Lecture #05 - Introduction to C

AMath 483/583

Primary References

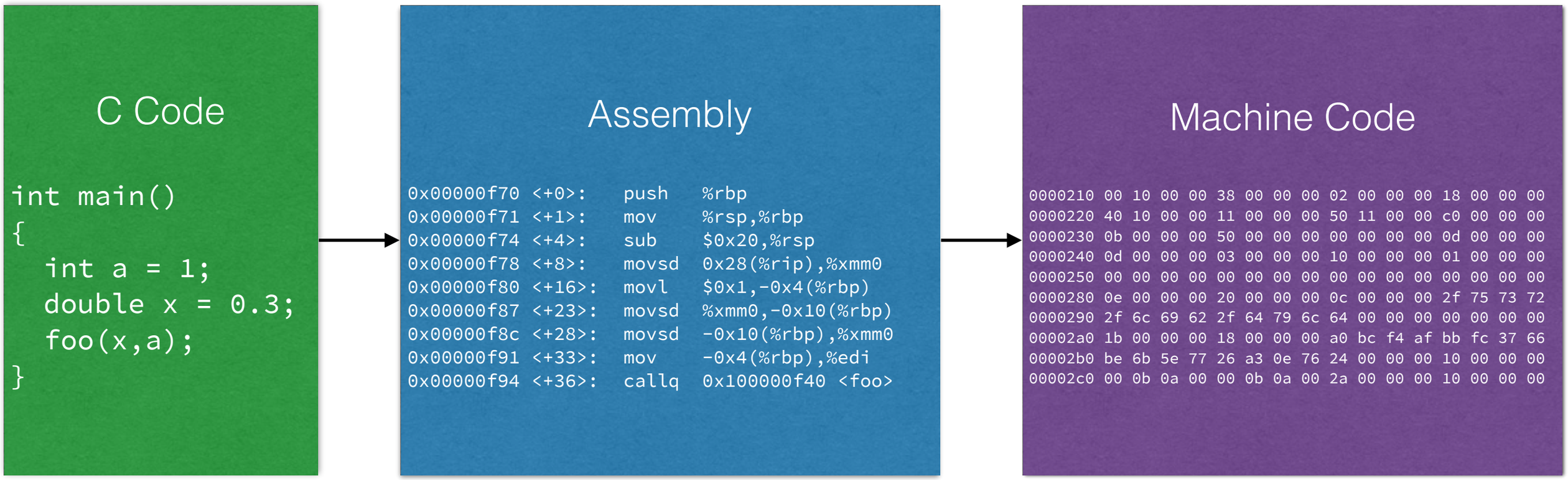
- See syllabus

Before we begin

- …quick tutorial of homework submission process.
  - writing solutions
  - testing solutions
  - using the notebook (what it is and what it isn't)
- (Note: this will change slightly with C projects.)

