CS 240 Stage 2!

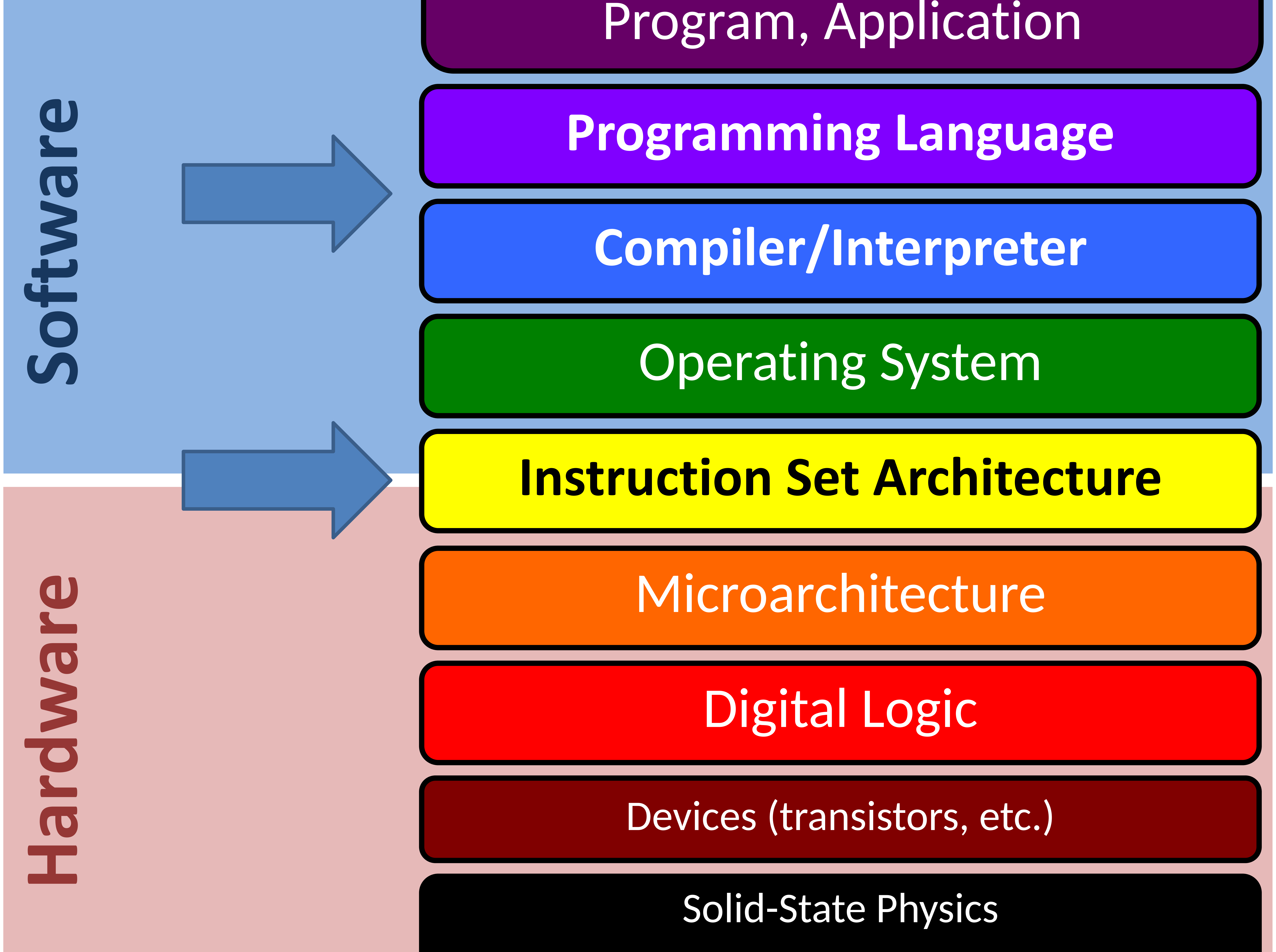
Hardware-Software Interface

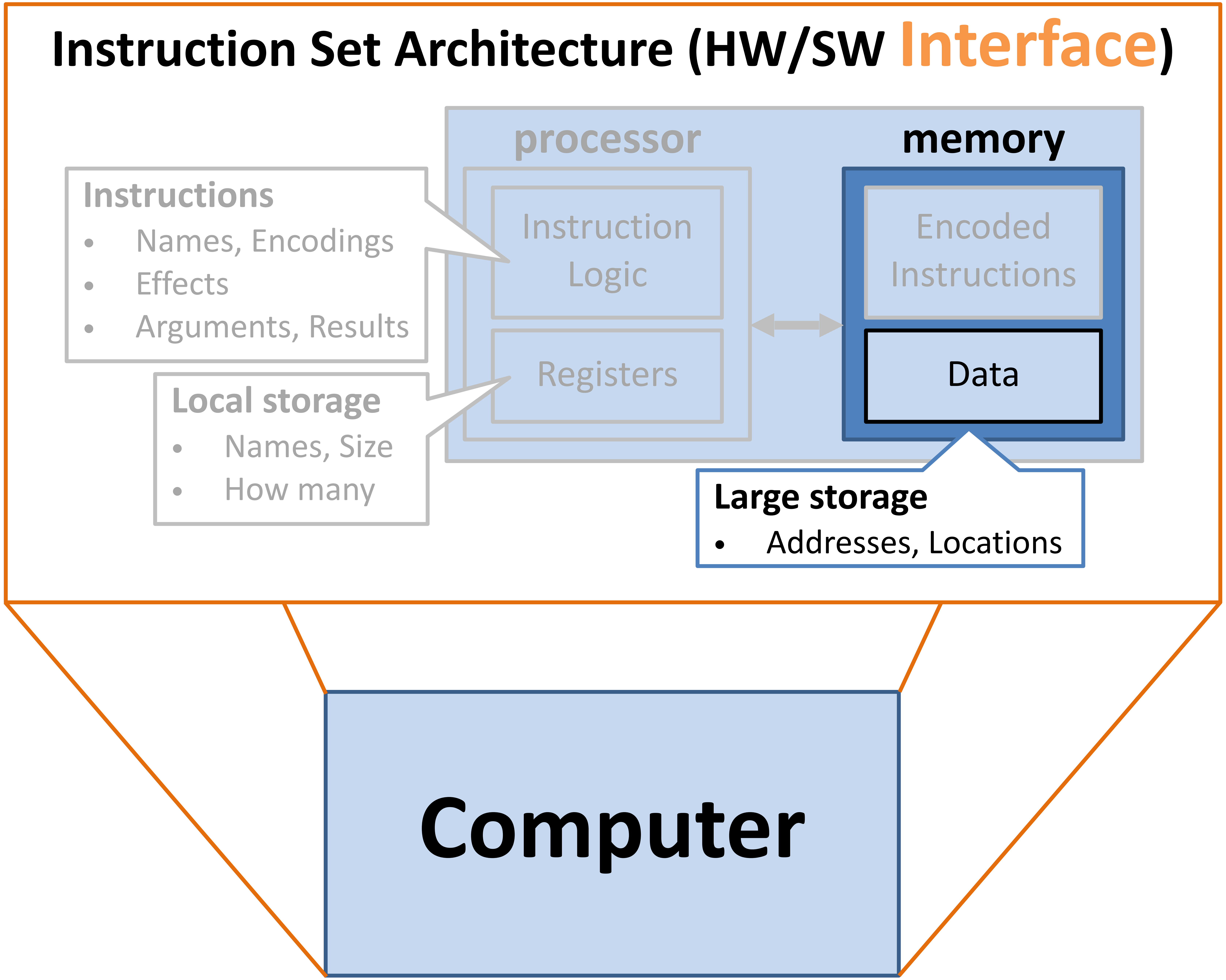
Memory addressing, C language, pointers
Assertions, debugging
Machine code, assembly language, program translation
Control flow
Procedures, stacks
Data layout, security, linking and loading

