Supplementary: The Computer Memory

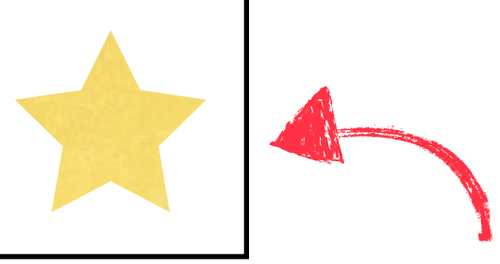
created by Wilhansen Li

The Java Abstraction

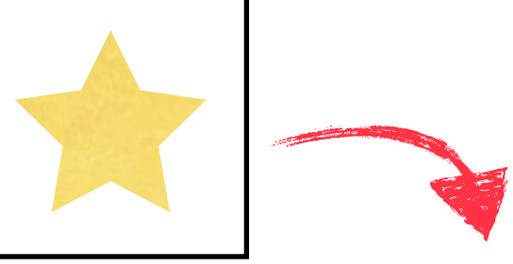
- Java's abstractions are helpful but they miss a lot of things about memory:
- In a `StackOverflowException`, exactly what stack overflowed?
- Where are objects, when you call `new`, coming from?
- When using `RandomAccessFile`, what happens when you `readLong` what should've been a `short`?
- What is memory?

This is Memory

The memory is a blank slate...



...that the computer can directly write...



...and read from.

(erasing is just writing a “null”)

Memory is divided into cells whose size is up to the system architecture. (usually one byte)

Each cell has an address.

(the addressing system is system dependent too)

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79

Most systems have a *flat* addressing scheme:

1) a number dictates the actual memory cell
2) adjacent cells have adjacent addresses (i.e. memory is contiguous)

0:0	0:1	0:2	0:3	0:4	0:5	0:6	0:7	0:8	0:9
1:0	1:1	1:2	1:3	1:4	1:5	1:6	1:7	1:8	1:9

Other systems have *segmented* addressing:

1) Memory is divided into segments (the size is also system-dependent, usually 64 KiB)

2) An address is composed of the segment address and the offset within the segment, written in the format `segment:offset`

Many systems offer both schemes

6:0	6:1	6:2	6:3	6:4	6:5	6:6	6:7	6:8	6:9
7:0	7:1	7:2	7:3	7:4	7:5	7:6	7:7	7:8	7:9

Why Segmentation?

- Many architectures load or cache memory by segments (more efficient than doing it byte-by-byte).
- Modern architectures assign memory access privileges by segments to minimize overhead.
- We'll discuss more of this in detail when we get to memory management.

Java “Equivalence”

- One way to look at it from a Java perspective is that memory is a huge array.

```
byte [] mem = x86.getMemory()
```

- The array index is the address and the type of the array is the memory unit.

C/C++ Equivalence

- In C/C++, an address is represented as a pointer.
- Incrementing a pointer in C/C++ is equivalent to going to the "next" address (the "next" depends on the data type of the pointer)
- Dereferencing a pointer is reading/writing the memory at the location referred by the pointer.

```
int *p = 0x0F;
  p = p + 1;
```

Checkpoint Questions

- In a x86 32-bit system, each memory cell is a byte and the maximum size of a CPU register is 32-bits. What's the maximum addressable memory?
- In most hard disks, each "cell" (called a sector) is 512 bytes. Up to how many bytes can be accessed if an address is 12-bits?

Memory Address Translation

- All modern architectures (like the x86 and ARMv4+) is capable of memory address translation.
- e.g. segments 0 may get translated to segment 4, segment 2 may get translated to segment 9 etc.

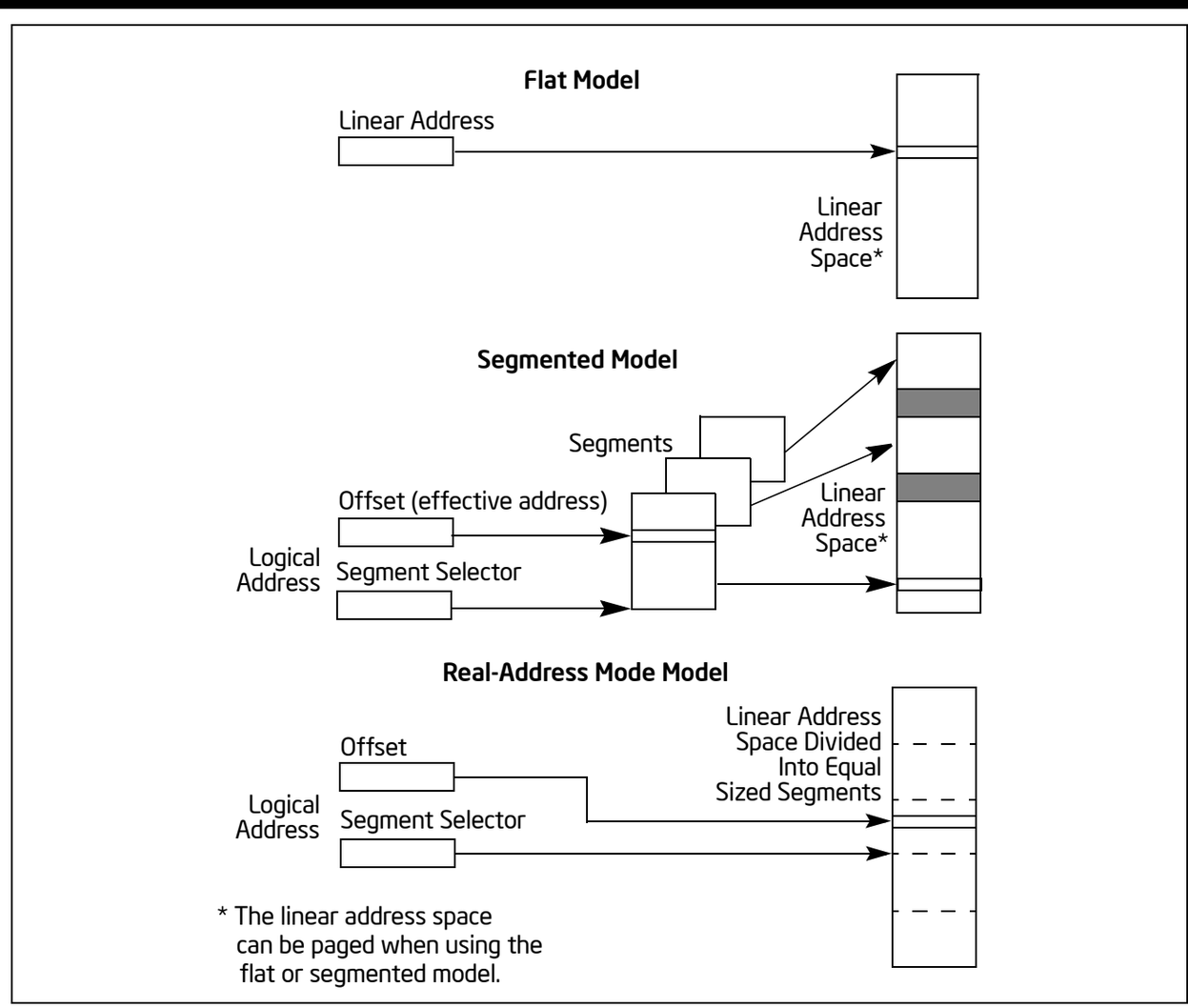


Figure 3-3. Three Memory Management Models

Source: Intel Developer Manual (vol.1 3-9)

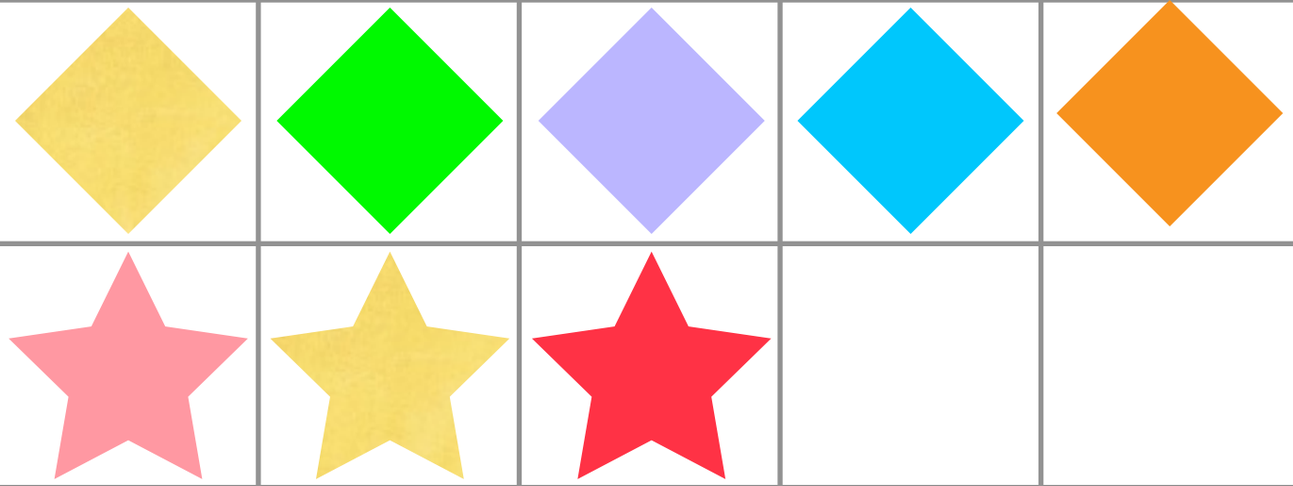
Memory Organization and Data Types

Memory is just a blank slate. It does not demarcate which piece of data goes where.

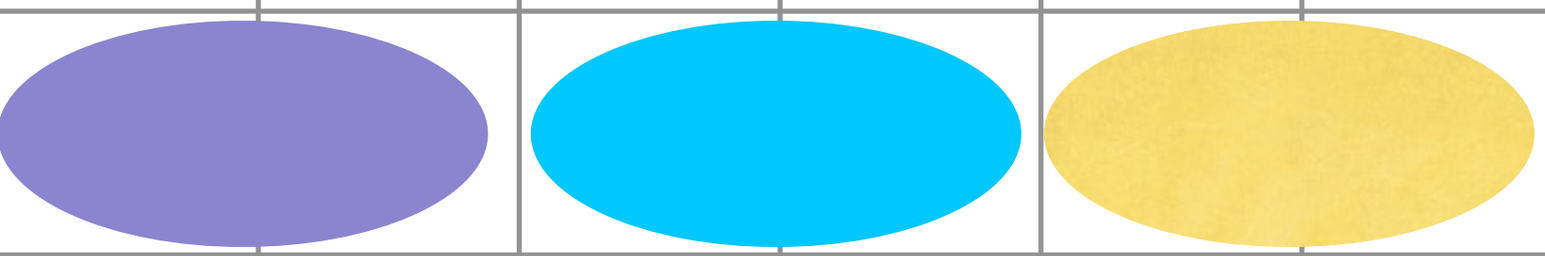
You can write all over the place...