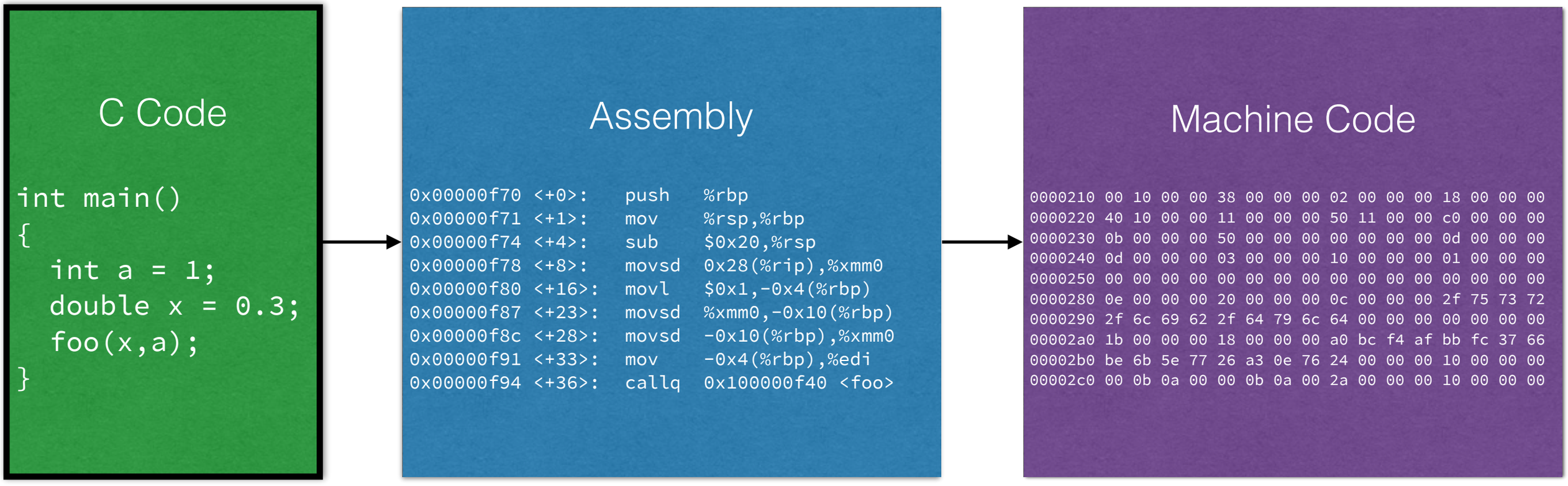
Compiled Programming Languages

- Python is “interpreted” - executed line by line
- Compiled Languages - faster, more efficient



Compiled Programming Languages

- C Code - human readable (in due time)



Compiled Programming Languages

- Assembly - instructions carried out by hardware
- Near 1-1 correspondence with CPU instructions

C Code

```
int main()
{
  int a = 1;
  double x = 0.3;
  foo(x,a);
}
```

Assembly

```
0x00000f70 <+0>:    push   %rbp
0x00000f71 <+1>:    mov    %rsp,%rbp
0x00000f74 <+4>:    sub    $0x20,%rsp
0x00000f78 <+8>:    movsd  0x28(%rip),%xmm0
0x00000f80 <+16>:   movl   $0x1,-0x4(%rbp)
0x00000f87 <+23>:   movsd  %xmm0,-0x10(%rbp)
0x00000f8c <+28>:   movsd  -0x10(%rbp),%xmm0
0x00000f91 <+33>:   mov    -0x4(%rbp),%edi
0x00000f94 <+36>:   callq  0x100000f40 <foo>
```

Machine Code

```
0000210 00 10 00 00 38 00 00 00 02 00 00 00 18 00 00 00
0000220 40 10 00 00 11 00 00 00 50 11 00 00 c0 00 00 00
0000230 0b 00 00 00 50 00 00 00 00 00 00 00 0d 00 00 00
0000240 0d 00 00 00 03 00 00 00 10 00 00 00 01 00 00 00
0000250 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000280 0e 00 00 00 20 00 00 00 0c 00 00 00 2f 75 73 72
0000290 2f 6c 69 62 2f 64 79 6c 64 00 00 00 00 00 00 00
00002a0 1b 00 00 00 18 00 00 00 a0 bc f4 af bb fc 37 66
00002b0 be 6b 5e 77 26 a3 0e 76 24 00 00 00 10 00 00 00
00002c0 00 0b 0a 00 00 0b 0a 00 2a 00 00 00 10 00 00 00
```

Compiled Programming Languages

- Machine Code - lowest level representation of code
- Literal repr. of instructions and data (in hex)

C Code

```
int main()
{
  int a = 1;
  double x = 0.3;
  foo(x,a);
}
```

Assembly

```
0x00000f70 <+0>:    push   %rbp
0x00000f71 <+1>:    mov    %rsp,%rbp
0x00000f74 <+4>:    sub    $0x20,%rsp
0x00000f78 <+8>:    movsd  0x28(%rip),%xmm0
0x00000f80 <+16>:   movl   $0x1,-0x4(%rbp)
0x00000f87 <+23>:   movsd  %xmm0,-0x10(%rbp)
0x00000f8c <+28>:   movsd  -0x10(%rbp),%xmm0
0x00000f91 <+33>:   mov    -0x4(%rbp),%edi
0x00000f94 <+36>:   callq  0x100000f40 <foo>
```

Machine Code

```
0000210 00 10 00 00 38 00 00 00 02 00 00 00 18 00 00 00
0000220 40 10 00 00 11 00 00 00 50 11 00 00 c0 00 00 00
0000230 0b 00 00 00 50 00 00 00 00 00 00 00 0d 00 00 00
0000240 0d 00 00 00 03 00 00 00 10 00 00 00 01 00 00 00
0000250 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000280 0e 00 00 00 20 00 00 00 0c 00 00 00 2f 75 73 72
0000290 2f 6c 69 62 2f 64 79 6c 64 00 00 00 00 00 00 00
00002a0 1b 00 00 00 18 00 00 00 a0 bc f4 af bb fc 37 66
00002b0 be 6b 5e 77 26 a3 0e 76 24 00 00 00 10 00 00 00
00002c0 00 0b 0a 00 00 0b 0a 00 2a 00 00 00 10 00 00 00
```

C Compilers

- Compilers turn C code into machine code
- C compilers:
  - free: gcc, clang
  - paid: icc (Intel), cl (Microsoft), pgcc (Portland Group / NVIDIA)

C Compilers

- Using gcc:

