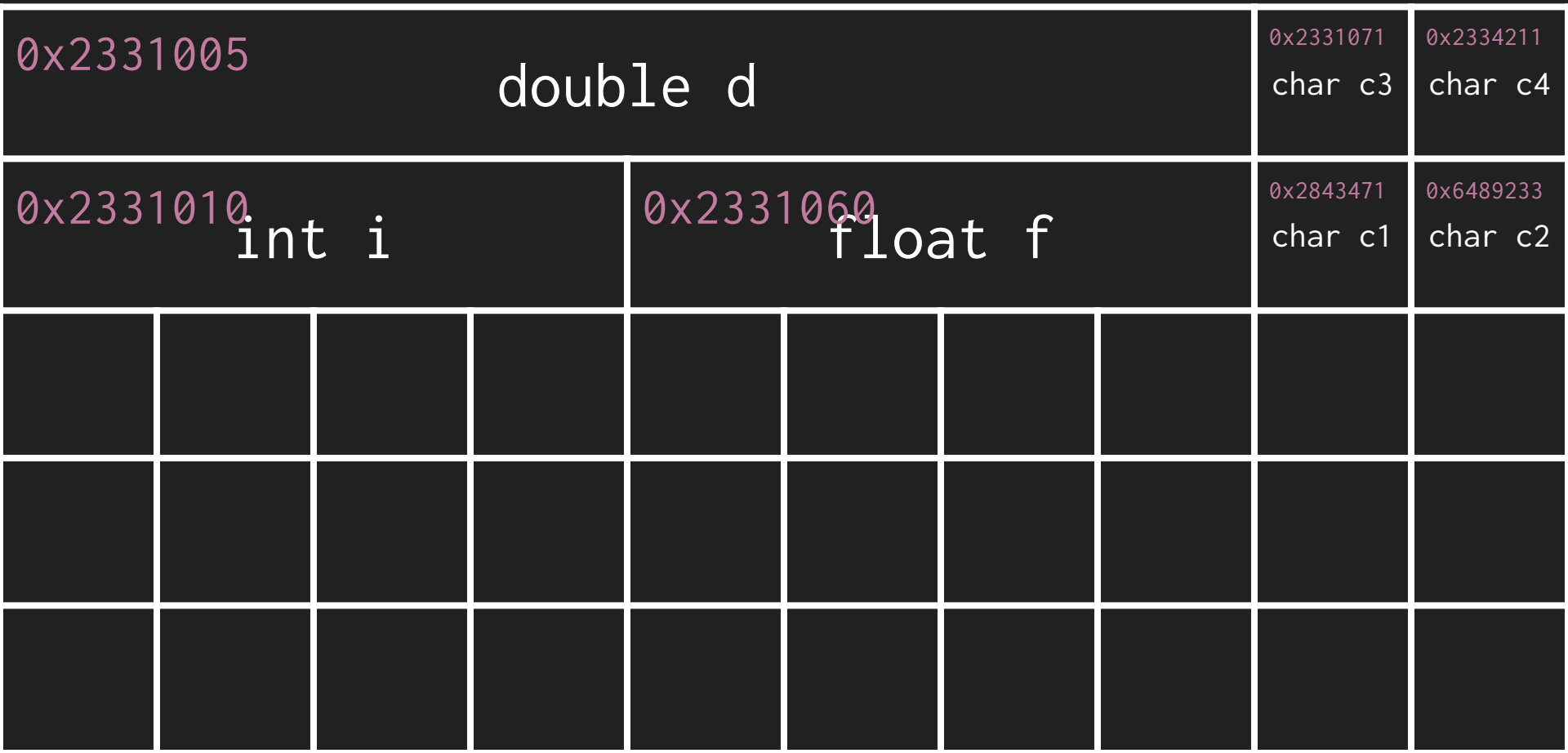
memory



memory

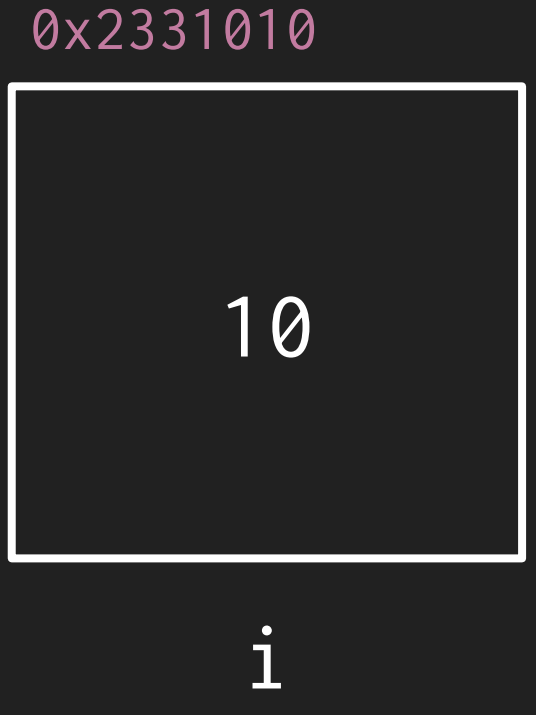


pointers

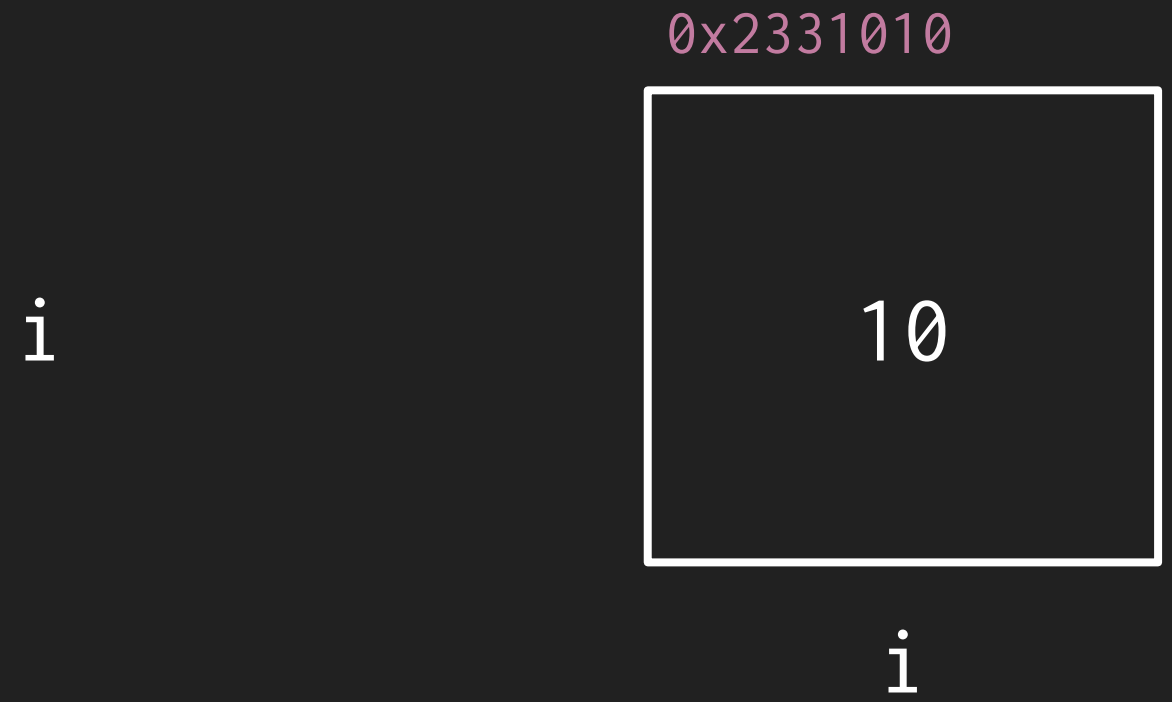

0x2331005
double d
0x2331071
char c3
0x2334211
char c4
0x2331010
int i
0x2331060
float f
0x2843471
char c1
0x6489233
char c2