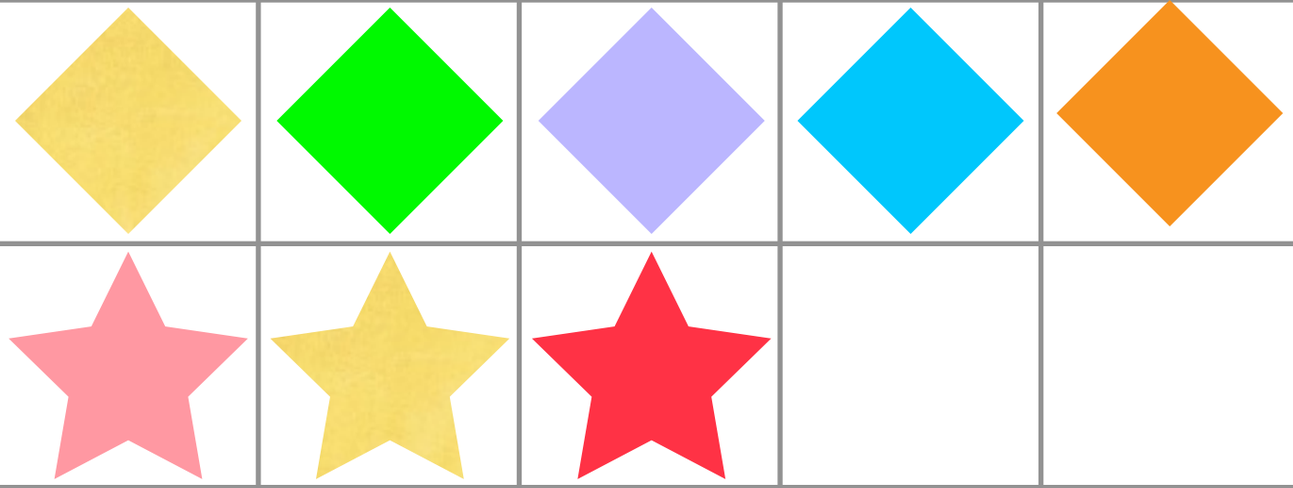
...but how will you know where to retrieve them?

(note that “use a hash map” or “use an array” also consumes memory)



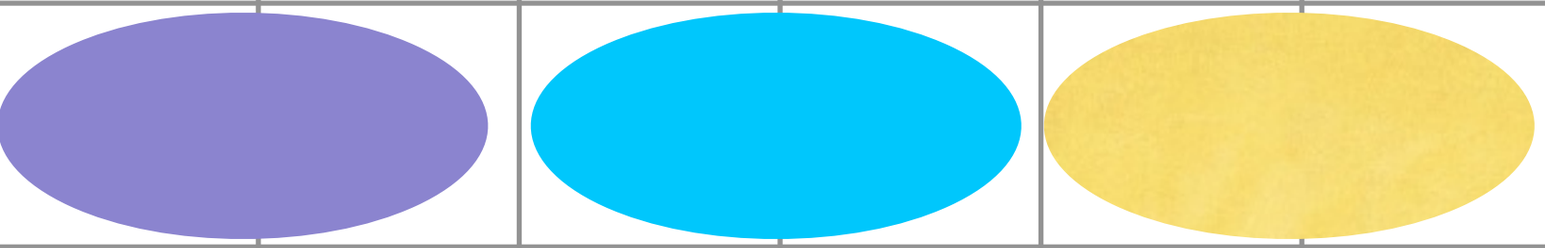
It’s up to the programmer (or programming language) to establish protocols on memory access.

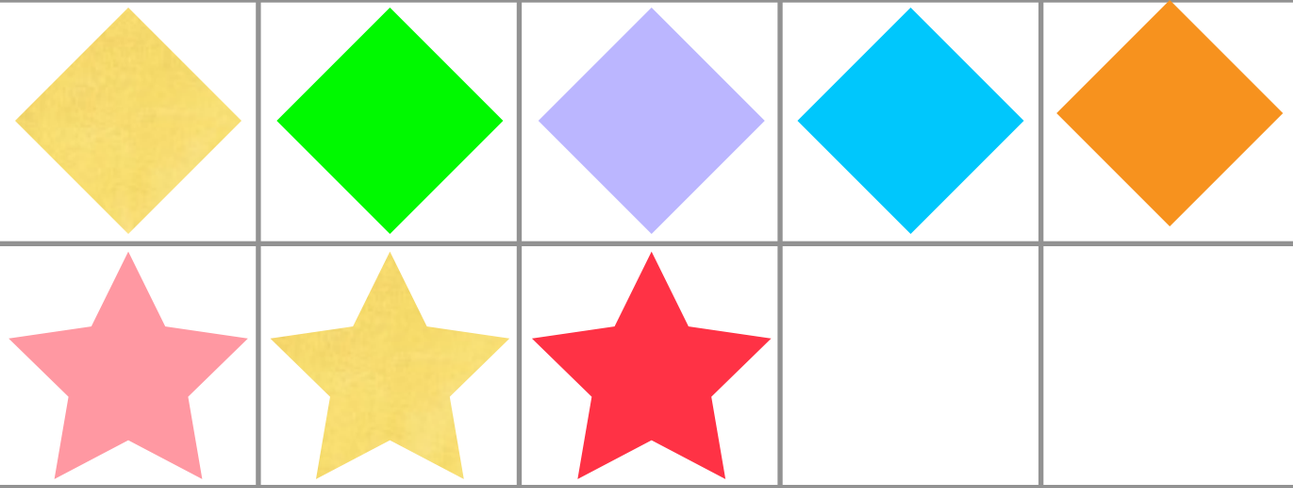




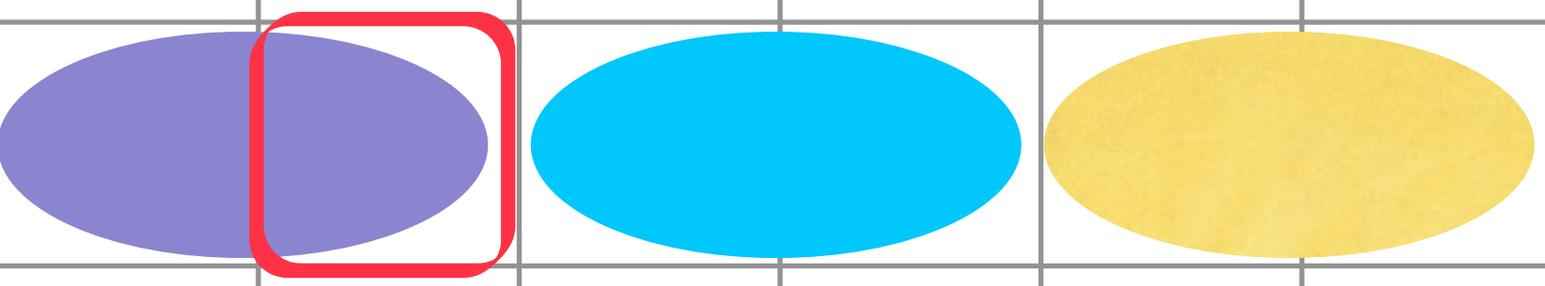
Another issue is knowing how large (i.e. how many bytes) a data is.

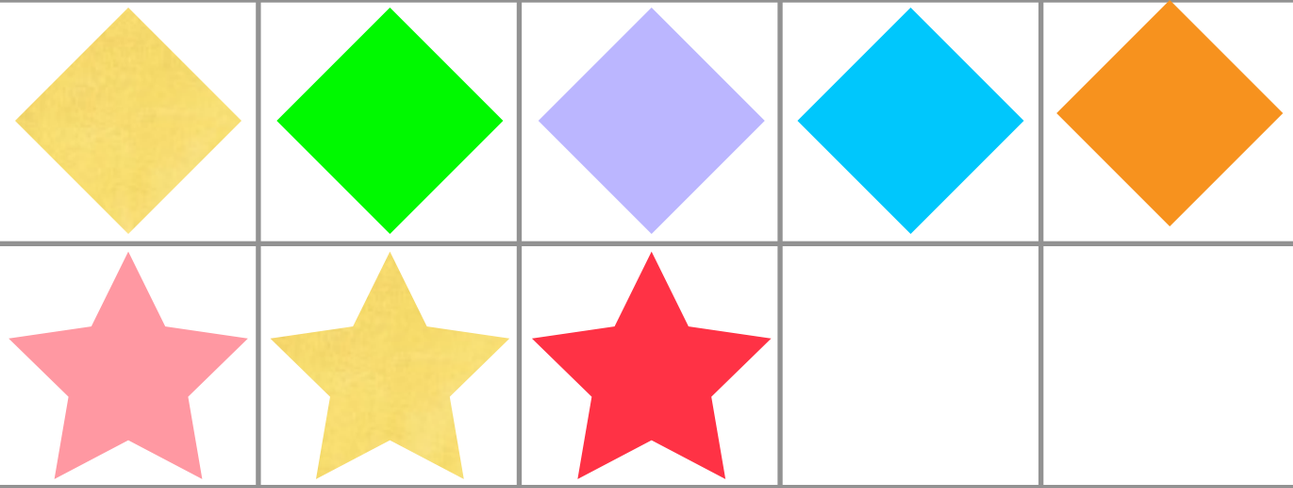
(Note that storing data size also consumes memory.)



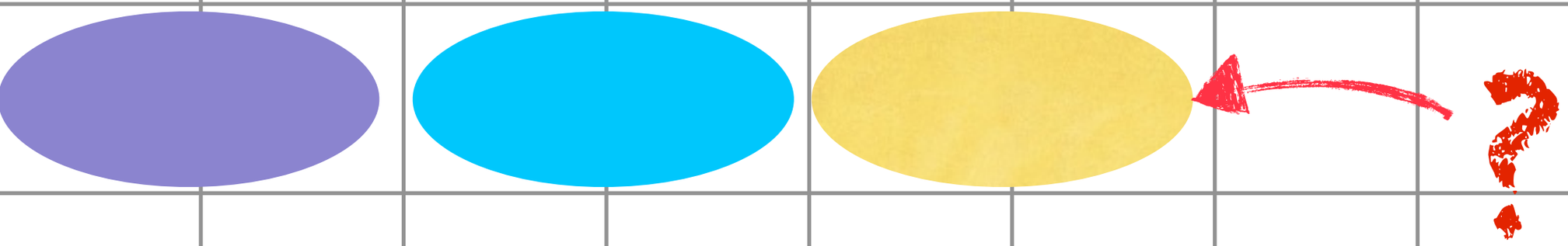


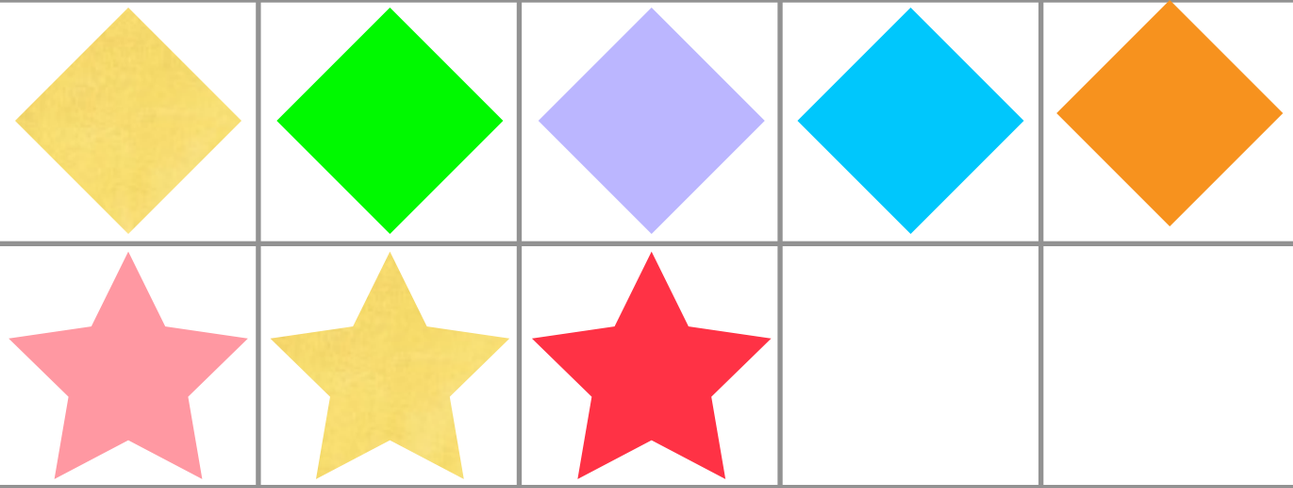
It’s entirely possible that you’ll be reading half the data that you wrote, (again, memory does not demarcate)