Programming with Memory

the memory model

pointers and arrays in C

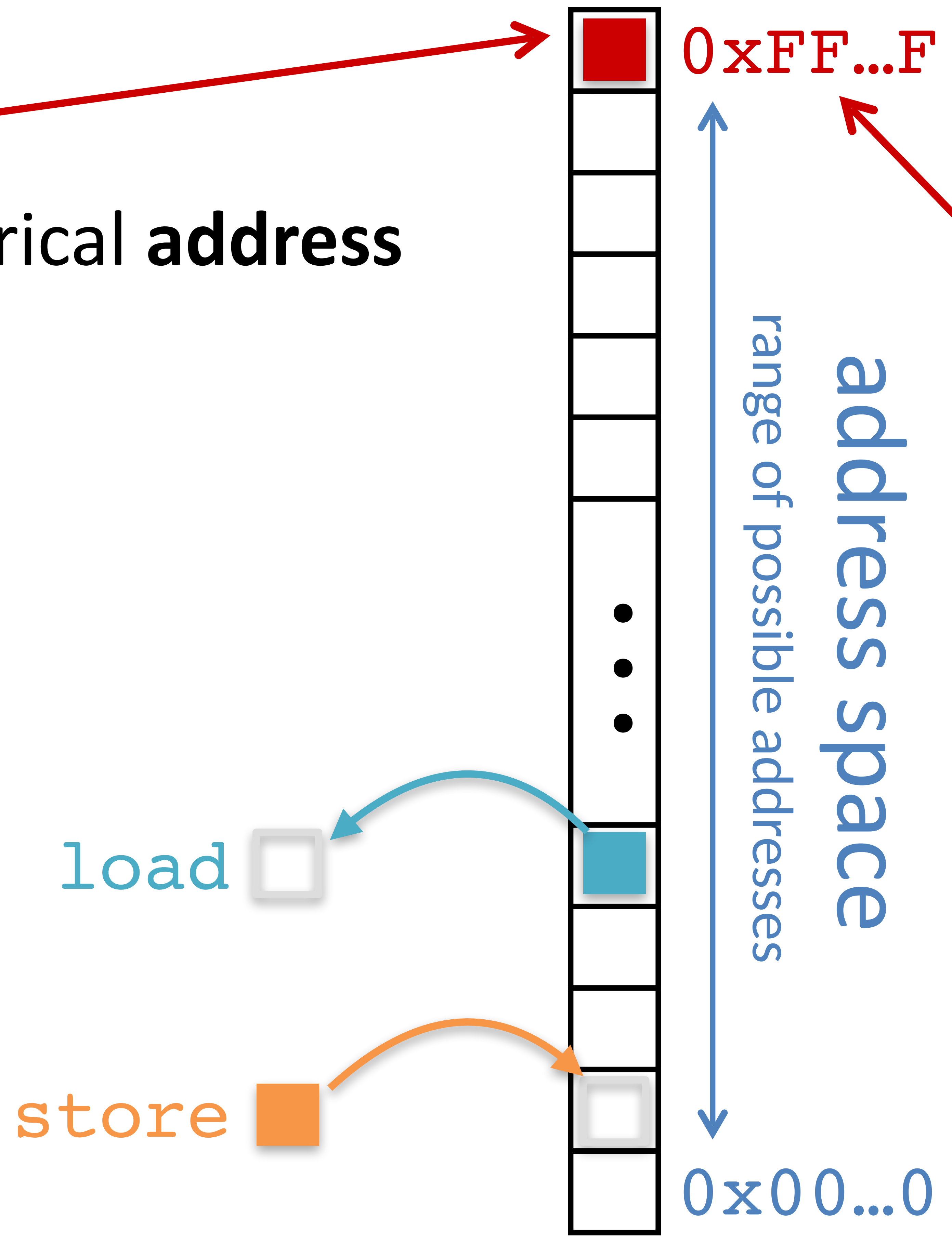

Software
Hardware
Program, Application
Programming Language
Compiler/Interpreter
Operating System
Instruction Set Architecture
Microarchitecture
Digital Logic
Devices (transistors, etc.)
Solid-State Physics