pointers

```
int i = 10;
```

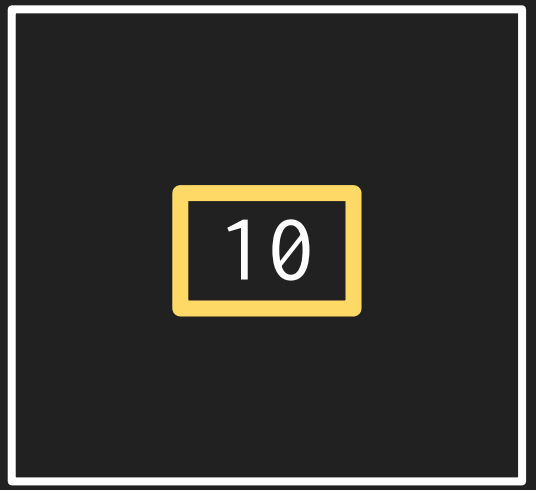


pointers

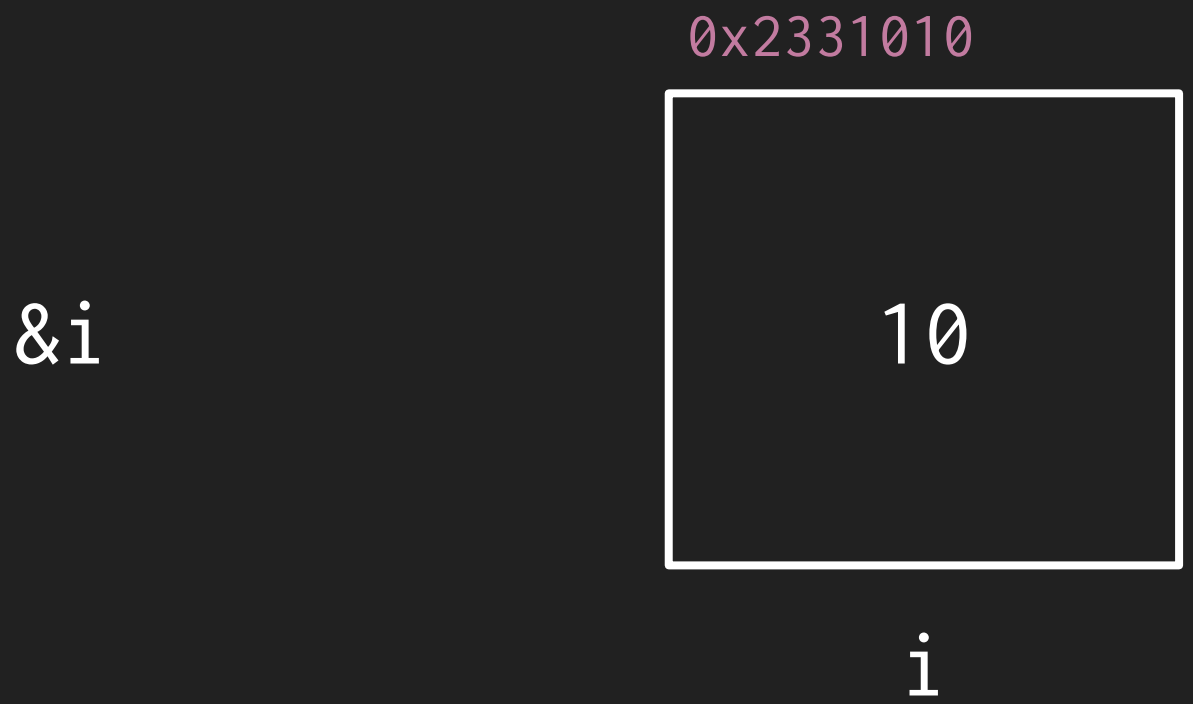


pointers

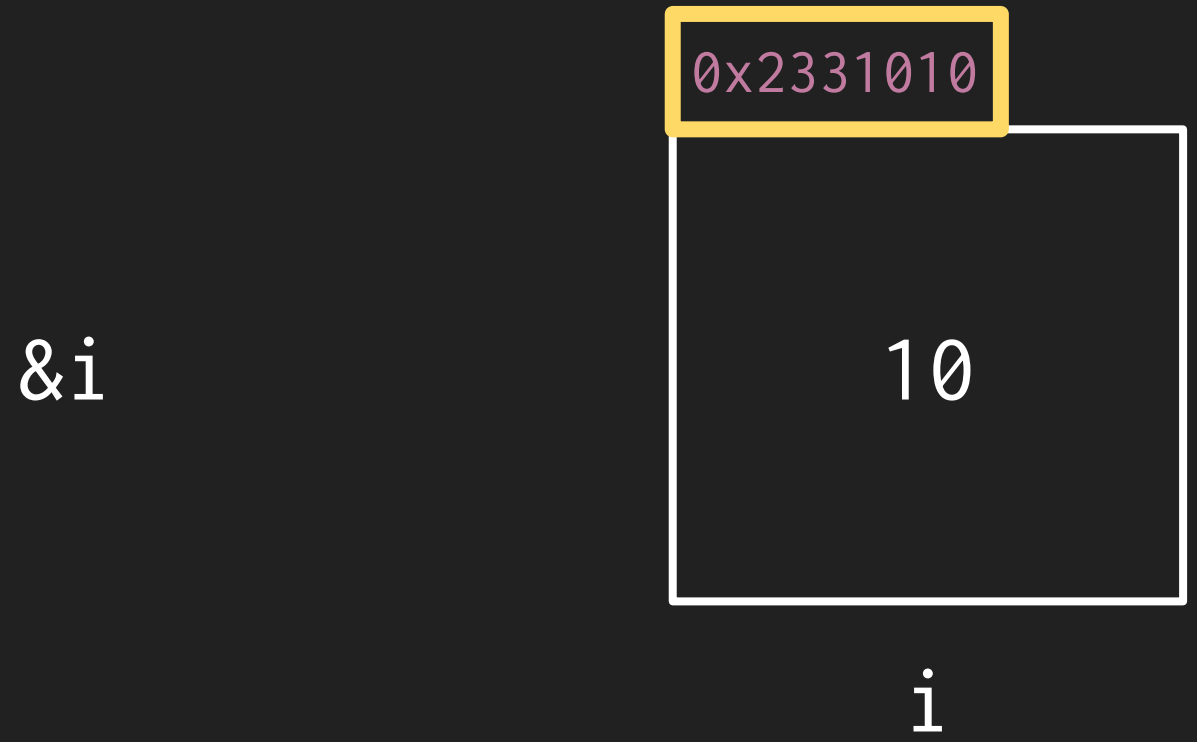
i



pointers

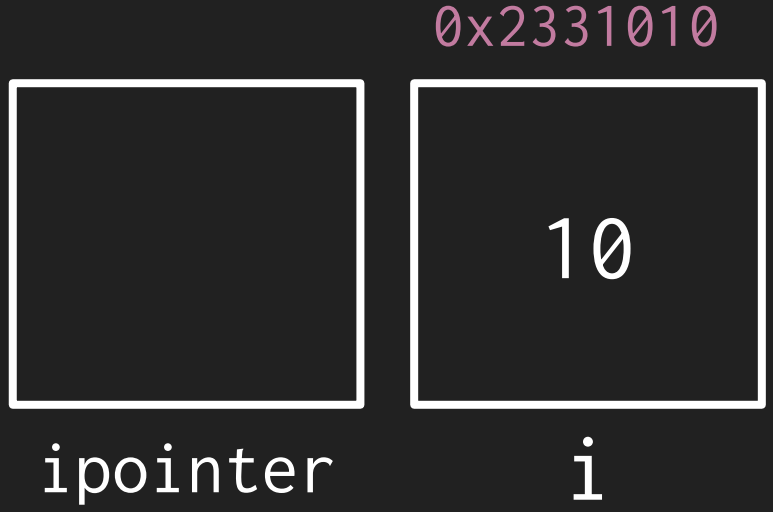


pointers



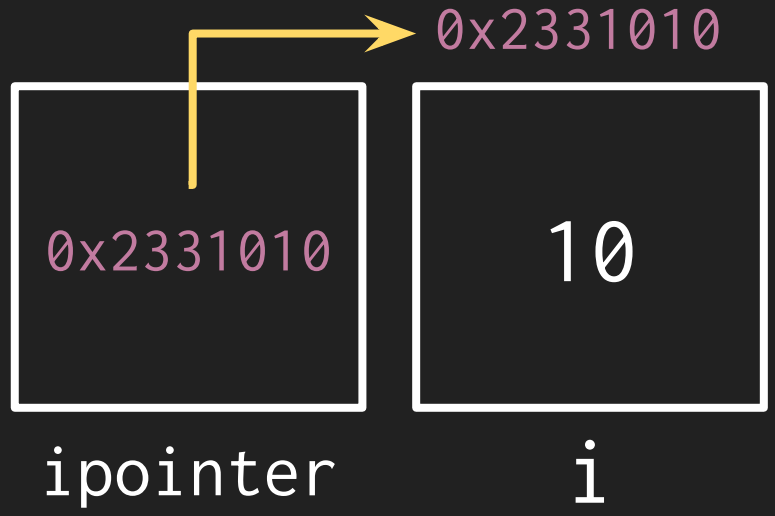
pointers

```
int *ipointer = &i;
```



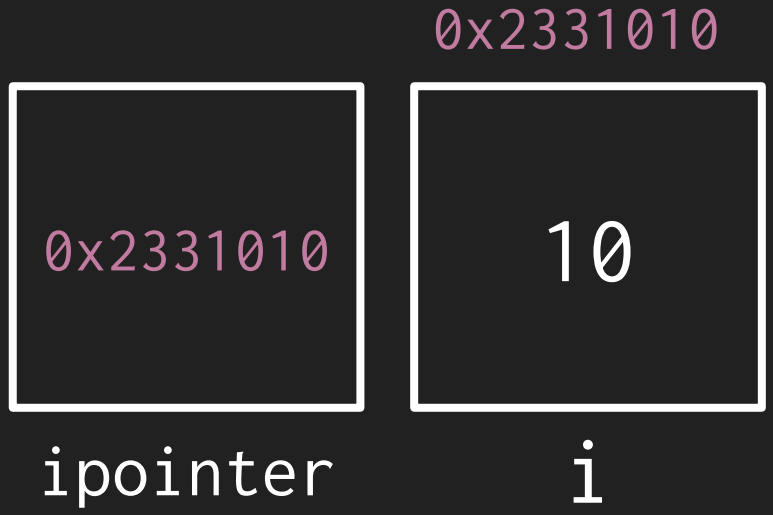
pointers

```
int *ipointer = &i;
```



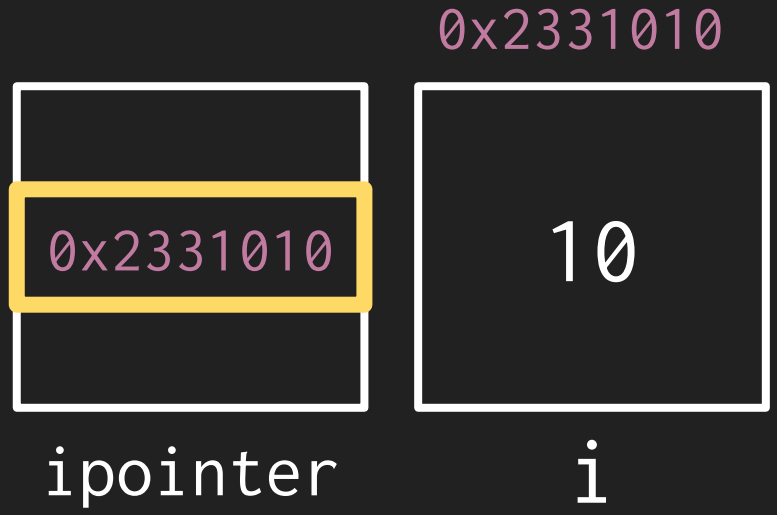
pointers

ipointer



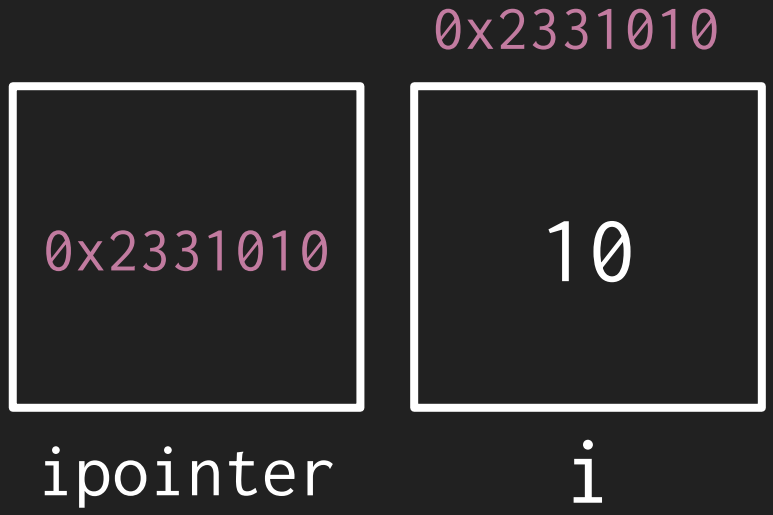
pointers

ipointer



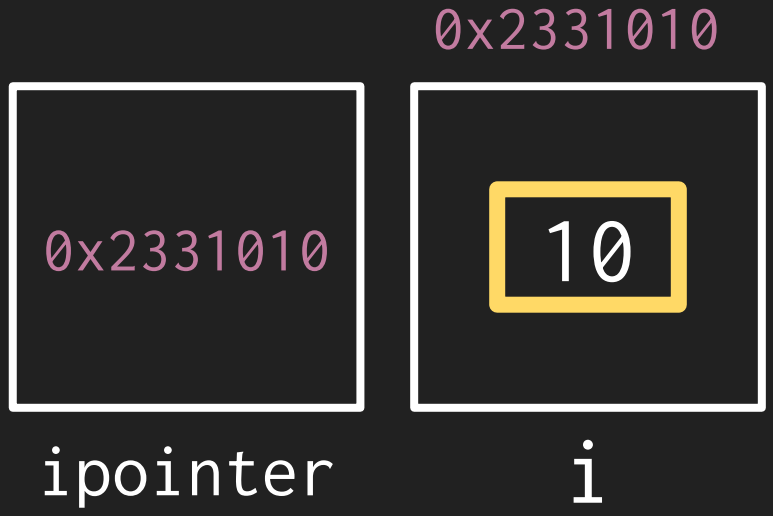
pointers

*ipointer



pointers

*ipointer



pointers

char*

~~strings~~

pointers

```
char *s = “hello”
```

pointers

```
char *s = “hello”
```

- As we start to work with pointers, just keep this image in mind:

k

```
int k;
```

- As we start to work with pointers, just keep this image in mind:


```
int k;
k = 5;
```

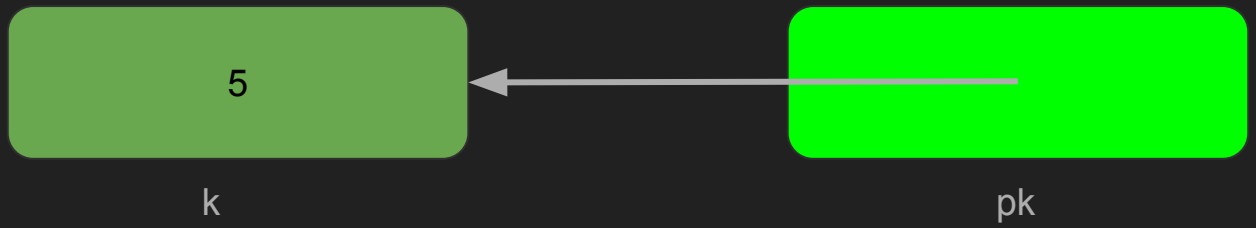
- As we start to work with pointers, just keep this image in mind:


```
int k;
k = 5;
int *pk;
```

- As we start to work with pointers, just keep this image in mind:


```
int k;
k = 5;
int *pk;
pk = &k;
```