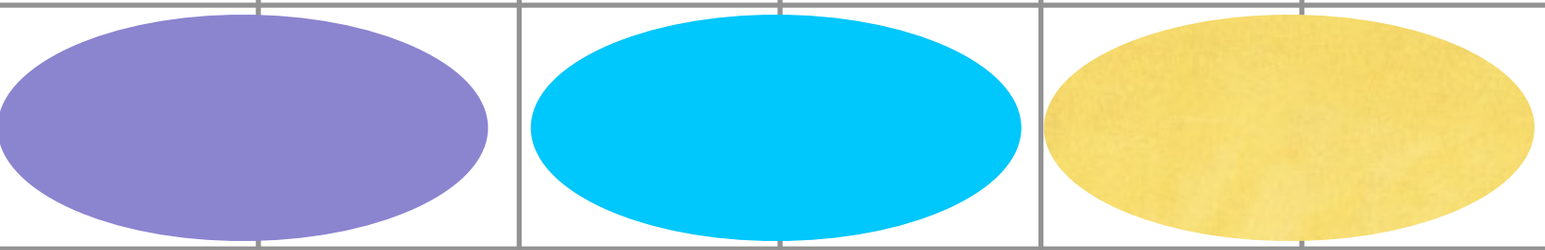


and there's also the issue on how to interpret data (a string? a number? a floating-point number? executable code?).





This is why programming languages have the concept of *data types*.



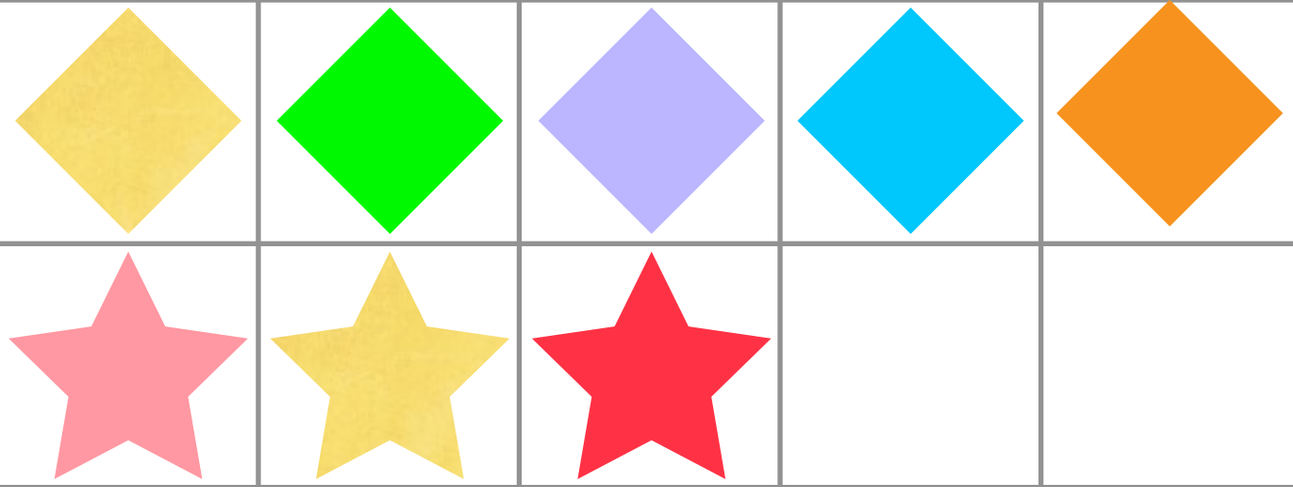
Checkpoint

- "Data about data" is called metadata.
  - What are the metadata associated for each piece of data?
  - Where is the metadata located/ stored in memory?

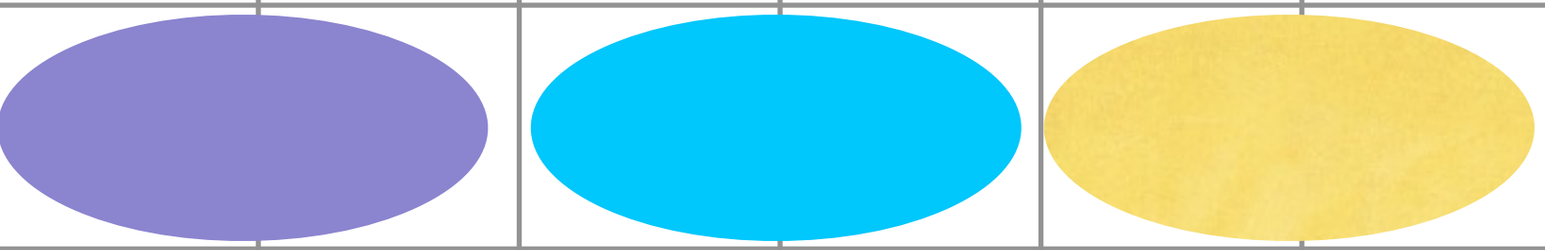
C/C++ Type System

- C/C++ has a weakly-typed system.
- A data's type can be reinterpreted (read: subverted) to another data type through pointer casts (`reinterpret_cast` in C++)
  - Examples: Reinterpret a pointer (i.e. address) as a number, reinterpret a 32-bit integer as a 4-character string.

```
int a = 0xA3DD3F7F;
int b = reinterpret_cast<int>(&a);
char *c = reinterpret_cast<char*>(&a);
```



The issue of "where" data is located still remains. For some cases, we have...



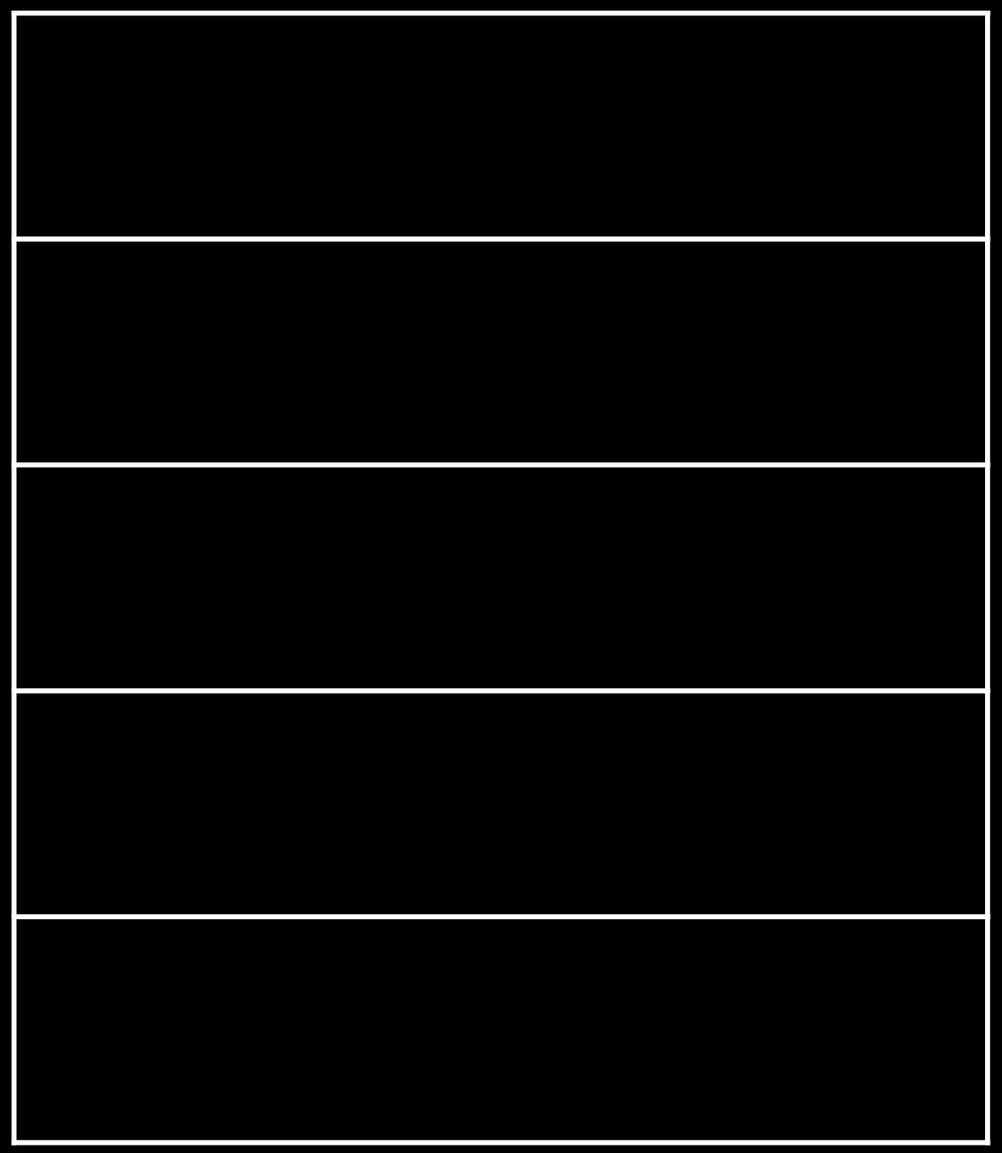
The Stack

"Once you pop, you can't stop"