Instruction Set Architecture (HW/SW Interface)
processor
memory
Instructions
Names, Encodings
Effects
Arguments, Results
Instruction Logic
Encoded Instructions
Registers
Data
Local storage
Names, Size
How many
Large storage
Addresses, Locations
Computer

Byte-addressable memory = mutable byte array

Location / cell = element

- Identified by unique numerical **address**
- Holds **one byte (8 bits)**



Address = index

- Unsigned number
- Represented by one word
- Computable and storable as a value

Operations:

- **Load:** read contents at given address
- **Store:** write contents at given address

Multi-byte values in memory

Store across contiguous byte locations.

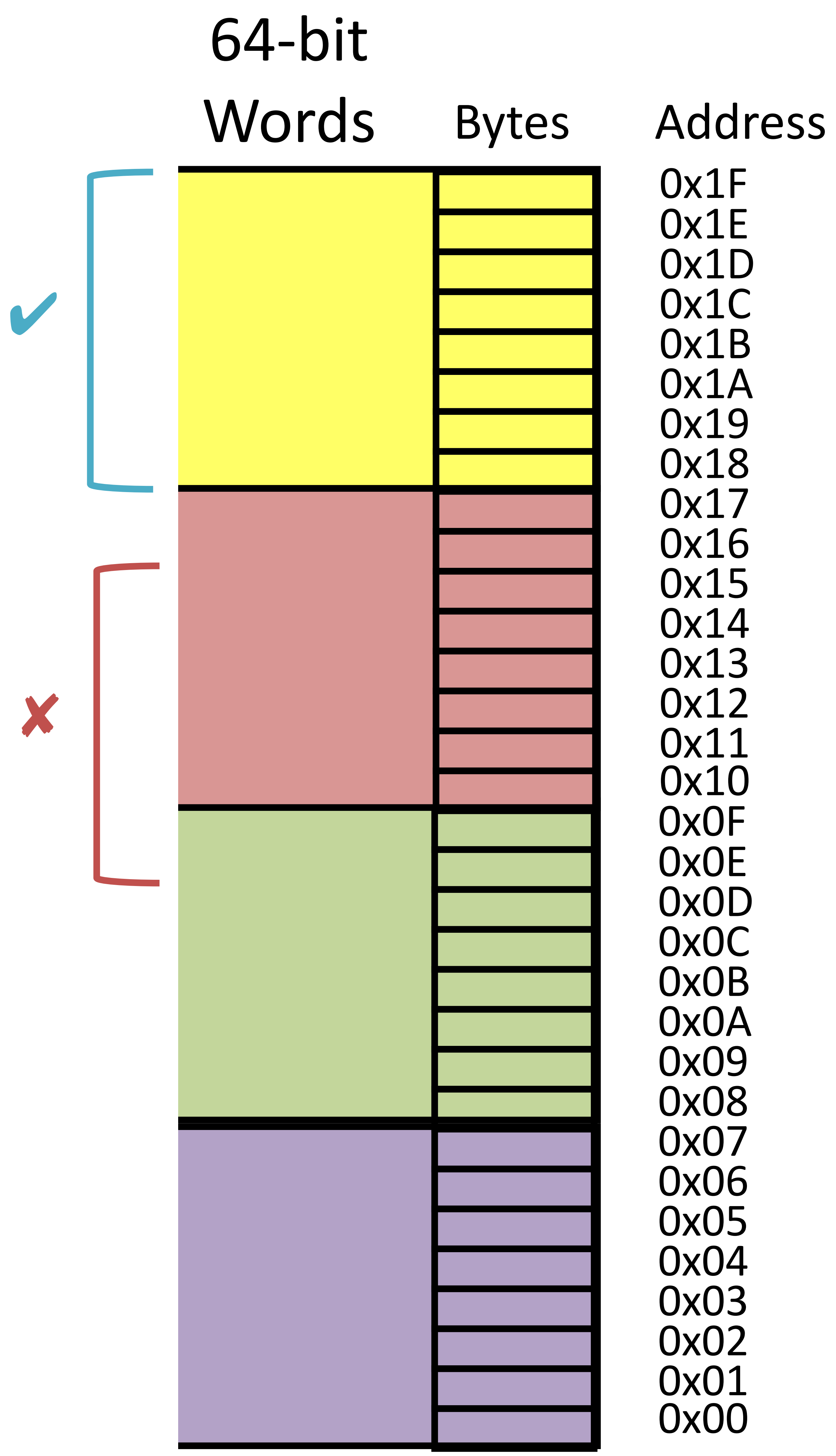
Example: 8 byte (64 bit) values

Alignment

Multi-byte values start at addresses that are multiples of their size

Bit order within byte always same.

Recall: byte ordering within larger value?