- As we start to work with pointers, just keep this image in mind:




```
int k;
k = 5;
int *pk;
pk = &k;
```

- The simplest pointer available to us in C is the `NULL` pointer.
  - As you might expect, this pointer points to nothing (a fact which can actually come in handy!)

- When you create a pointer and you don't set its value immediately, you should **always** set the value of the pointer to `NULL`.

- You can check whether a pointer is `NULL` using the equality operator (==).

- What might happen if we try to dereference a pointer whose value is `NULL`?

- What might happen if we try to dereference a pointer whose value is `NULL`?

SEGMENTATION FAULT

- Surprisingly enough, this is actually good behavior! It defends against accidental dangerous manipulation of unknown pointers.
  - That's why we recommend you set your pointers to `NULL` immediately if you aren't setting them to a known, desired value.

File I/O

- The ability to read data from and write data to files is the primary means of storing **persistent data**, which exists outside of your program.

- The functions we use to manipulate files all are found in **stdio.h**. Every one of them accepts a `FILE *` as one of its parameters, except `fopen()` which is used to get a file pointer in the first place.

- Some of the most common file input/output (I/O) functions we'll use are the following:

- `fopen()` opens a file and returns a pointer to it. Always check its return value to make sure you don't get back `NULL`.

```
FILE *ptr = fopen(<filename>, <operation>);
```

```
FILE *ptr = fopen("test.txt", "r");
FILE *ptr2 = fopen("test2.txt", "w");
```

- `fread()` and `fwrite()` work for a generalized quantity (`qty`) of blocks of an arbitrary (`size`), holding those blocks in (or writing them from) a temporary buffer, usually an array, for local use within the program.

```
fread(<buffer>, <size>, <qty>, <file pointer>);
```

```
fwrite(<buffer>, <size>, <qty>, <file pointer>);
```

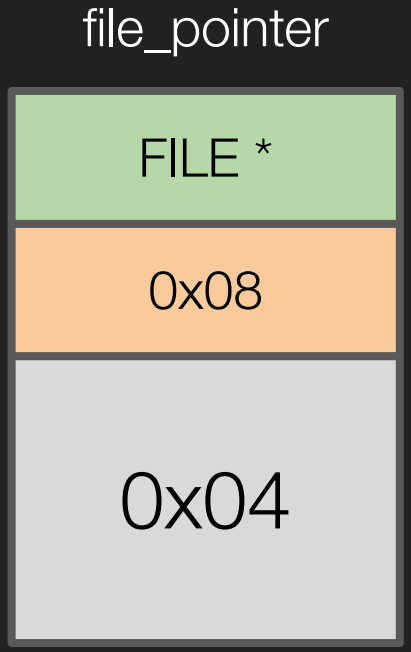
```
int arr[10];
fread(arr, sizeof(int), 10, ptr);
fwrite(arr, sizeof(int), 10, ptr2);
```

- `fclose()` closes a previously opened file pointer.

```
fclose(<file pointer>);
```

```
fclose(ptr);
```

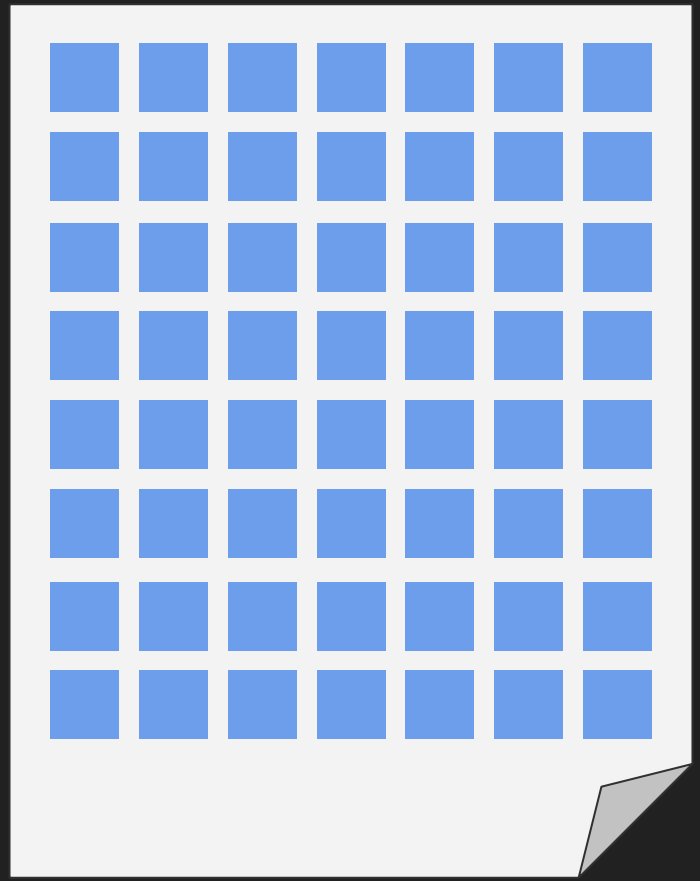
```
fclose(ptr2);
```




```
FILE *file_pointer =
fopen("test.txt", "r");
```