The Stack

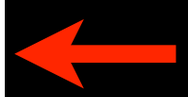
- Upon initialization, the operating system assigns a couple of segments as the stack.
- The OS also allocates stack memory for a process when it launches.
- Why a stack? Because of how imperative programs behave...
- Refresher: Calling a function consumes memory for the return address and parameters.

```c
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```

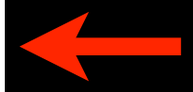


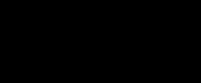
Suppose `foo()` is called.

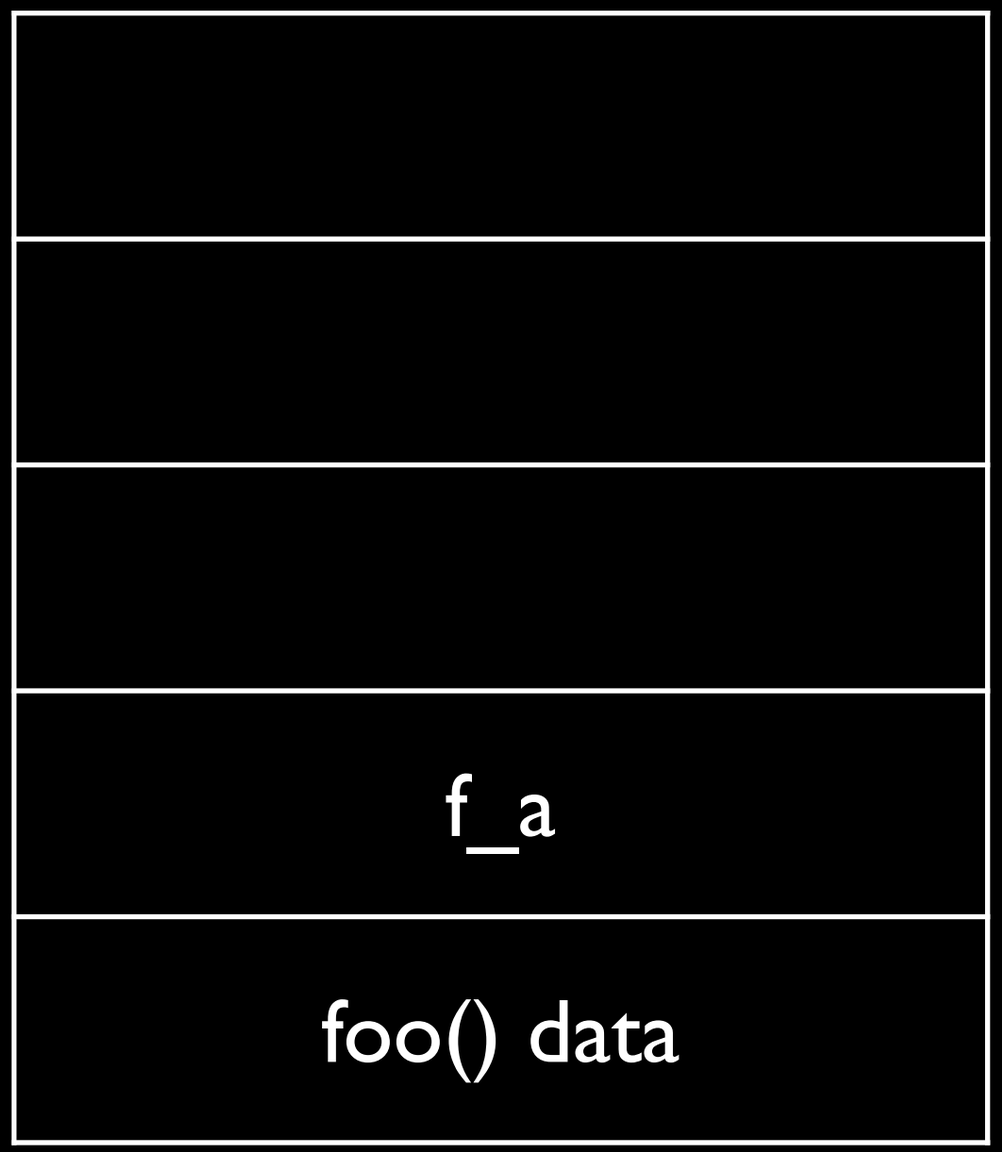
```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```



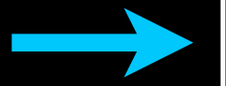
```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```

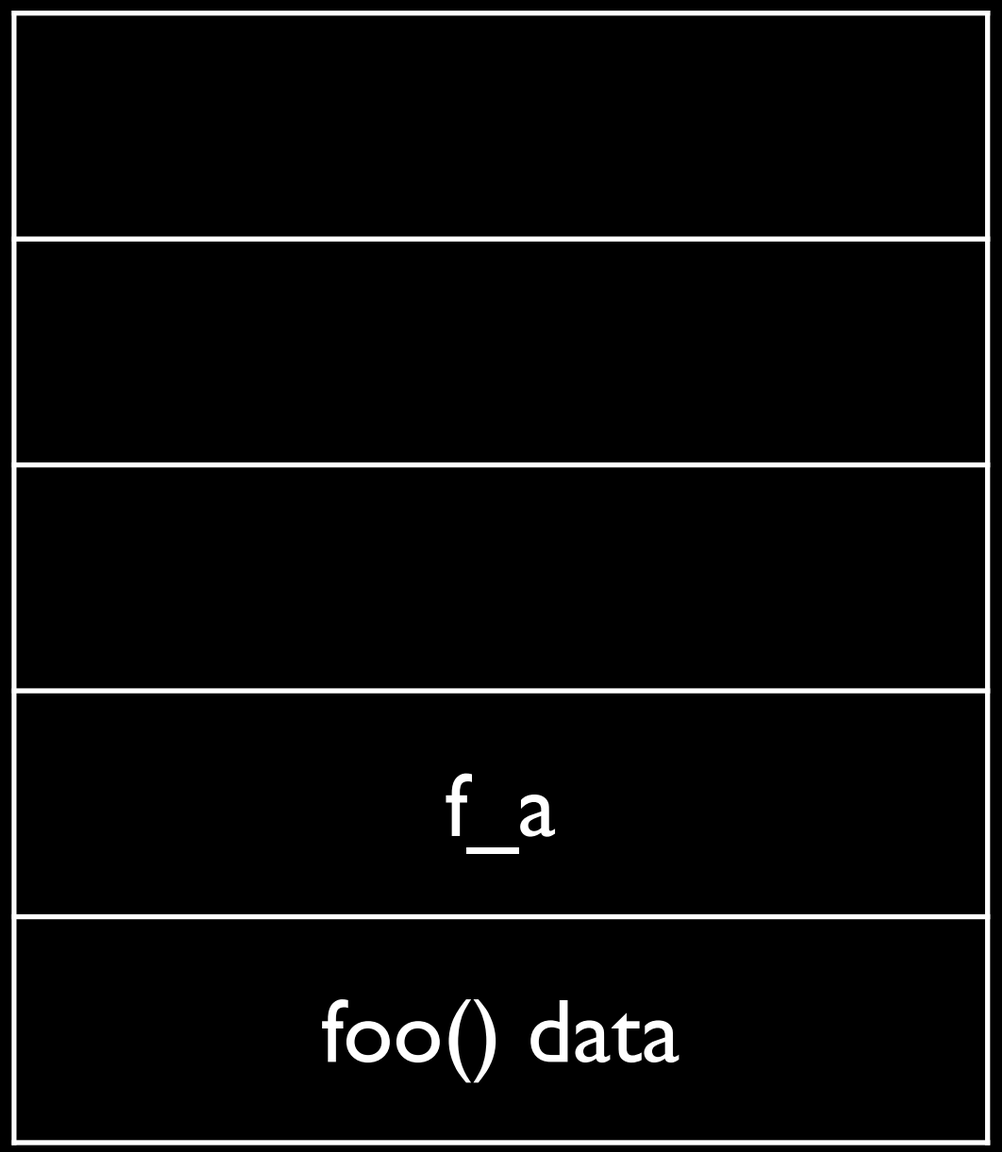




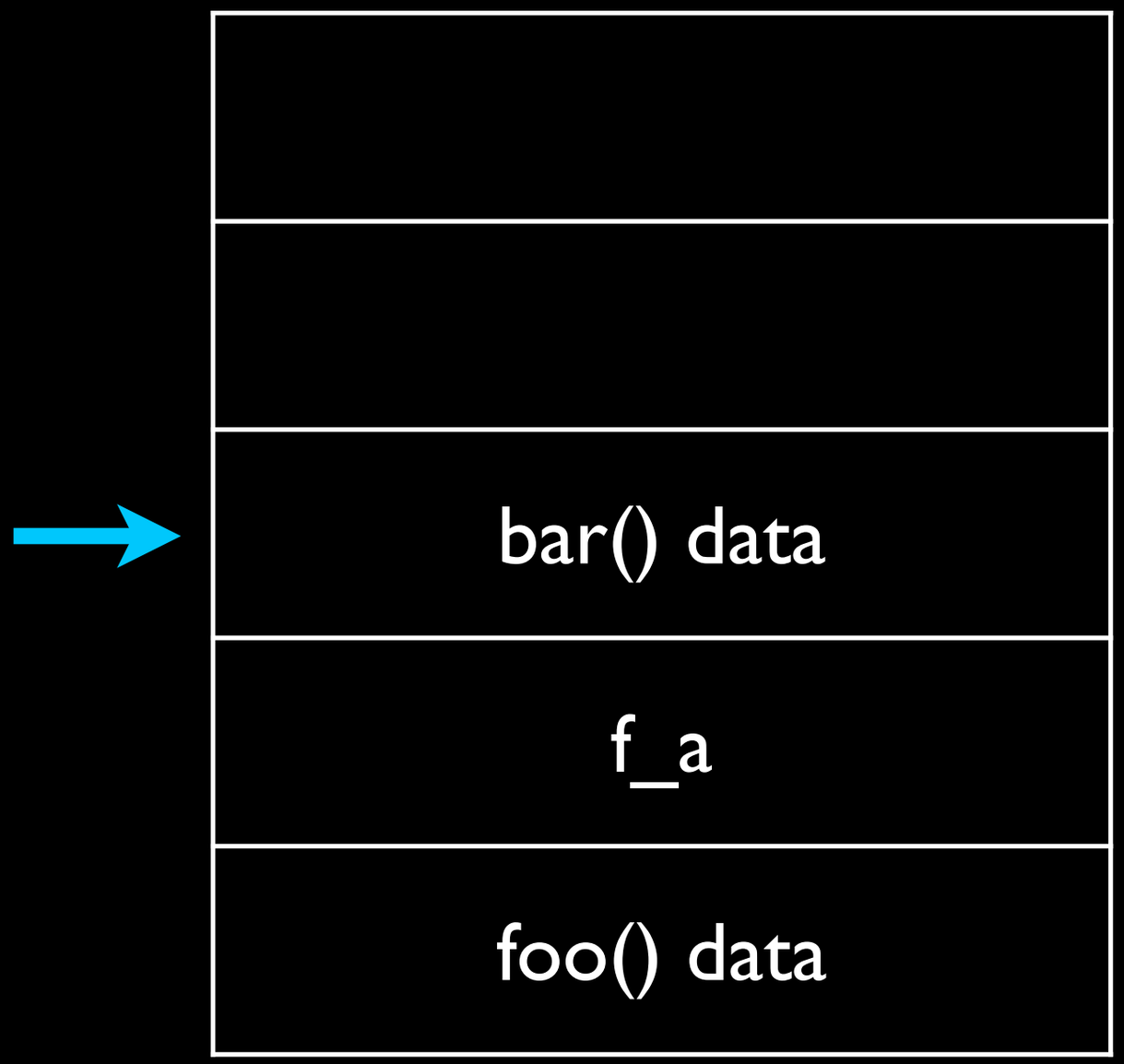