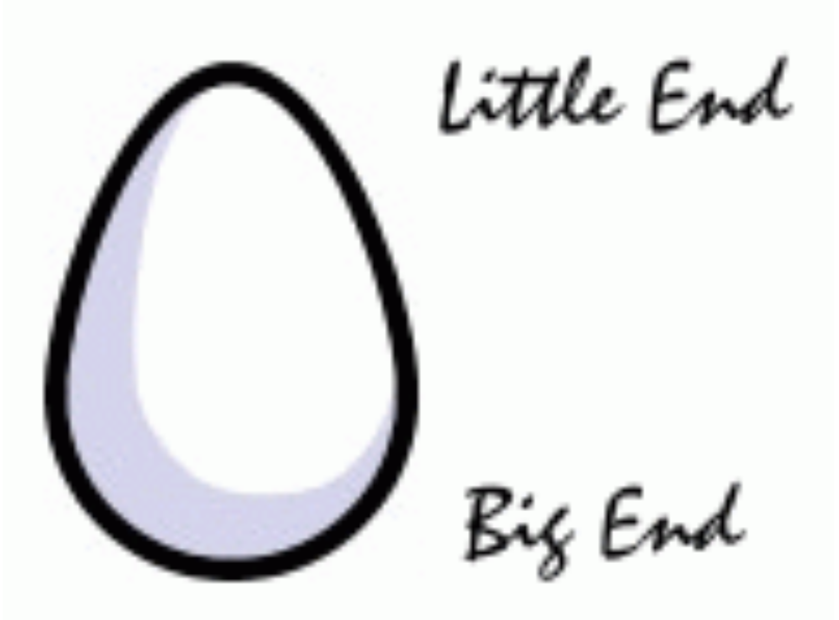


Endianness: details

In what order are the individual bytes of a multi-byte value stored in memory?

most significant byte (bits 31–24) / least significant byte (bits 7–0)

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
2A	B6	00	0B



Address	Contents
03	**2A**
02	B6
01	00
00	**0B**

Little Endian: least significant byte first

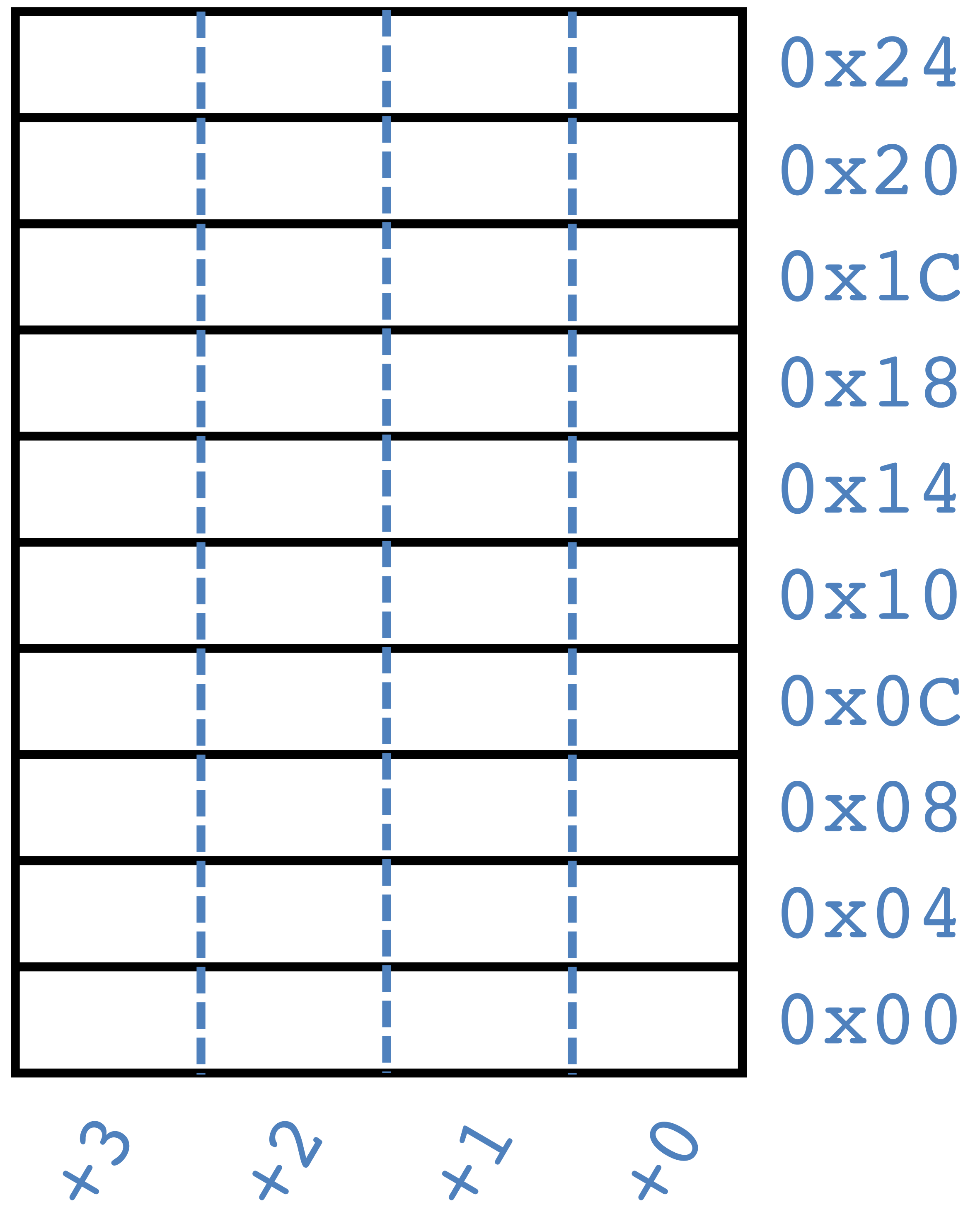
- low order byte at low address
- high order byte at high address
- used by **x86**, … and **CS240!**

Address	Contents
03	**0B**
02	00
01	B6
00	**2A**

Big Endian: most significant byte first

- high order byte at low address
- low order byte at high address
- used by networks, SPARC, …

Data, addresses, and pointers



For these slides, we'll draw the bytes in this reverse order so that multi-byte values can be read directly

memory drawn as 32-bit values, little endian order

Data, addresses, and pointers

address = index of a location in memory

pointer = a reference to a location in memory,
represented as an address stored as data

Let's store the number 240 at address `0x20`.

$240_{10} = \mathrm{F0}_{16}$ = `0x00 00 00 F0`

At address `0x08` we store a pointer to the contents at address `0x20`.