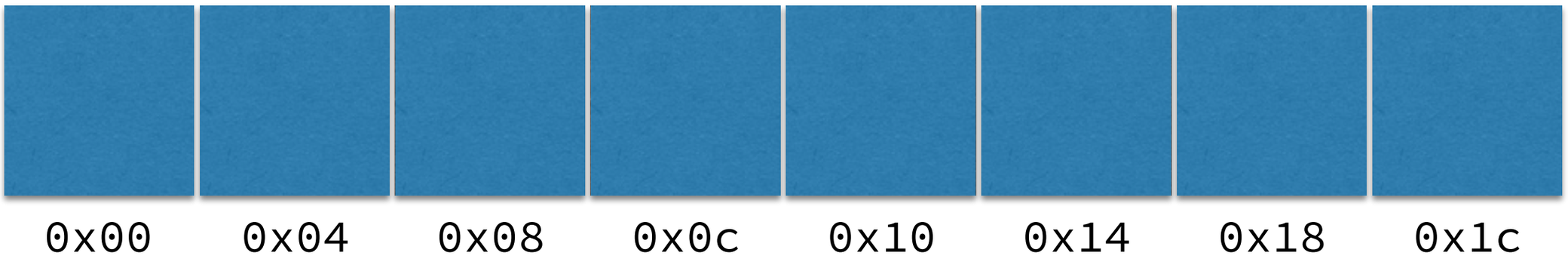
```
$ gcc mysource.c -o output_binary

$ ./output_binary

Hello, world!
```

Demo

Basics of C, Compiling C Code

Pointers

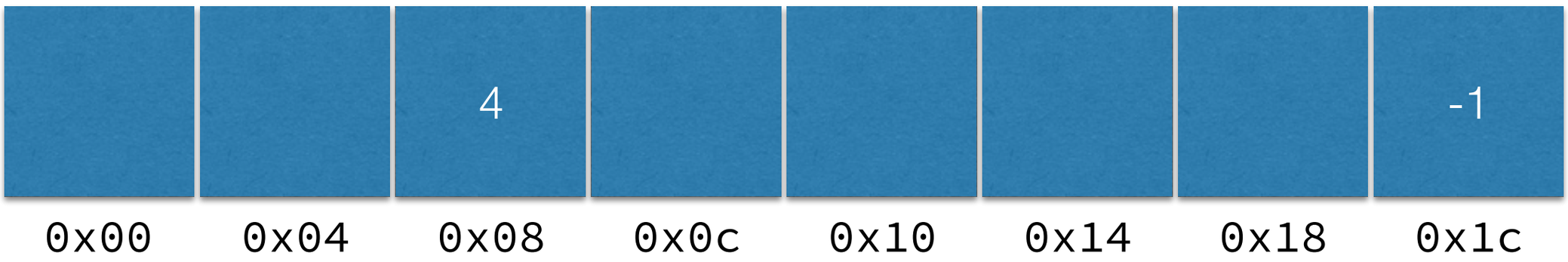
- Pointers are challenging to understand at first
- Makes more sense when you think about memory



RAM: imagine each block = 4 bytes (an int)