```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```

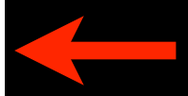


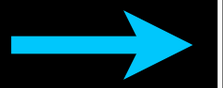


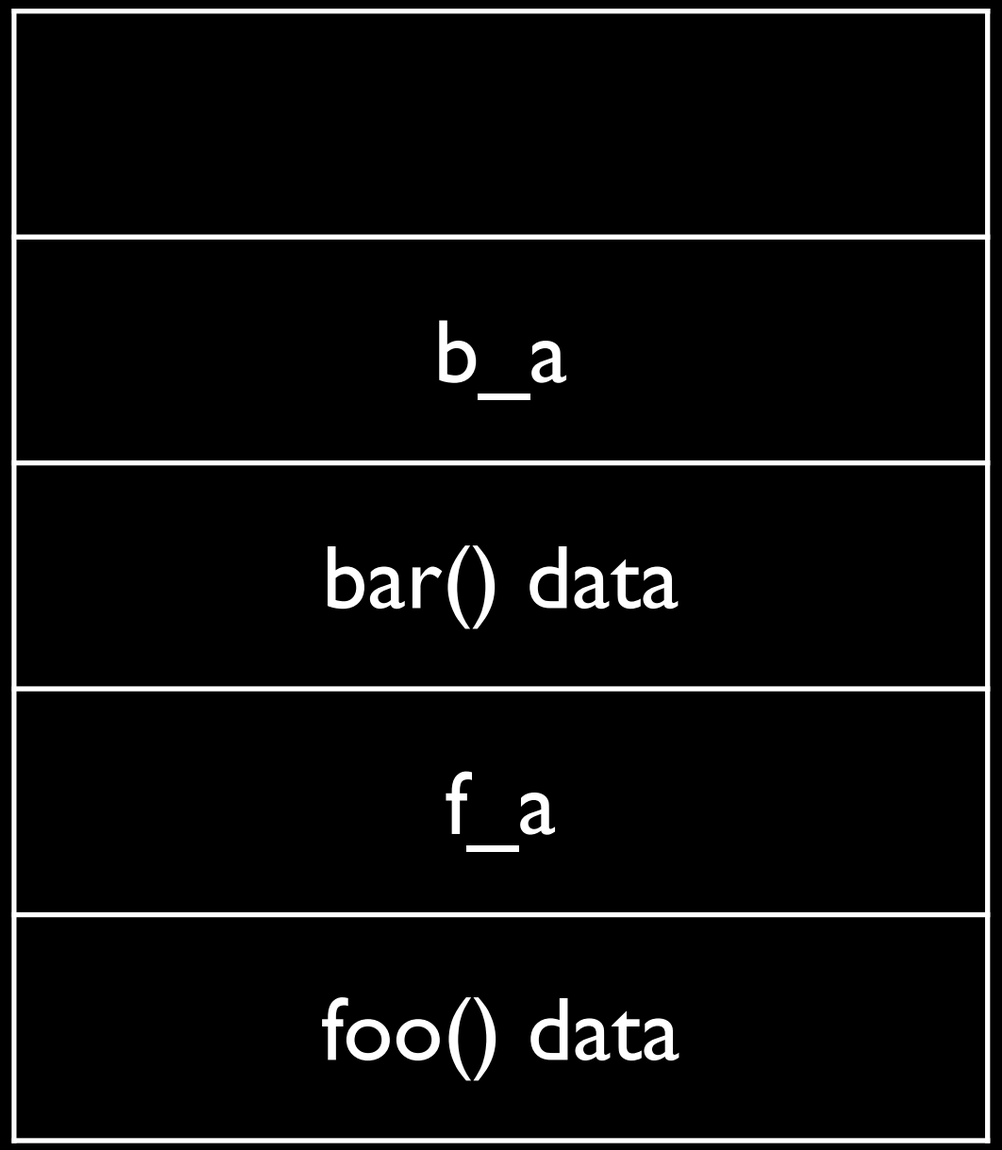
```c
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```



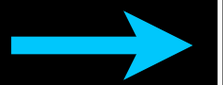
```c
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```