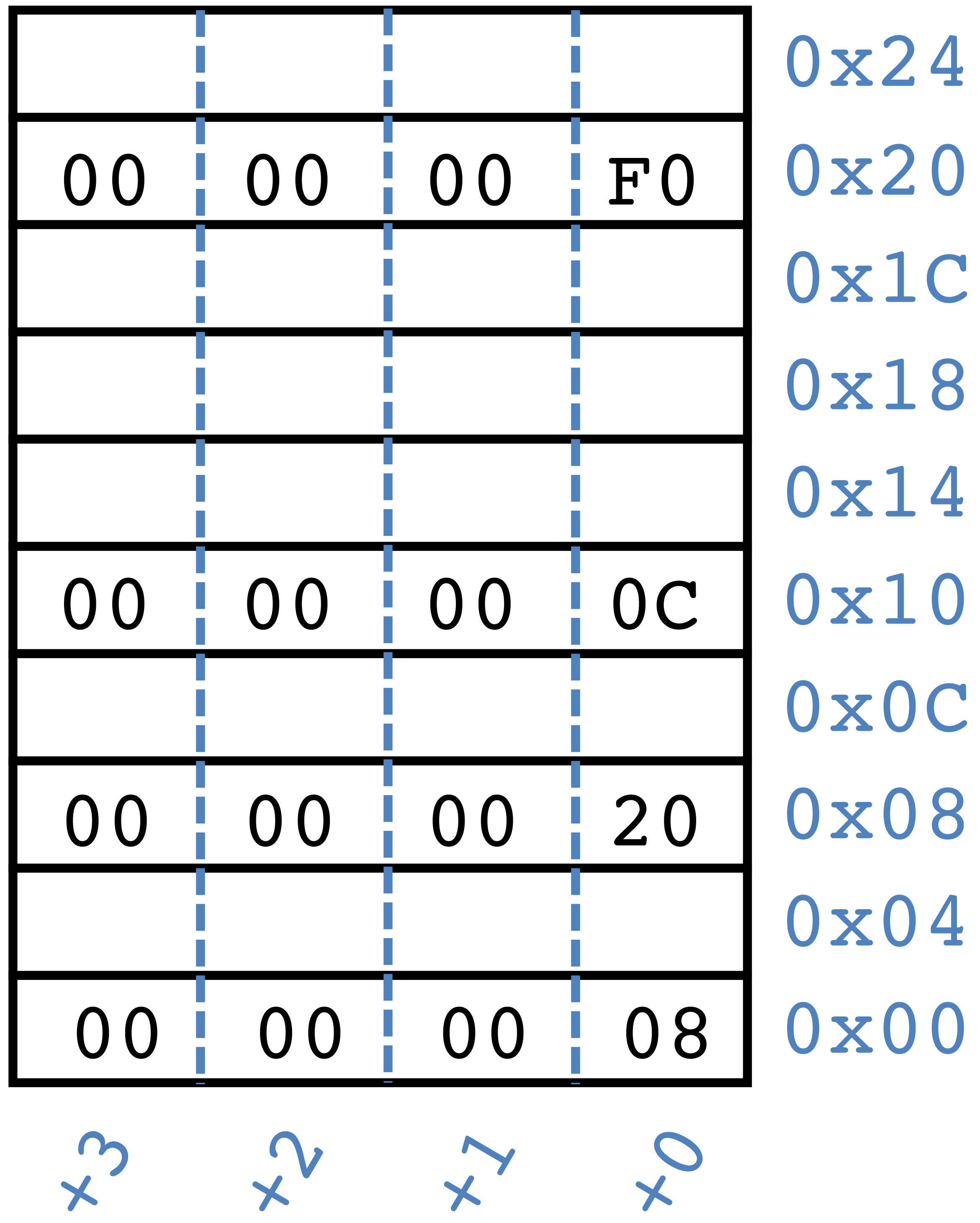
At address `0x00`, we store a pointer to a pointer.

The number 12 is stored at address `0x10`.

Is it a pointer?

How do we know if values are pointers or not?

How do we manage use of memory?