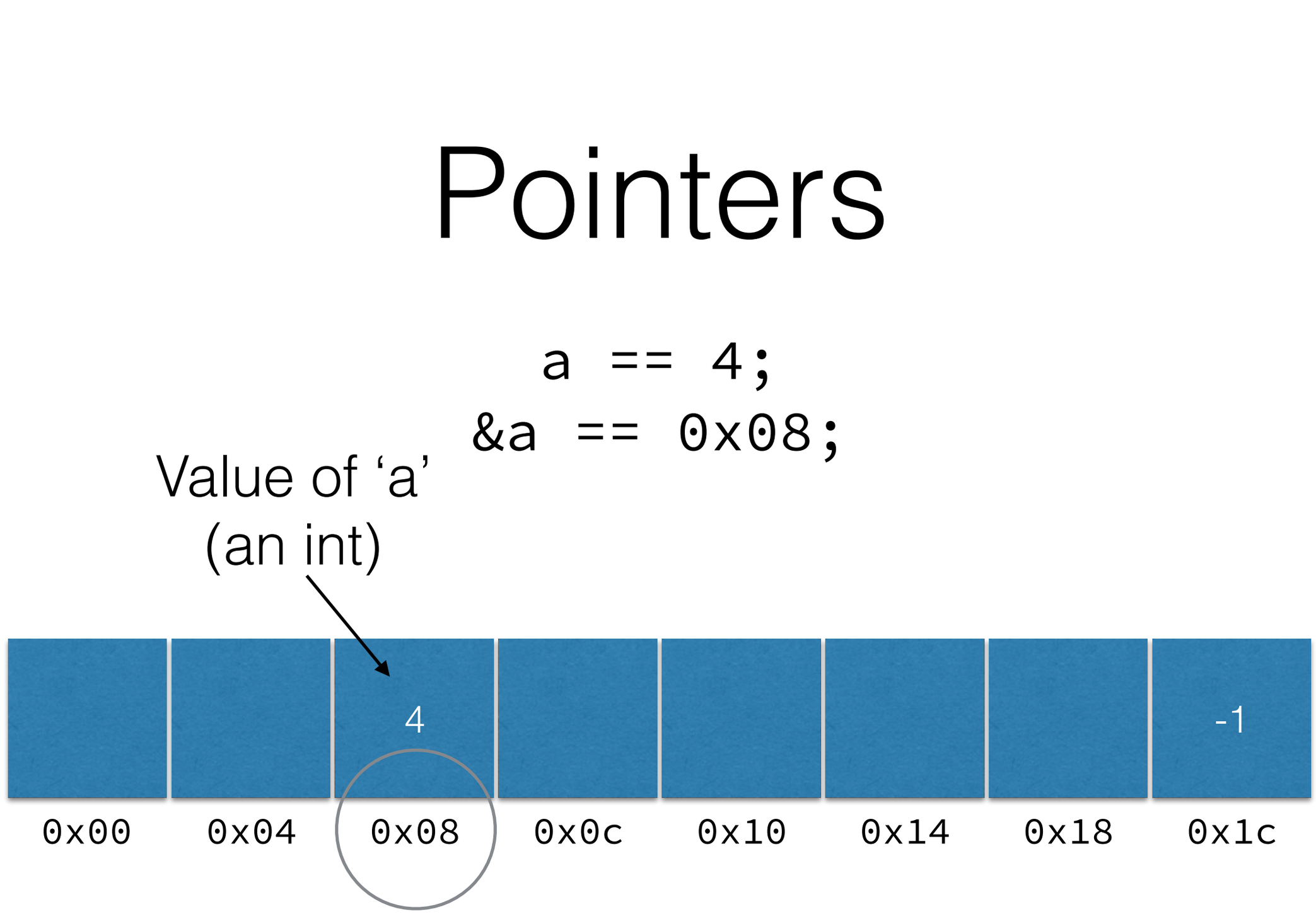
Pointers

```
int a = 4;
int b = -1;
```



Pointers

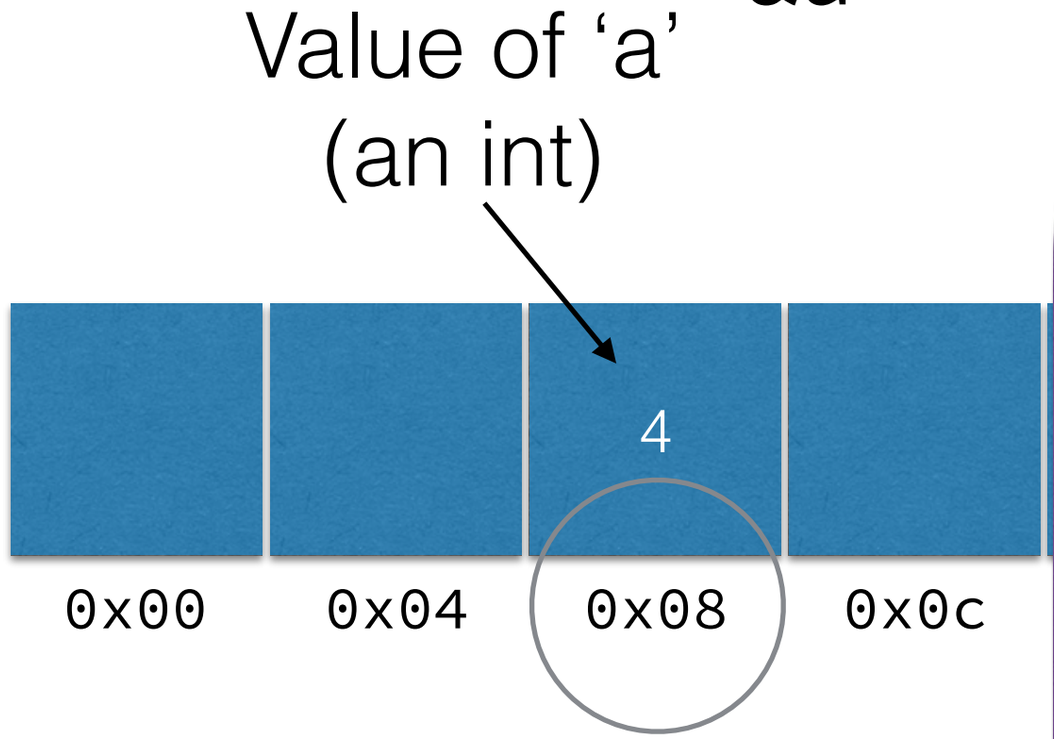
```
a == 4;
&a == 0x08;
```



Address of ‘a’