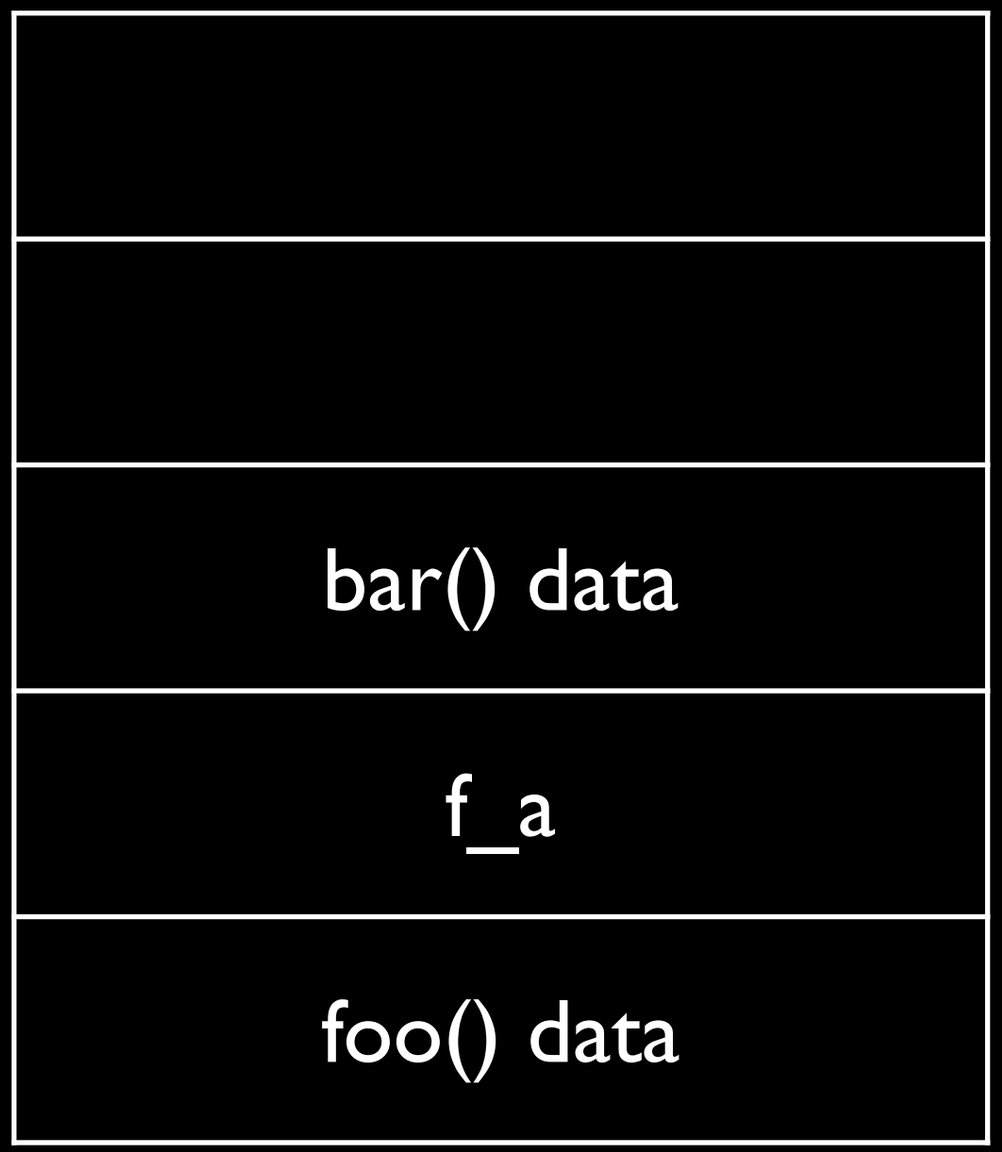




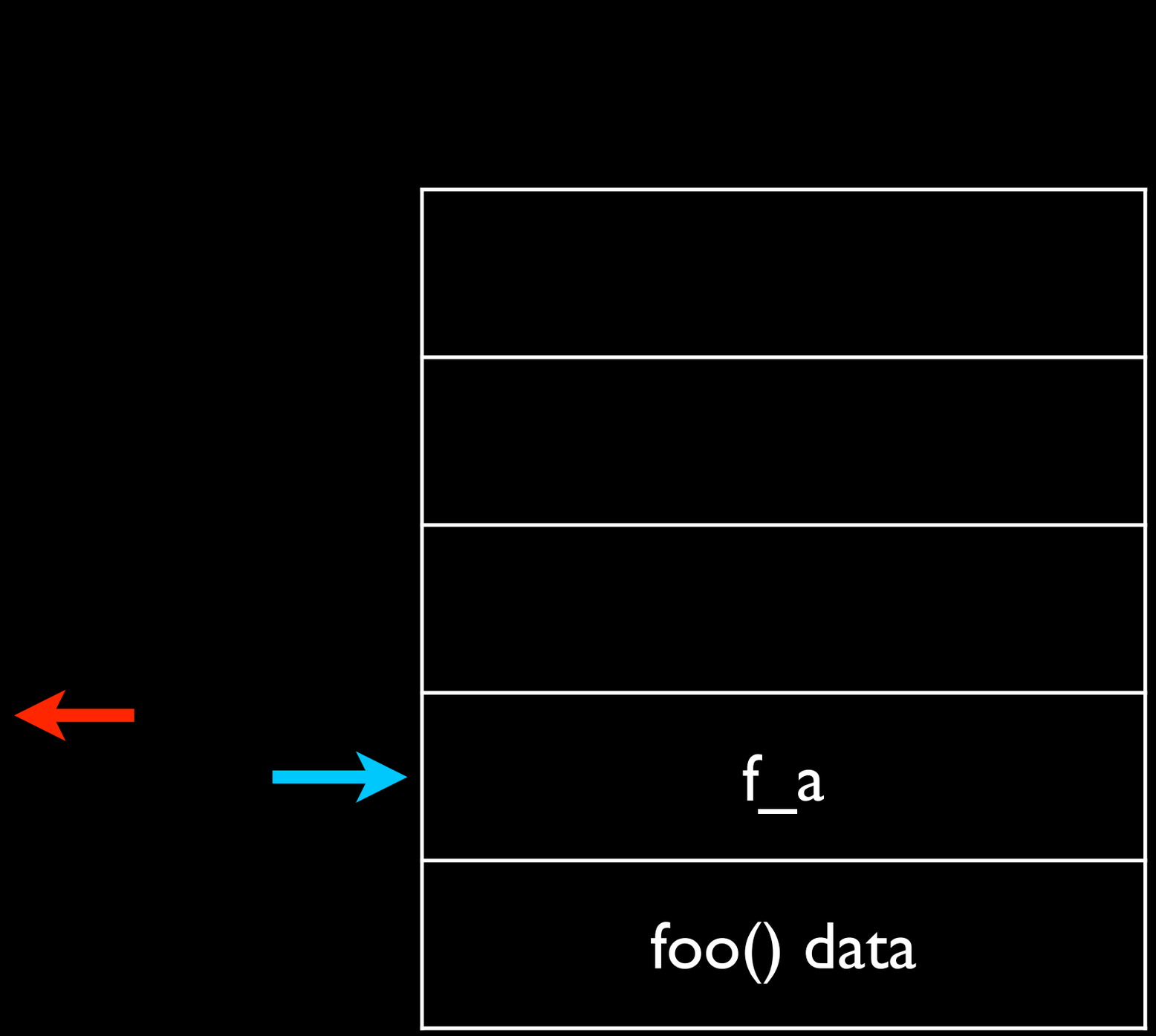


```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```

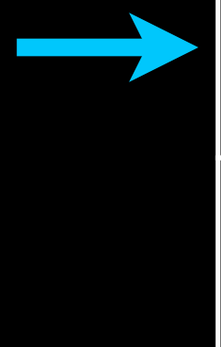


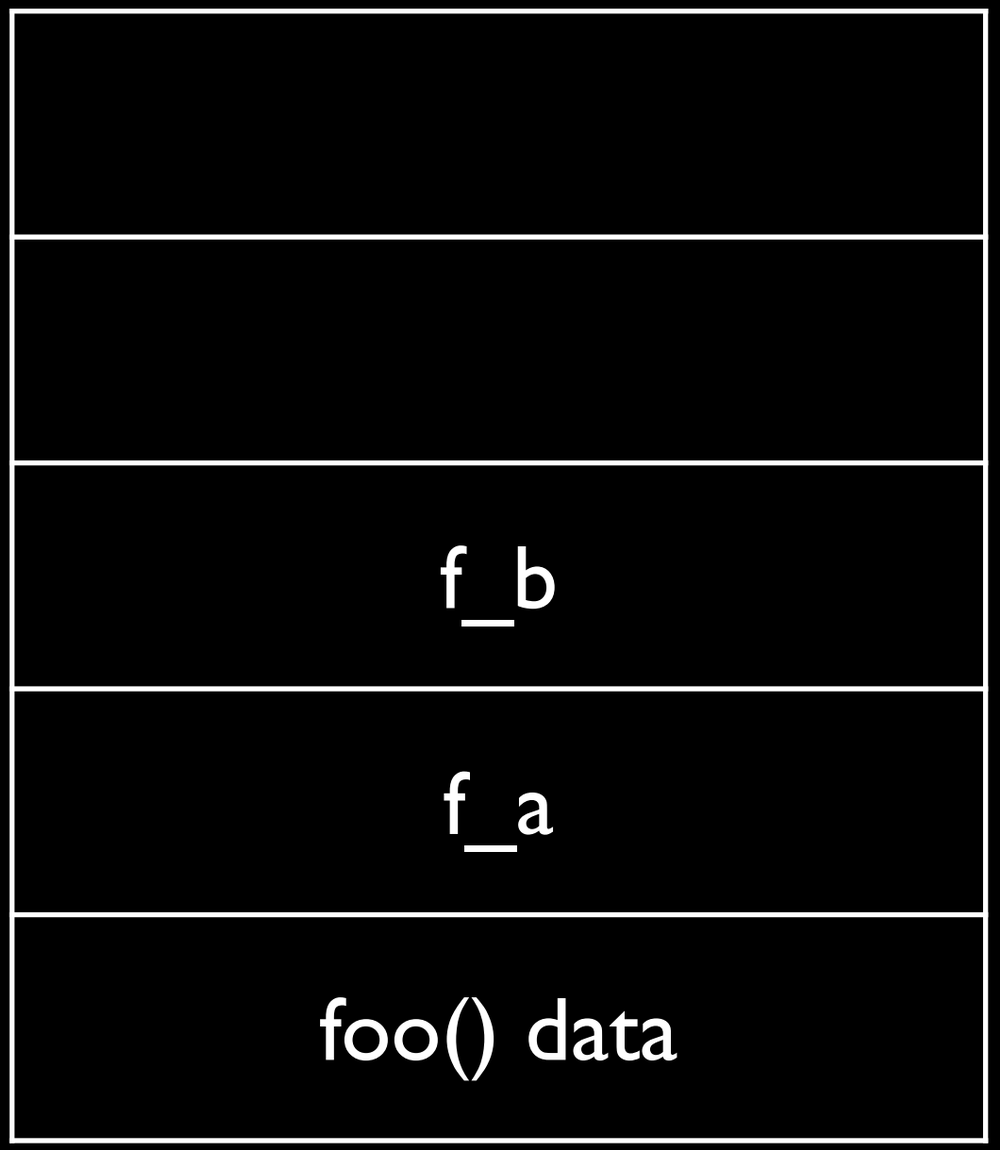


```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```

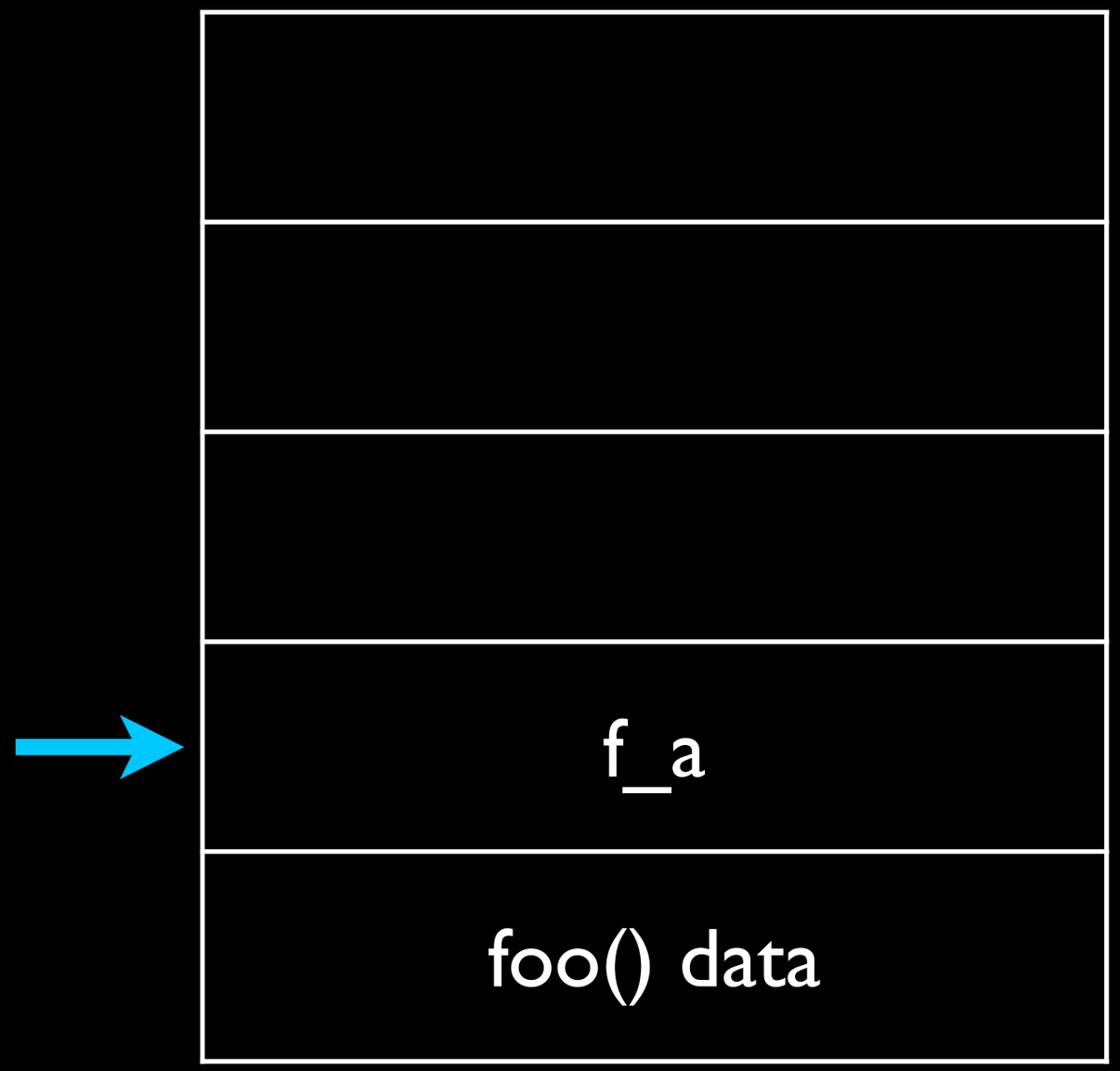


```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```

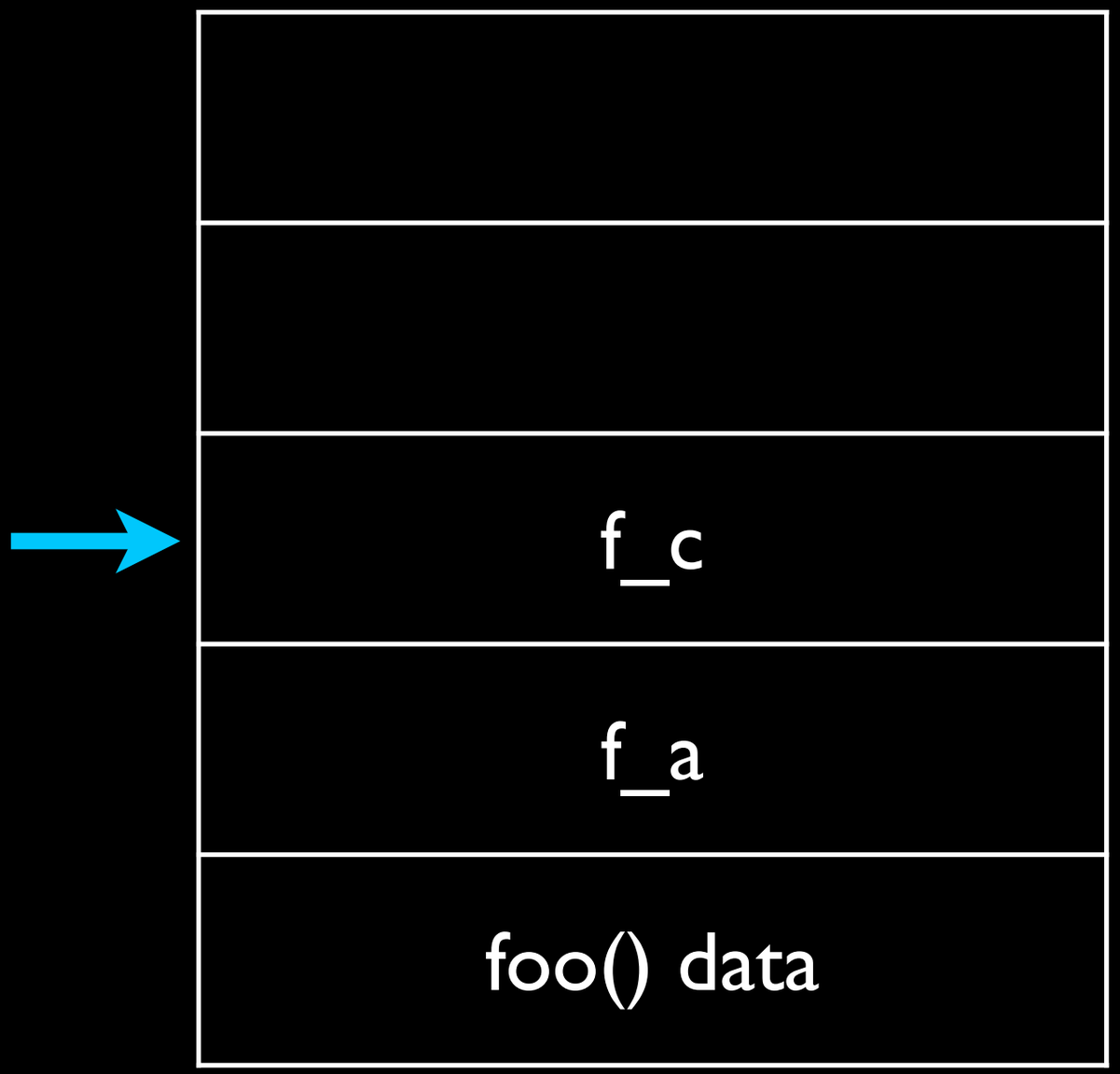




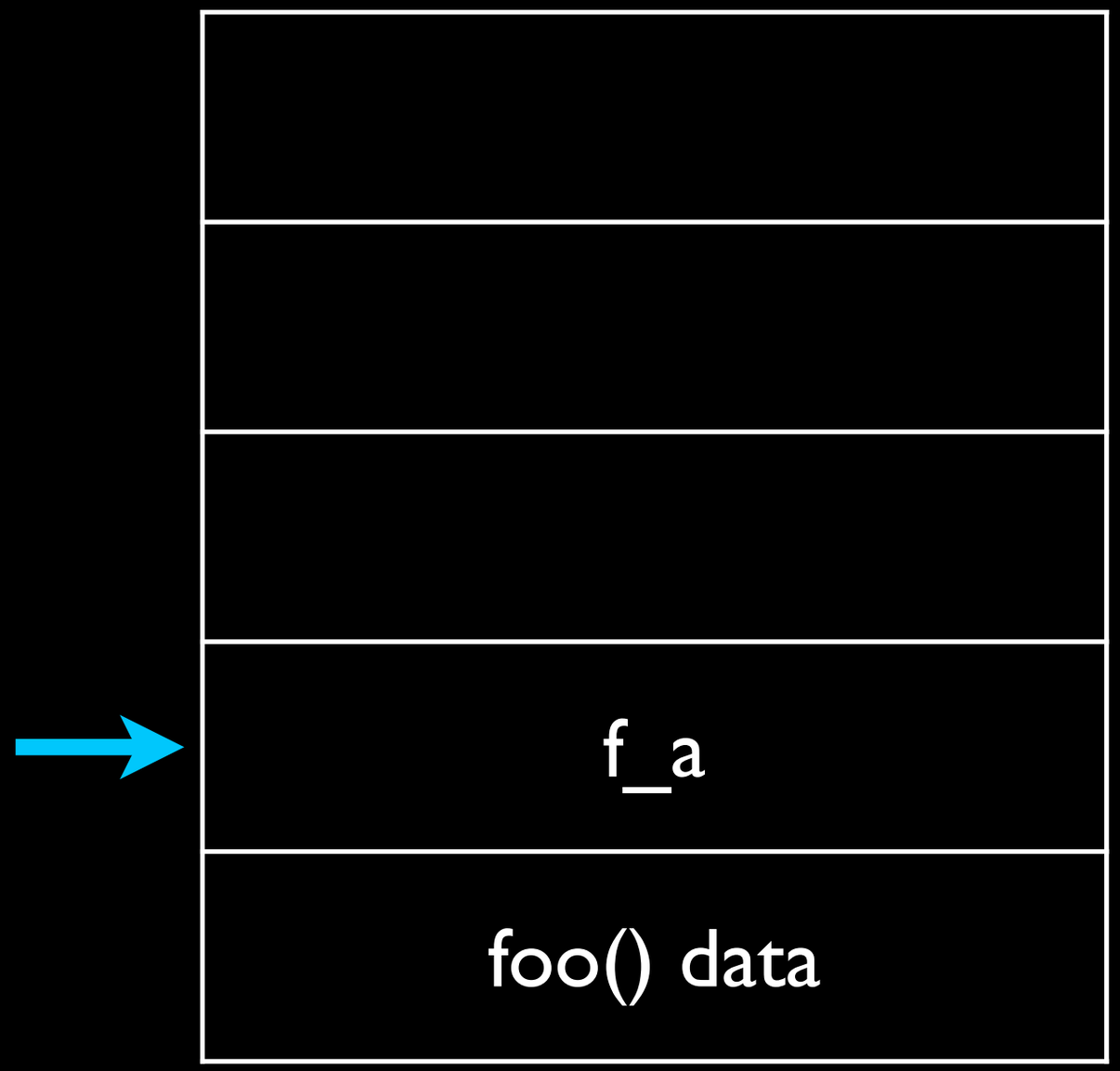
```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```