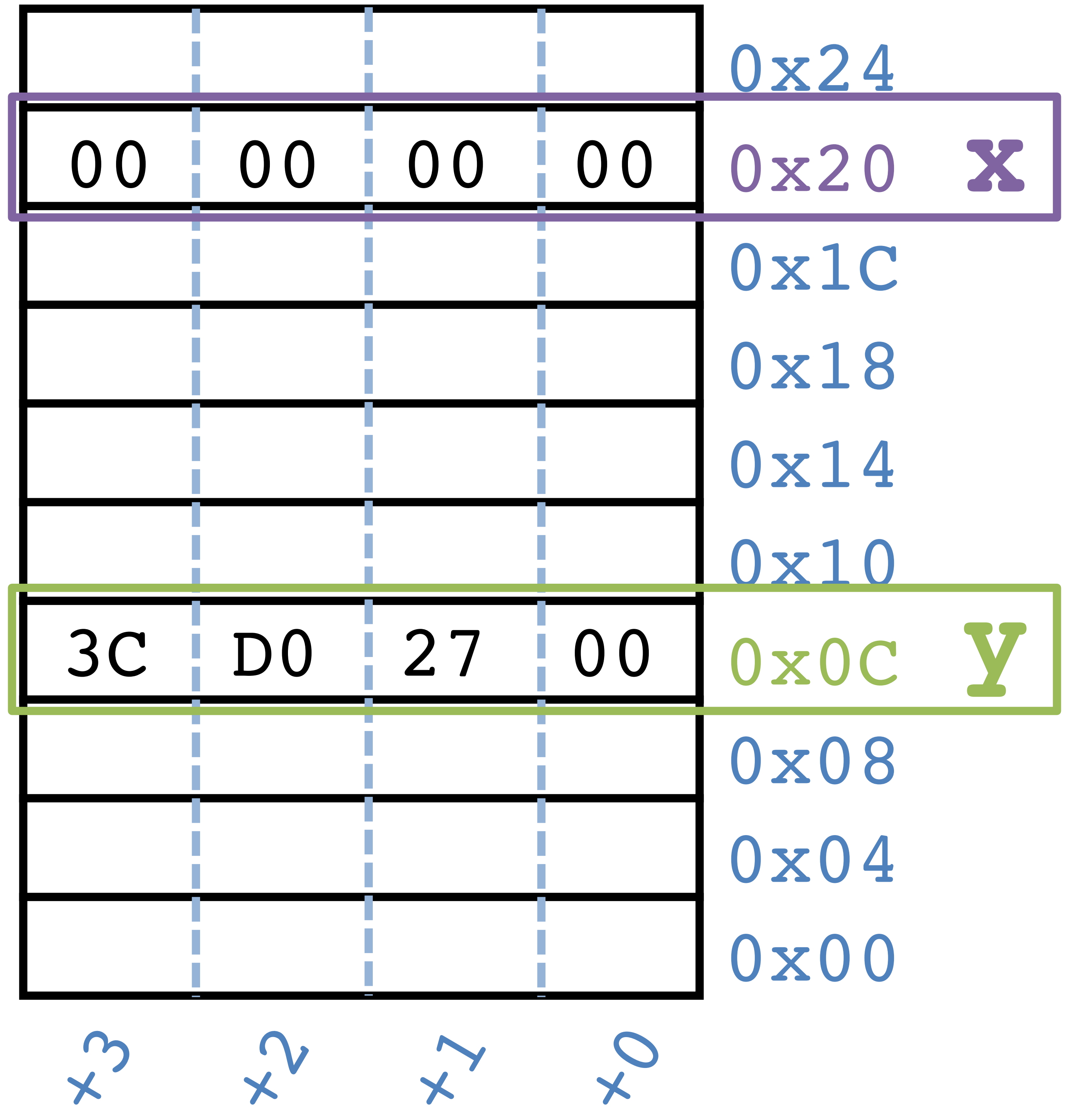
memory drawn as 32-bit values,
little endian order

C: Variables are locations

The compiler creates a map from variable name → location.

Declarations do not initialize!

```
int x; // x @ 0x20
int y; // y @ 0x0C

x = 0; // store 0 @ 0x20

// store 0x3CD02700 @ 0x0C
y = 0x3CD02700;

// 1. load the contents @ 0x0C
// 2. add 3
// 3. store sum @ 0x20
x = y + 3;
```

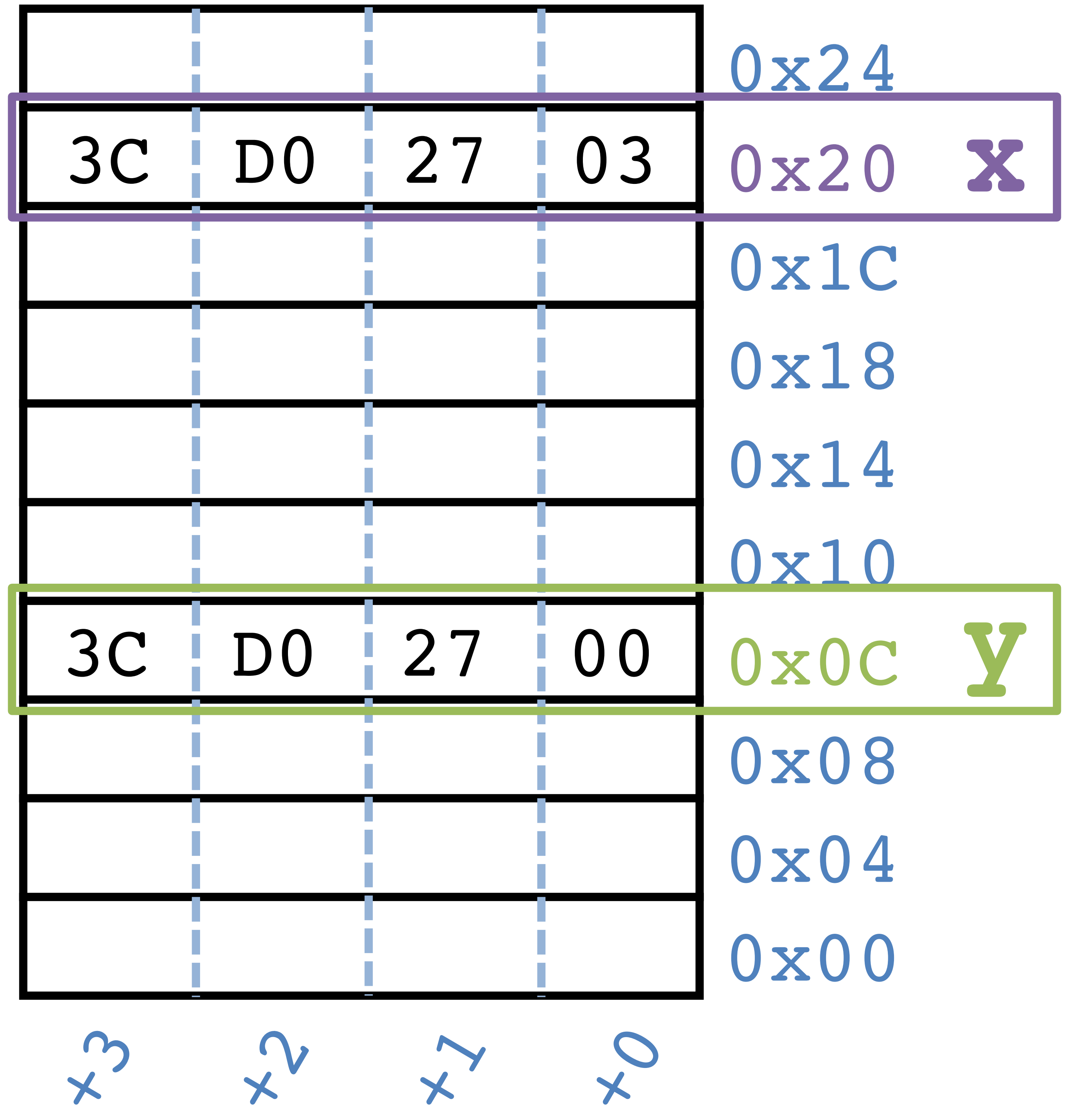


C: Variables are locations

The compiler creates a map from variable name → location.