file_pointer

file_pointer

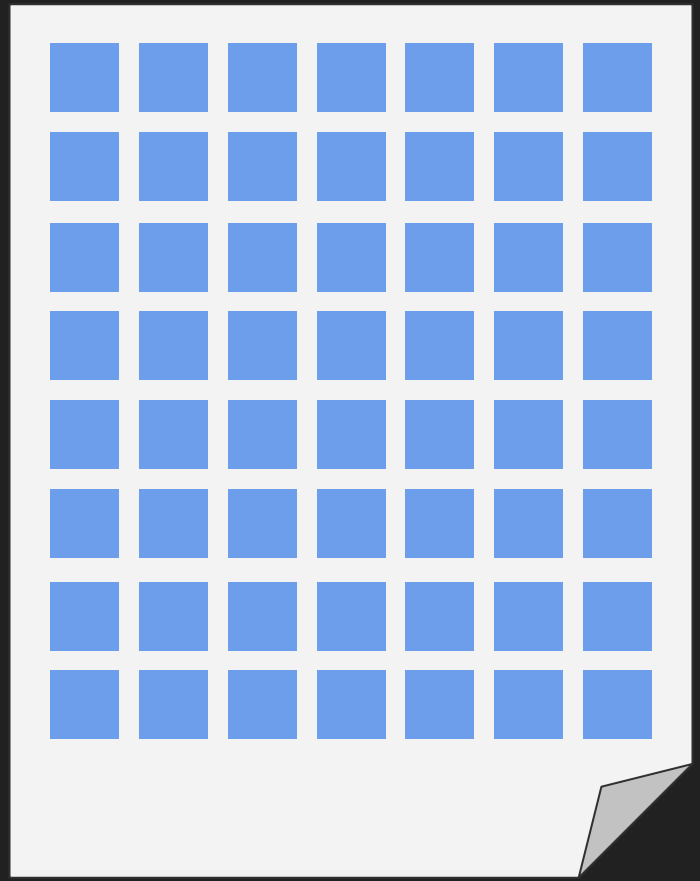


```
fread(buffer, 1, 4, file_pointer);
```

```
fread(buffer, 1, 4, file_pointer);
```

Location to read from

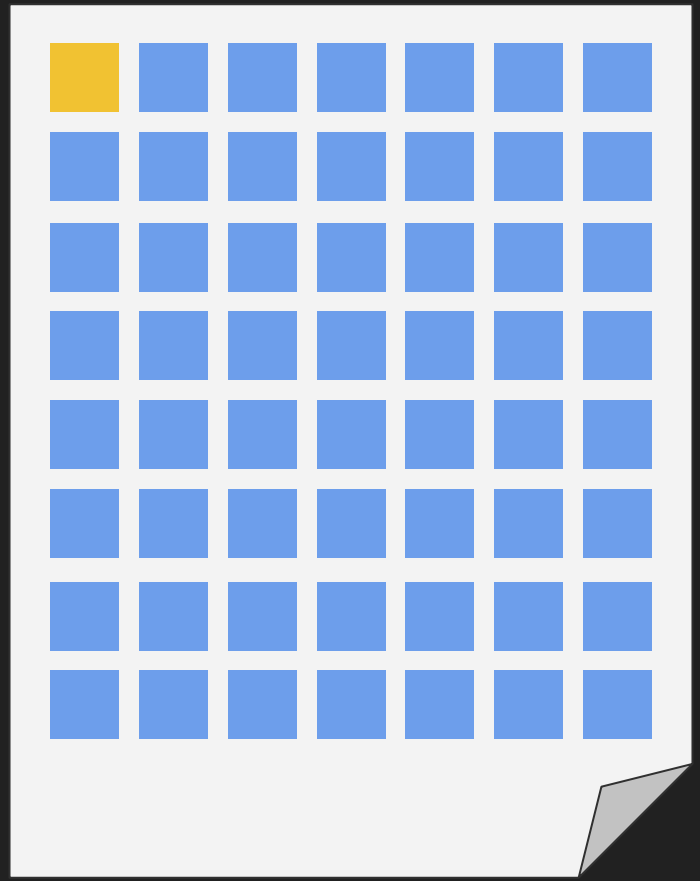
file_pointer



```
fread(buffer, 1, 4, file_pointer);
```

Size of blocks to read (in bytes)

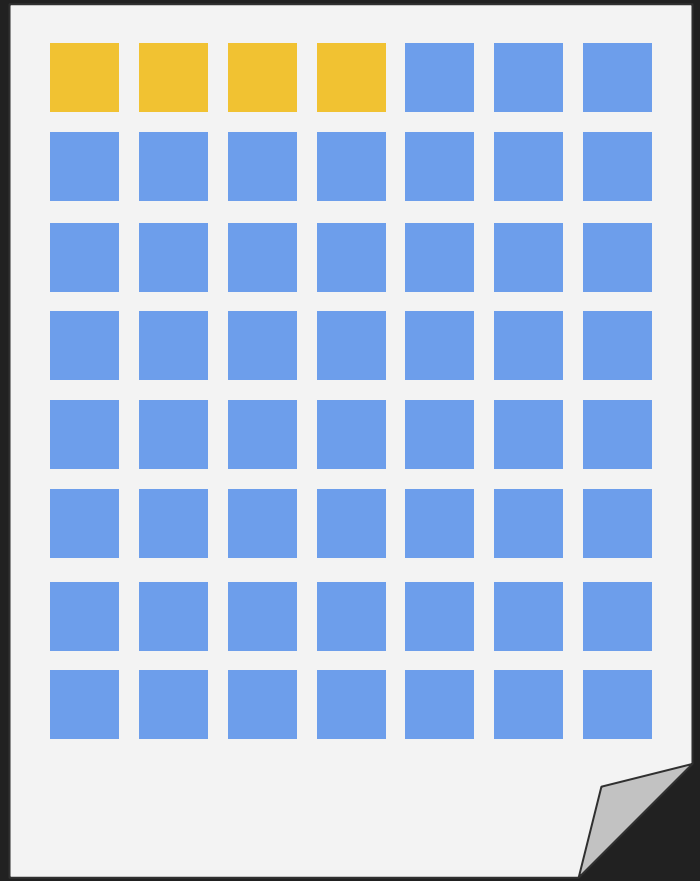
file_pointer



```
fread(buffer, 1, 4, file_pointer);
```