Pointers

```
a == 4;
&a == 0x08;
```



Address of ‘a’

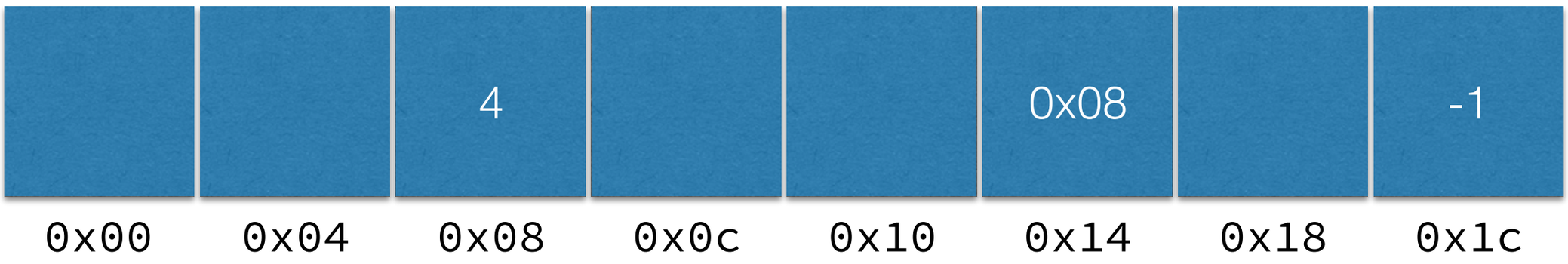
Bucket Analogy

Bucket with the label “a”.
Contains the number 4.
Sits on the shelf in position “0x08”

NEXT SLIDE: you can put shelf positions in buckets…

Pointers

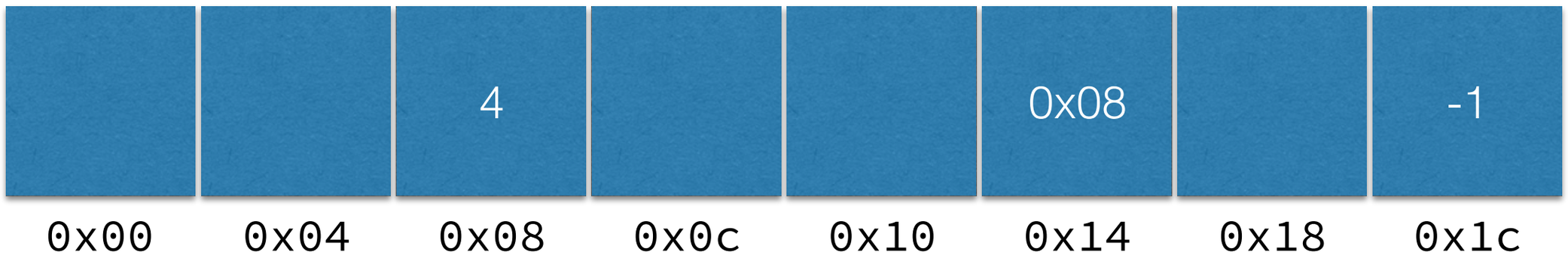
```
int* ptr = &a;
```



Pointers

`int*` `ptr = &a;`

Pointers are of type `T*`



Pointers

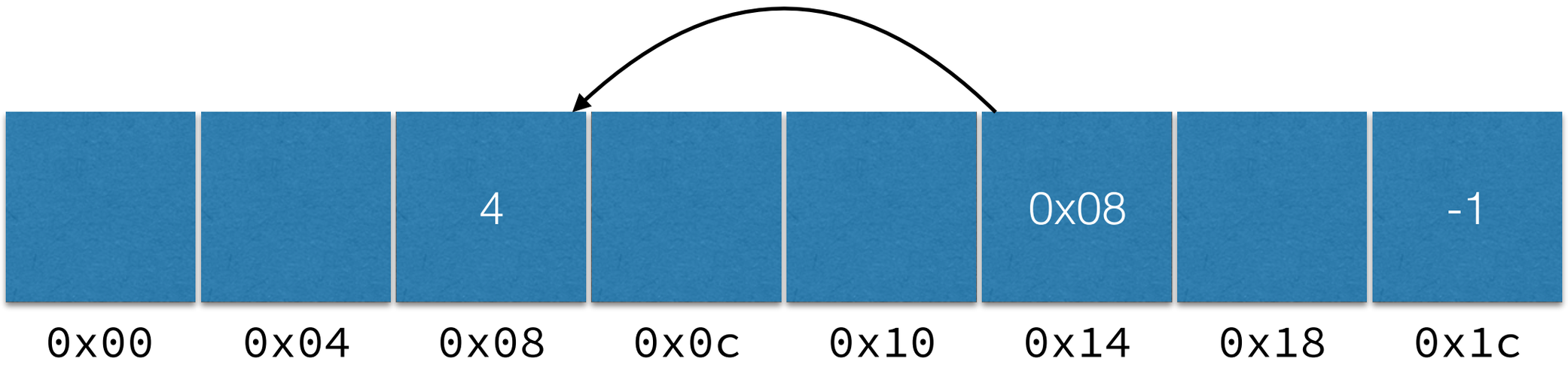
```
ptr == &a;
&ptr == 0x14;
*ptr == 4;
```