Declarations do not initialize!

```
int x; // x @ 0x20
int y; // y @ 0x0C

x = 0; // store 0 @ 0x20

// store 0x3CD02700 @ 0x0C
y = 0x3CD02700;

// 1. load the contents @ 0x0C
// 2. add 3
// 3. store sum @ 0x20
x = y + 3;
```



C: Pointer operations and types

address = index of a location in memory

pointer = a *reference* to a location in memory, an address stored as data

Expressions using addresses and pointers:

&___ **address of** the memory location representing ___

a.k.a. "reference to ___"

*___ **contents at** the memory address given by ___

a.k.a. "dereference ___"

Pointer types:

___* address of a memory location holding a ___

a.k.a. "a reference to a ___"

C: Types determine sizes

Used by CS Linux, most modern machines

Sizes of data types (in bytes)

Java Data Type	C Data Type	32-bit word	**64-bit word**
boolean	*bool*	1	1
byte	char	1	1
char		2	2
short	short int	2	2
int	int	4	4
float	float	4	4
	long int	4	8
double	double	8	8
long	long long	8	8
	long double	8	16
(reference)	(pointer) *	4	8

address size = word size

C: Pointer example

& = address of

** = contents at*

```c
int* p;
```

Declare a variable, `p`

that will hold the address of a memory location holding an int

```c
int x = 5;
int y = 2;
```

Declare two variables, `x` and `y`, that hold ints, and store 5 and 2 in them, respectively.

```c
p = &x;
```

Take the address of the memory representing `x`

... and store it in the memory location representing `p`.
Now, "*p points to x.*"

```c
y = 1 + *p;
```

Add 1 to the contents of memory at the address given by the contents of the memory location representing `p`

... and store it in the memory location representing y.

C: Pointer example

& = address of

** = contents at*

location

C assignment:

Left-hand-side* = *right-hand-side;

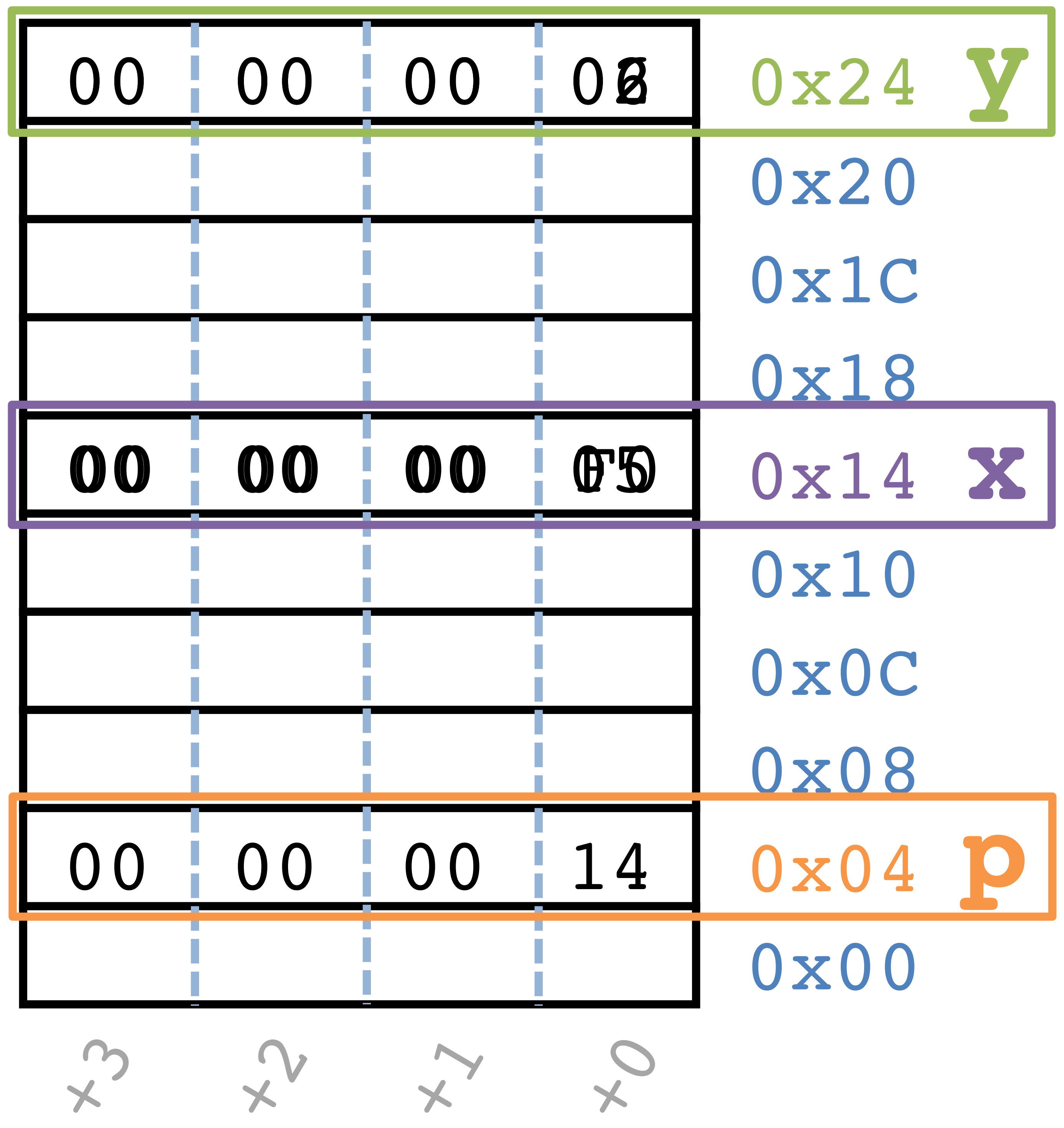
value

```
int* p;     // p @ 0x04
int x = 5;  // x @ 0x14, store 5 @ 0x14
int y = 2;  // y @ 0x24, store 2 @ 0x24
p = &x;     // store 0x14 @ 0x04

// 1. load the contents @ 0x04  (=0x14)
// 2. load the contents @ 0x14  (=0x5)
// 3. add 1
// 4. store sum as contents @ 0x24
y = 1 + *p;

// 1. load the contents @ 0x04  (=0x14)
// 2. store 0xF0 as contents @ 0x14
*p = 240;
```