How many blocks to read

file_pointer

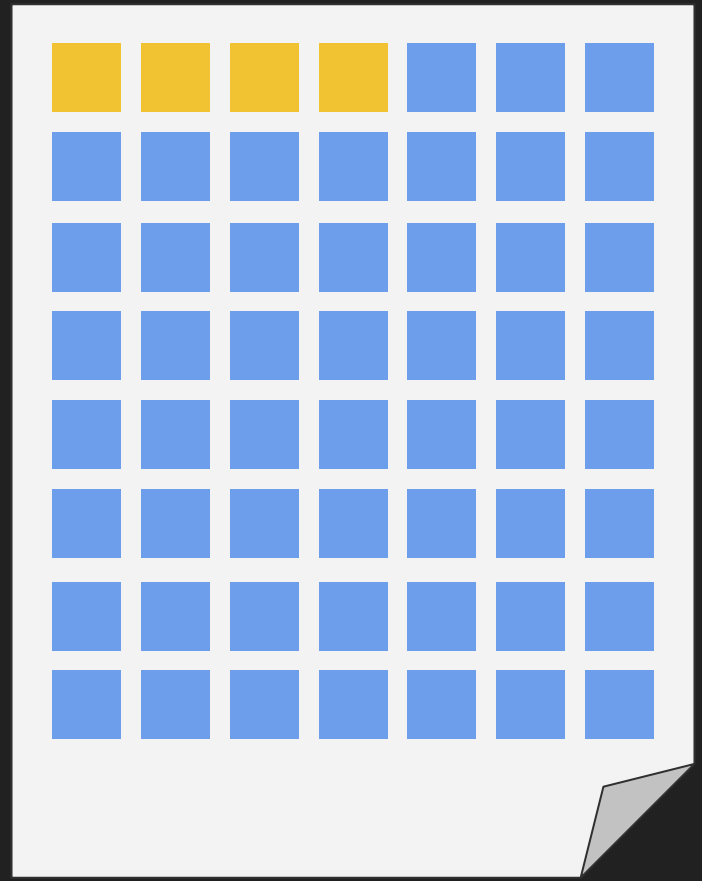


```
fread(buffer, 1, 4, file_pointer);
```

Location to store blocks

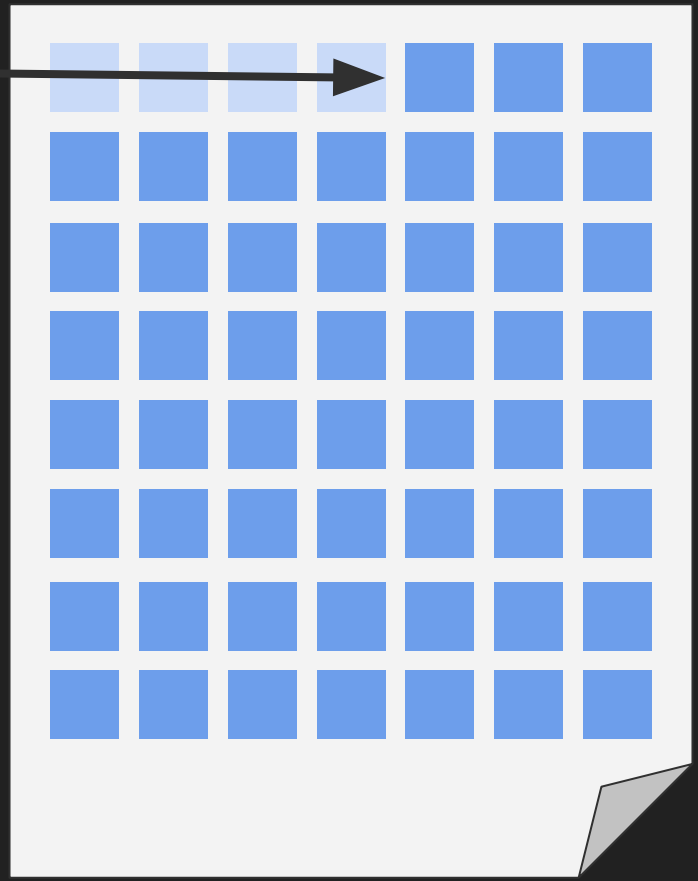
file_pointer

buffer



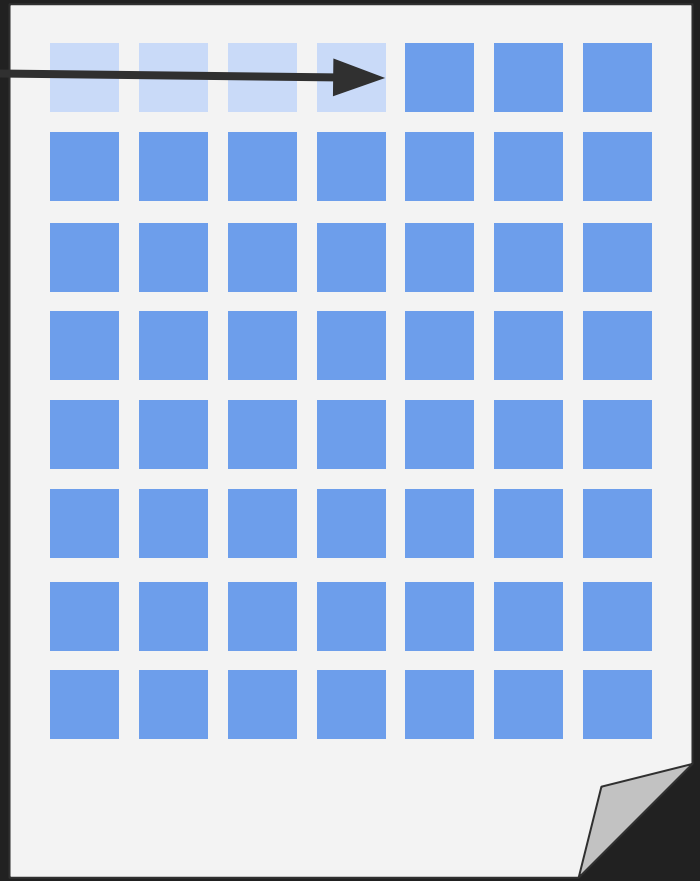
file_pointer

buffer



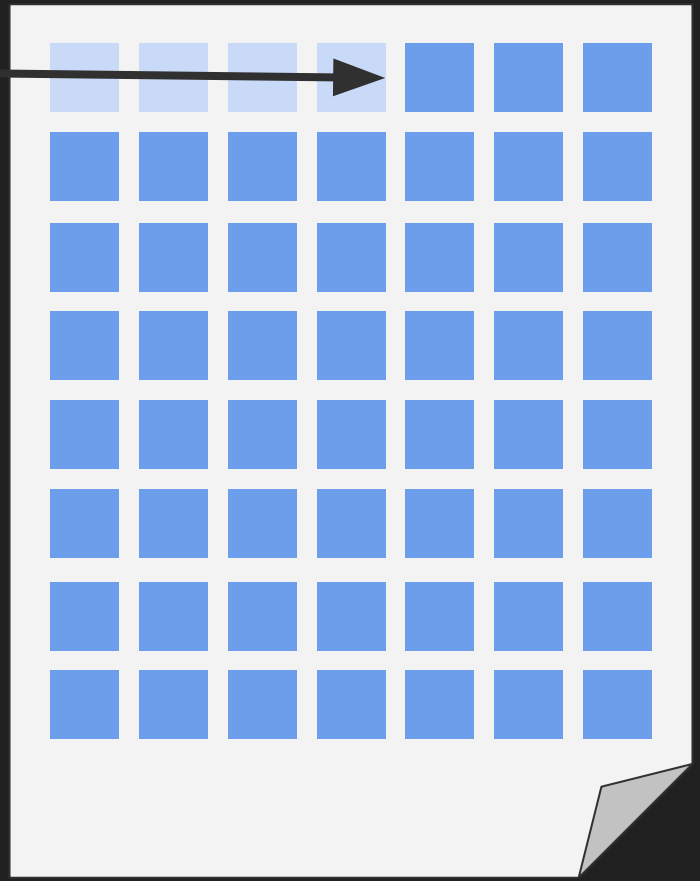
file_pointer

buffer[0]