Value of `ptr` (an address)

Address of ‘ptr’
(it’s a value/variable, too, and lives in memory)

Pointers

```
ptr == &a;
&ptr == 0x14;
*ptr == 4;
```



"The thing being pointed to by `ptr` is equal to 4."
(The thing at the address represented by ptr.)

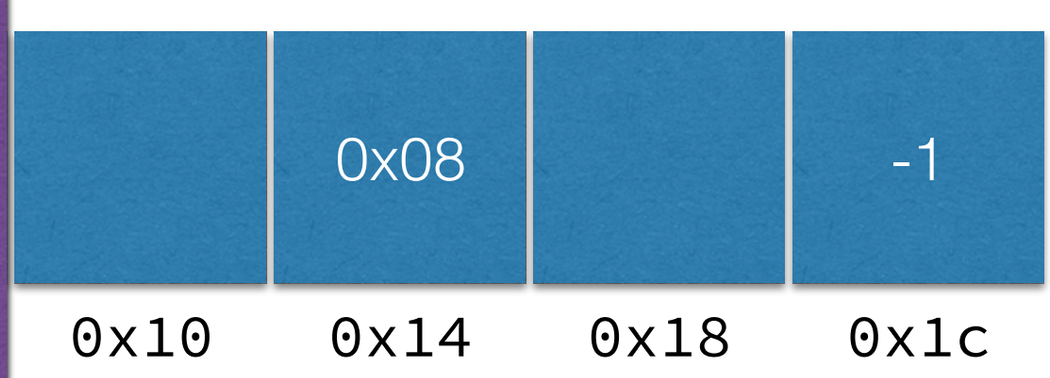
Pointers

```
ptr == &a;
&ptr == 0x14;
*ptr == 4;
```

Bucket Analogy

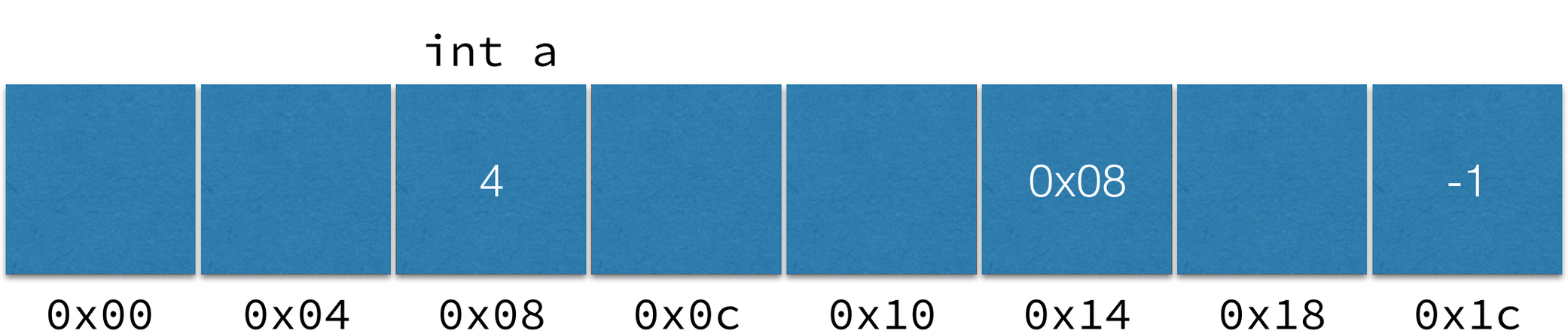
Bucket with the label "`ptr`".
Contains the number `0x08`.
Sits on the shelf in position "`0x14`"