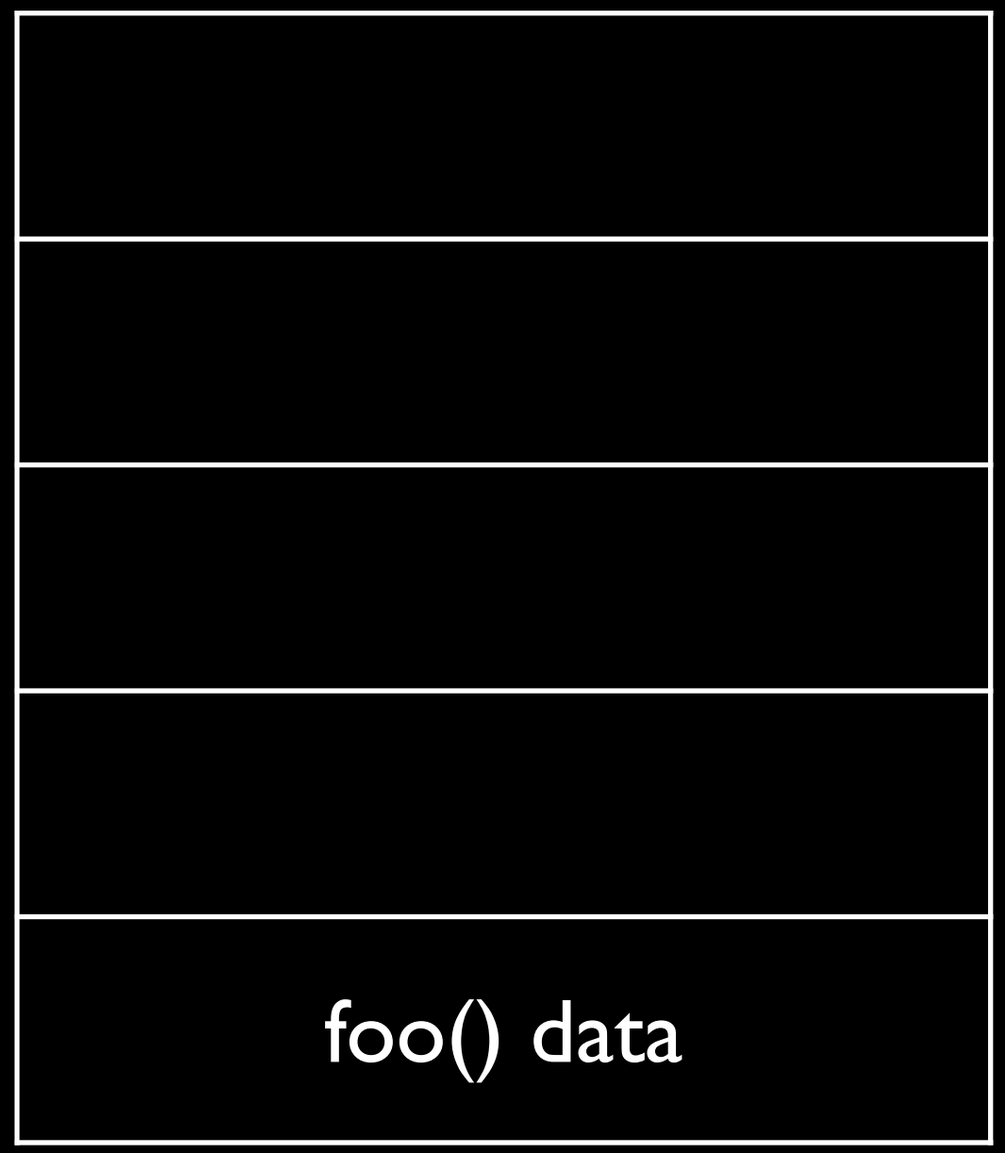
```c
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```



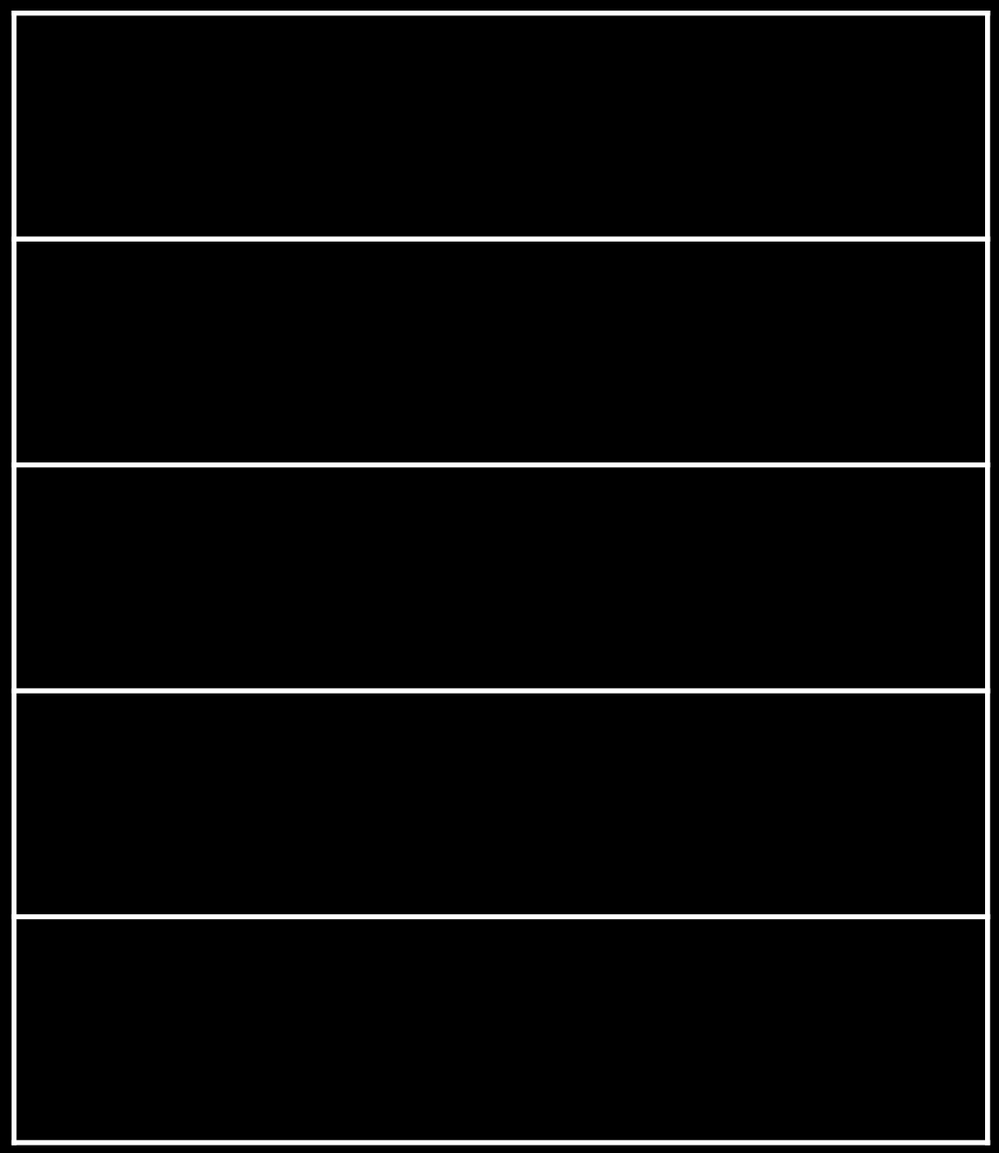
```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```



```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```

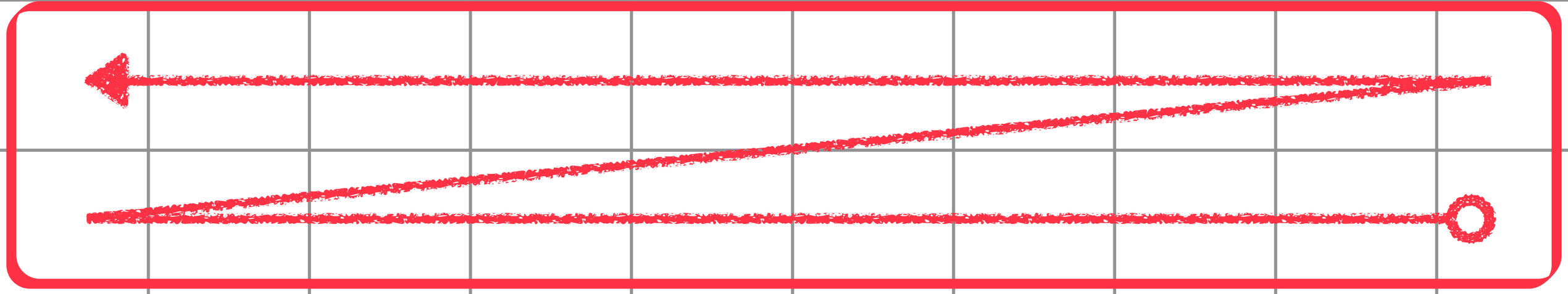