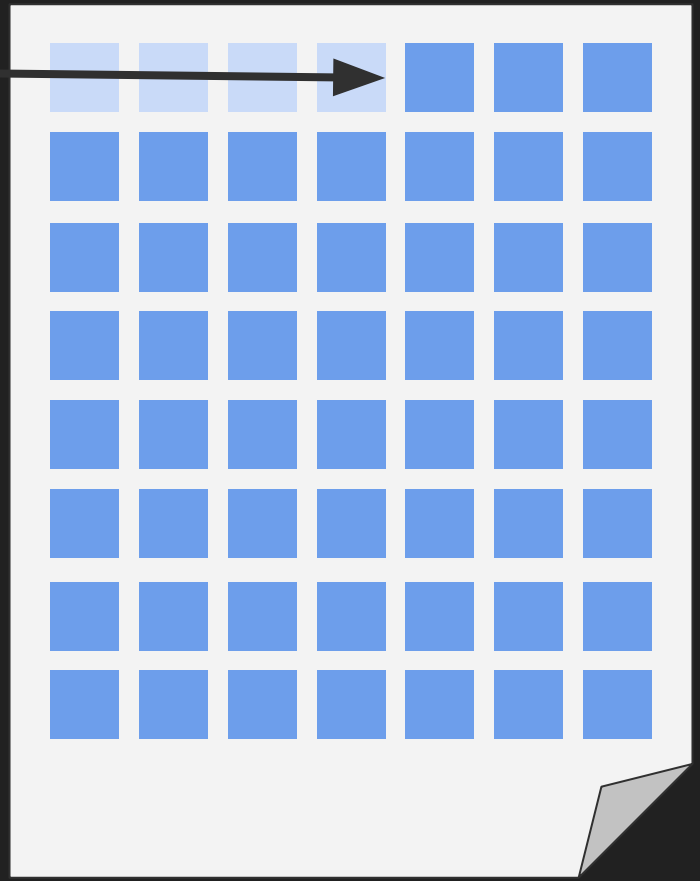
file_pointer

buffer[1]



file_pointer

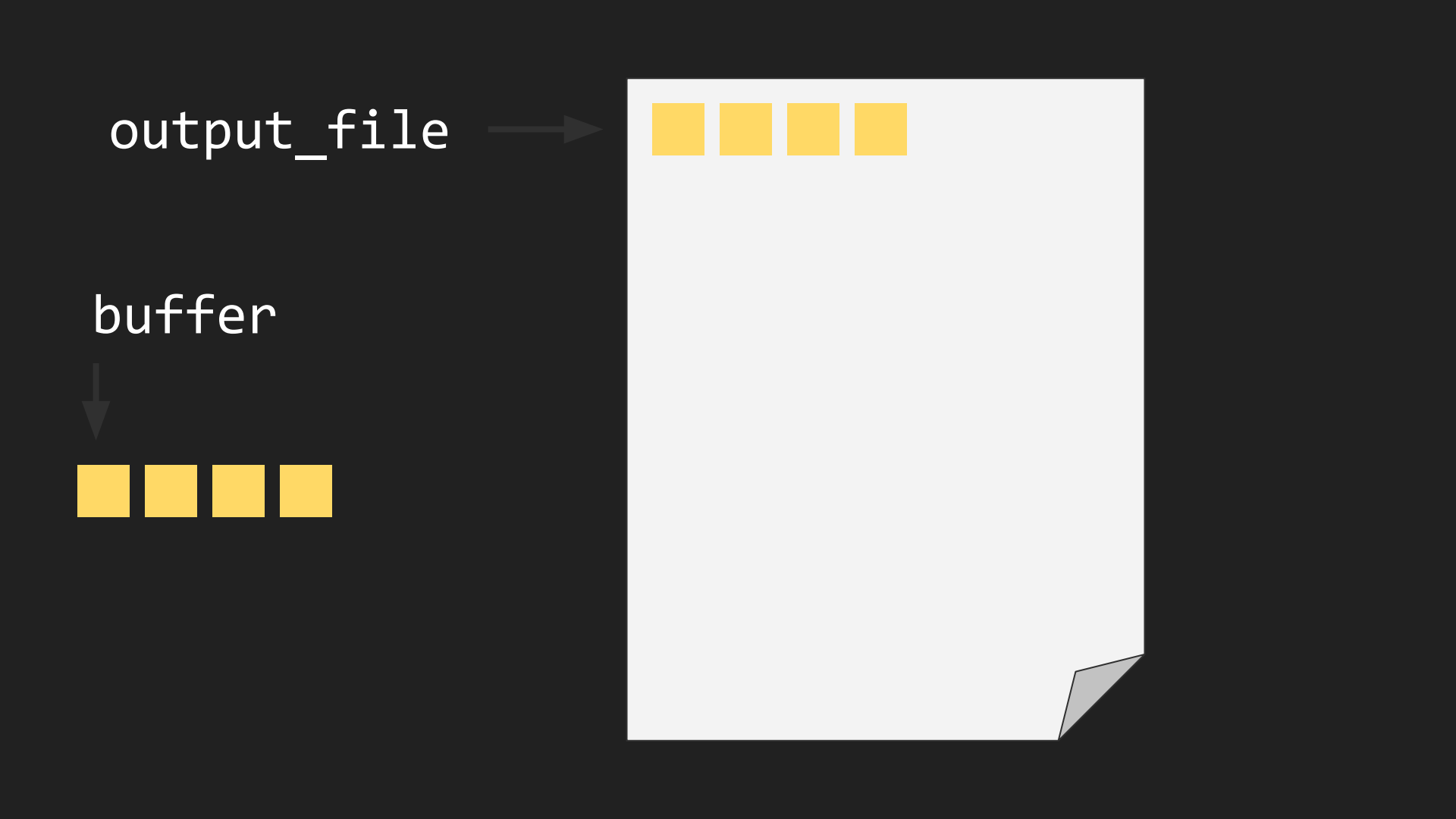
buffer[2]



```
fwrite(buffer, 1, 4, output_file);
```

output_file

buffer


output_file
buffer

1. open your input file (read)
2. open your output file (write)
3. read from your input file
4. process data
5. write to your output file
6. close all files (free memory)