Value in this bucket points to shelf position `0x08`



Pointers

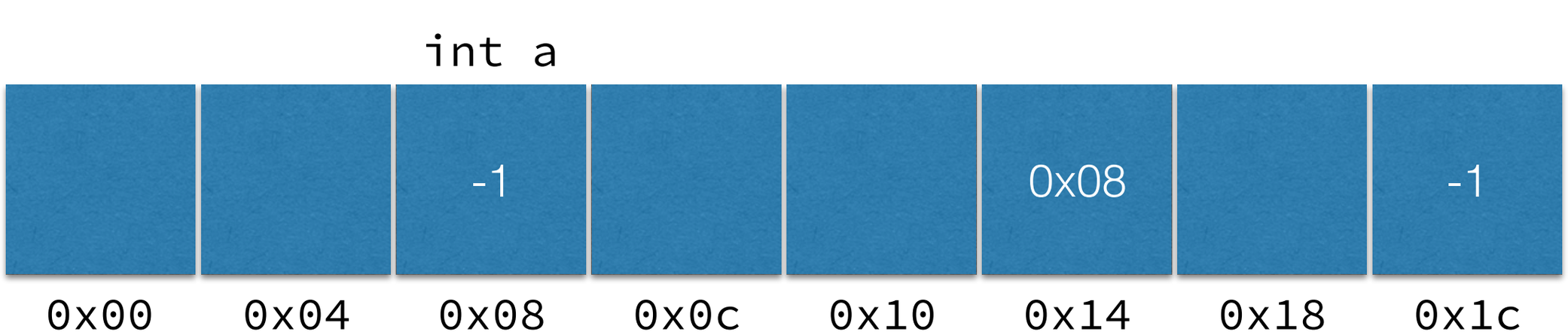
```
*ptr = -1;
```



When you have an address you can modify the contents at that address.

Pointers

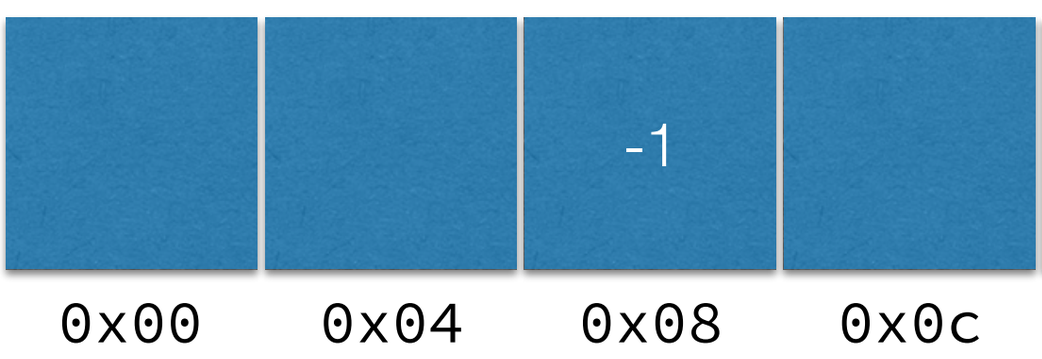
```
*ptr = -1;
```



When you have an address you can modify the contents at that address.

Pointers

```
*ptr = -1;
```



When you have an ad
the contents at

[Bucket Analogy]

`ptr` gives me the address of a bucket.

`*ptr` is the value inside the pointed to bucket.

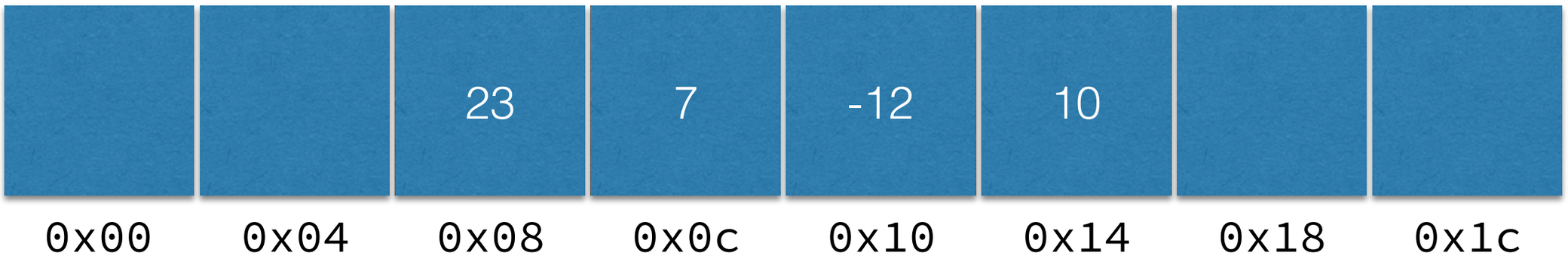
`*ptr = -1;`
Change the thing in the pointed to bucket (with label “a”) to -1.