```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```



And this is why it's called *stack* memory.

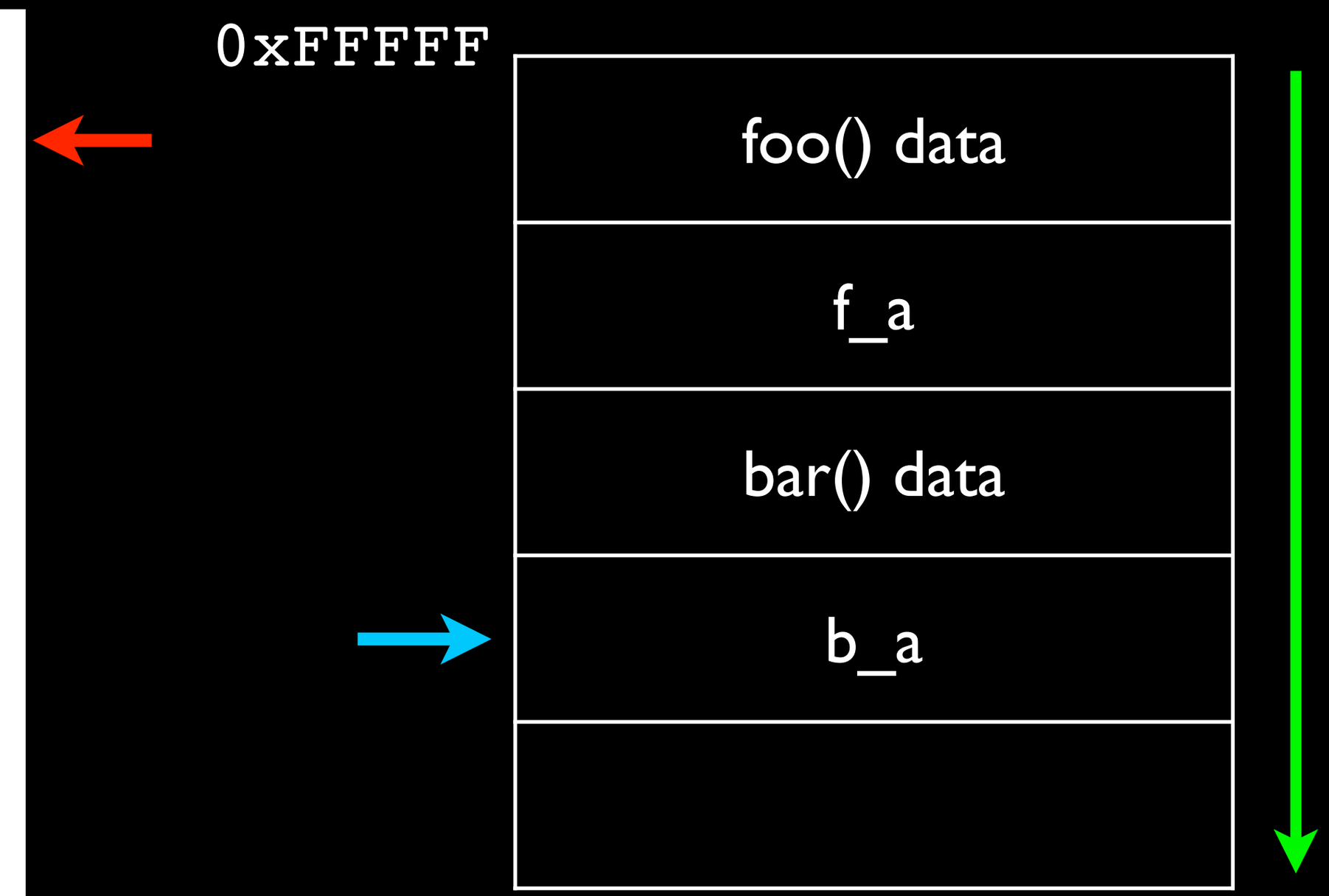
So again, the OS allocates (i.e. assigns) and area of memory as the stack...



that grows in one direction.

(note how the stack starts at the high address)

```
int bar() {
  int b_a;
}

int foo() {
  int f_a;
  bar();
  {
   int f_b;
  }
  int f_c;
}
```

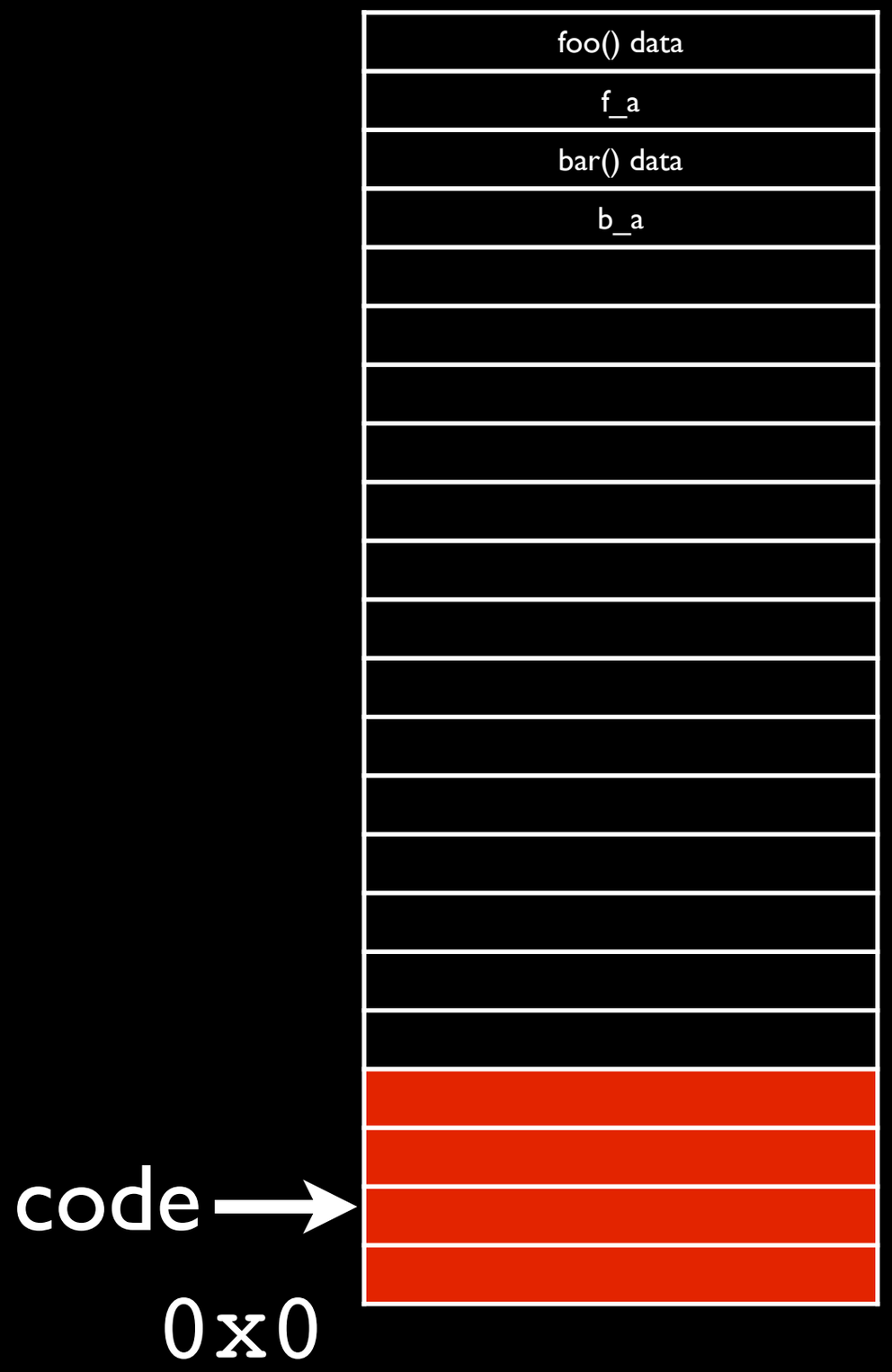


The stack actually grows *downwards* in x86
(but still remains a stack)

In real mode, code and data are located on the same segment.

If the stack grows upwards, we'll need to know the size of the code in advance (where the stack would start)

So by making the stack at the top of the memory then grow down, we need not worry about having the stack start at the proper address.



Anything else I missed?

```
int *a = new int[n];
```

The Heap

The stack handles static allocations...

Code

but cannot handle allocations that's not patterned after the stack.

Code

so a *heap* is allocated for dynamic allocations.

Code

The operating system internally manages heap allocations with varying overhead (heap metadata).

Code

More details on this when we get to memory management.

Code

Sources

- Intel Developer Manual: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-manual-325462.pdf (Homepage: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html)
- What Every Programmer Should Know About Memory (Ulrich Drepper) : http://www.akkadia.org/drepper/cpumemory.pdf