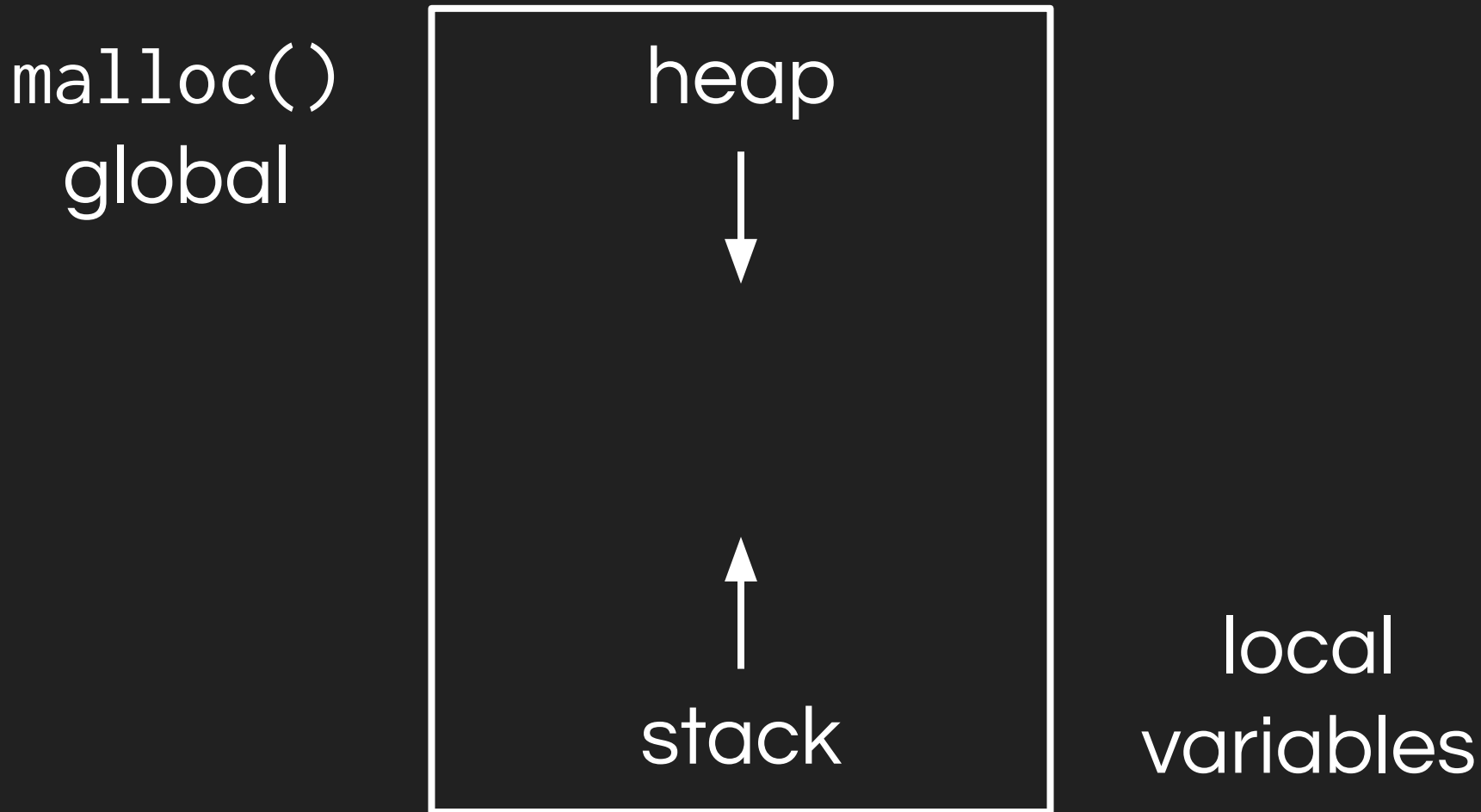
Stack

The stack is a contiguous block of memory set aside when a program starts running

- Stored in the stack:
  - metadata
  - any variables held in "read-only" memory
  - all local variables
- Everything in stack memory is “forgotten” when the program is complete!

Heap

- Memory allocated during runtime is called dynamically allocated memory and it's held on the heap.
- The heap is essentially a region of unused memory that can be allocated with a call to malloc():
  - int* ptr = malloc(sizeof(int));
- This space can be used to store data, it can be passed between functions, and (unlike stack variables) it won't be lost when a function returns.
- Don’t forget to free()!


malloc()
global
heap
stack
local
variables

- We get this dynamically-allocated memory via a call to the function `malloc()`, passing as its parameter the number of *bytes* we want. `malloc()` will return to you a **pointer** to that newly-allocated memory.
  - If malloc() can't give you memory (because, say, the system ran out), you get a NULL pointer.

```
// Statically obtain an integer
int x;

// Dynamically obtain an integer
int *px = malloc(sizeof(int));
```

- There’s a catch: Dynamically allocated memory is not automatically returned to the system for later use when no longer needed.

- Failing to return memory back to the system when you no longer need it results in a **memory leak**, which compromises your system’s performance.

- All memory that is dynamically allocated must be released back by `free()`-ing its pointer.

- Every block of memory that you `malloc()`, you must later `free()`.
- **Only** memory that you obtain with `malloc()` should you later `free()`.
- Do not `free()` a block of memory more than once.

Hexadecimal

- We typically use the decimal system (base 10) to represent numeric data.

- Computers use the binary system (base 2) to represent numeric (and indeed all) data.

- As computer scientists, it’s useful to be able to express data the same way the computer does, but trying to parse a big chain of 0s and 1s can be annoying.

- The **hexadecimal system** (base 16) is a much more concise way to express data on a computer system.

```
0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
```

- Hexadecimal makes it so that a group of four binary digits (bits) can be expressed with a single character, since there are 16 possible combinations of 4 bits (0 or 1).

Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Decimal	Binary	Hex
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f

Decimal	Binary	Hex
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
7	0111	0x7

Decimal	Binary	Hex
8	1000	0x8
9	1001	0x9
10	1010	0xa
11	1011	0xb
12	1100	0xc
13	1101	0xd
14	1110	0xe
15	1111	0xf

- To convert a binary number to hexadecimal, group four bits together from **right to left**, padding the leftmost group with extra 0s at the front if necessary.

- To convert a binary number to hexadecimal, group four bits together from **right to left**, padding the leftmost group with extra 0s at the front if necessary.

```
01000110101000101011100100111101
```

- To convert a binary number to hexadecimal, group four bits together from **right to left**, padding the leftmost group with extra 0s at the front if necessary.

```
01000110101000101011100100111101

0100 0110 1010 0010 1011 1001 0011 1101
```

- To convert a binary number to hexadecimal, group four bits together from **right to left**, padding the leftmost group with extra 0s at the front if necessary.

```
01000110101000101011100100111101

0100 0110 1010 0010 1011 1001 0011 1101

 4    6    A    2    B    9    3    D
```

- To convert a binary number to hexadecimal, group four bits together from **right to left**, padding the leftmost group with extra 0s at the front if necessary.

```
01000110101000101011100100111101

0100 0110 1010 0010 1011 1001 0011 1101

 4    6    A    2    B    9    3    D

0x46A2B93D
```

github.com/gblanc25/cs50

EXTRAPRACTICE → WEEK4 → pointers.c

Volume