The Dirty Secret

Arrays are pointers

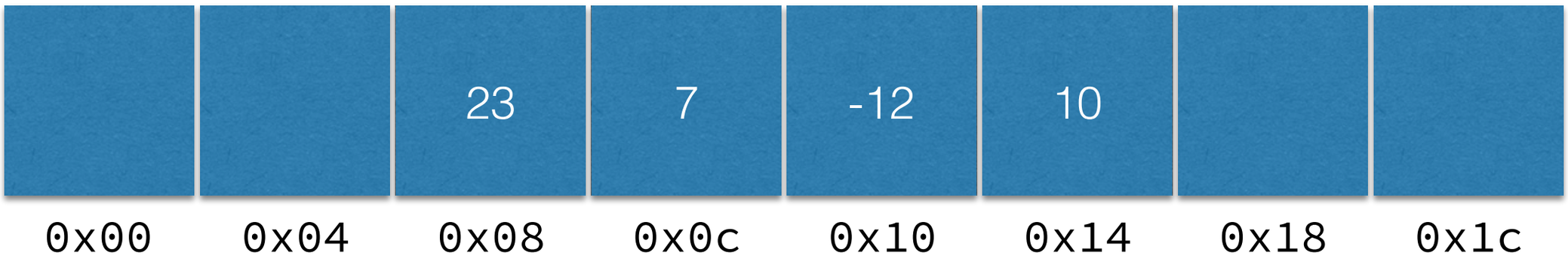
Pointers and Arrays

```
int arr[4] = {23, 7, -12, 10};
```



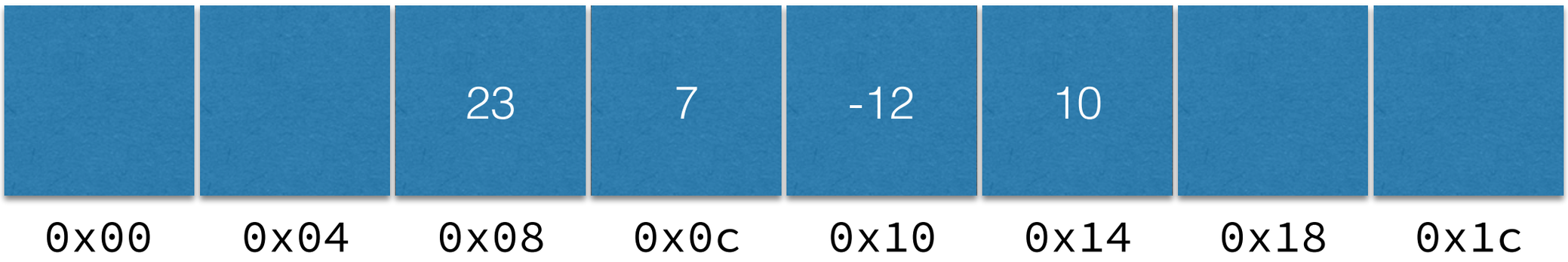
Pointers and Arrays

```
arr[0] == 23;
arr[1] == 7;
arr[2] == -12;
arr[3] == 10;
```



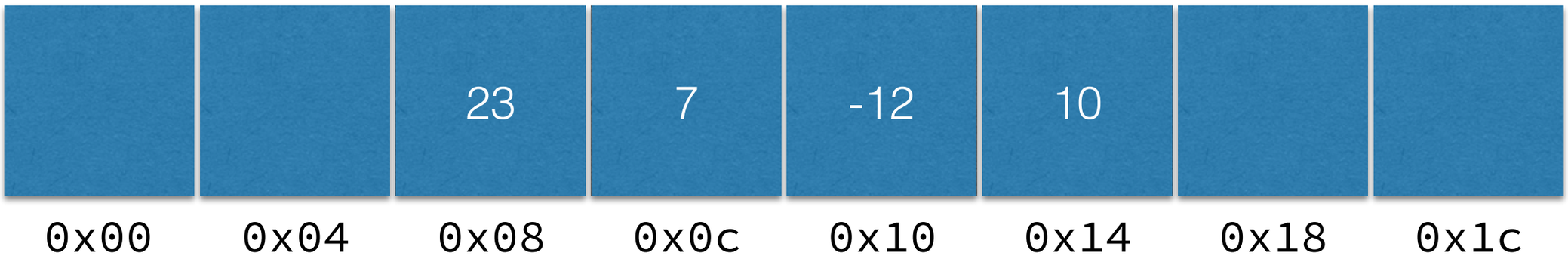
Pointers and Arrays

```
 arr == 0x08;
arr+1 == 0x0c;
arr+2 == 0x10;
arr+3 == 0x14;
```



Pointers and Arrays

```
*(arr) == 23;
*(arr+1) == 7;
*(arr+2) == -12;
*(arr+3) == 10;
```



`arr[n]` is syntactic sugar for `*(arr+n)`

Demo

Pointers and Arrays

Additional Time

- math.h - sin, cos, log, sqrt, floor, ceil, abs,
- Functions and Function prototypes
- gcc output assembly code

```
$ gcc -s mysource.c -o mysource.s
```

- Next time: dynamically allocated arrays