What is the type of *p?

What is the type of &x?

What is `*(&y)` ?



C: Pointer type syntax

Spaces between base type, *, and variable name mostly do not matter.

The following are **equivalent**:

```
int* ptr;
```

I see: "The variable **ptr** holds an **address of an int** in memory."

```
int * ptr;
```

```
int *ptr;
```

more common C style

Looks like: "Dereferencing the variable **ptr** will yield an **int**."

Or "The **memory location** where the variable **ptr** points holds an **int**."

Caveat: do not declare multiple variables unless using the last form.

`int* a, b;` means `int *a, b;` means `int* a; int b;`

C: Arrays

Arrays are adjacent memory locations storing the same type of data.

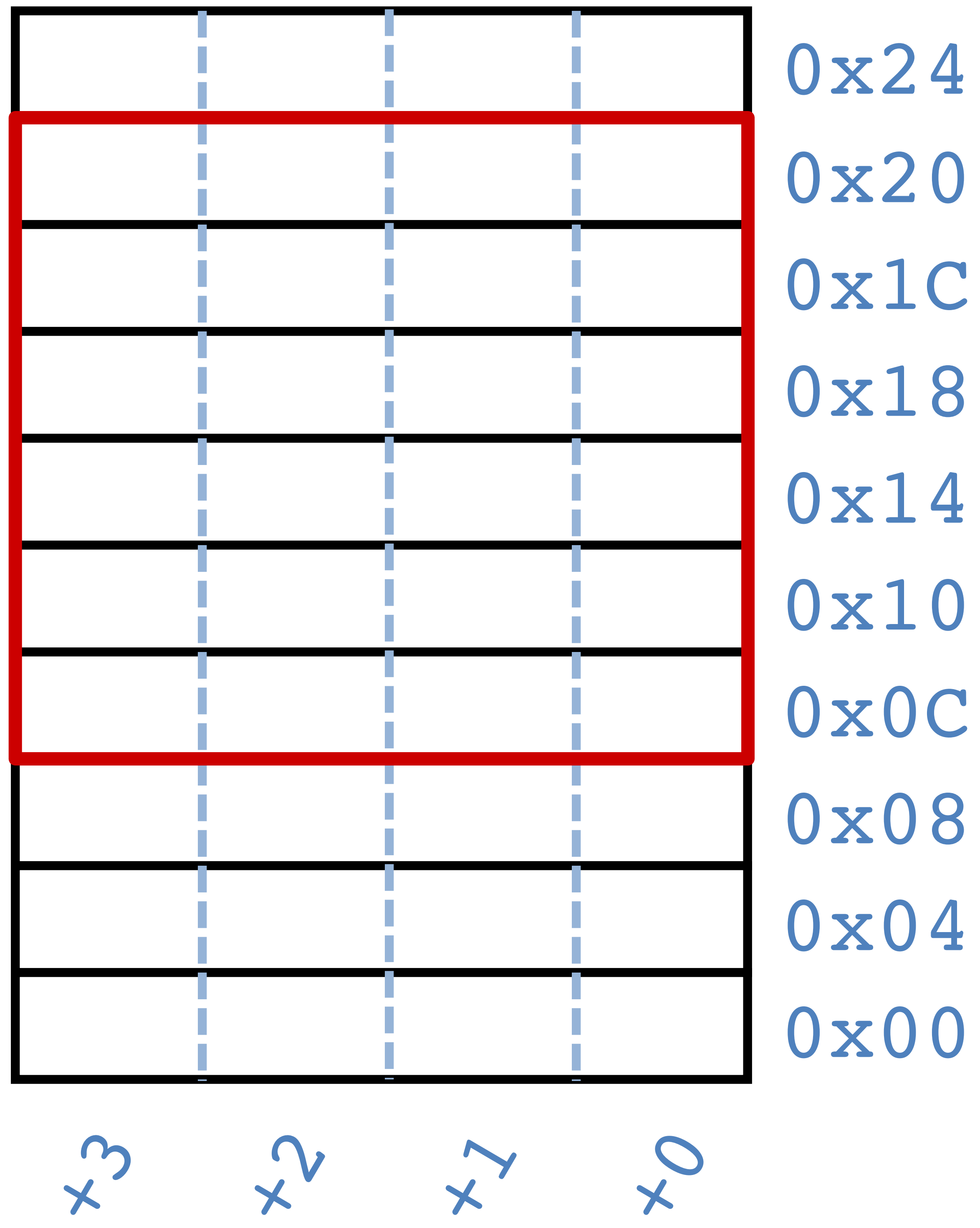
a is a name for the array's base address, can be used as an *immutable* pointer.

Declaration: `int a[6];`

element type

name

number of elements



C: Arrays

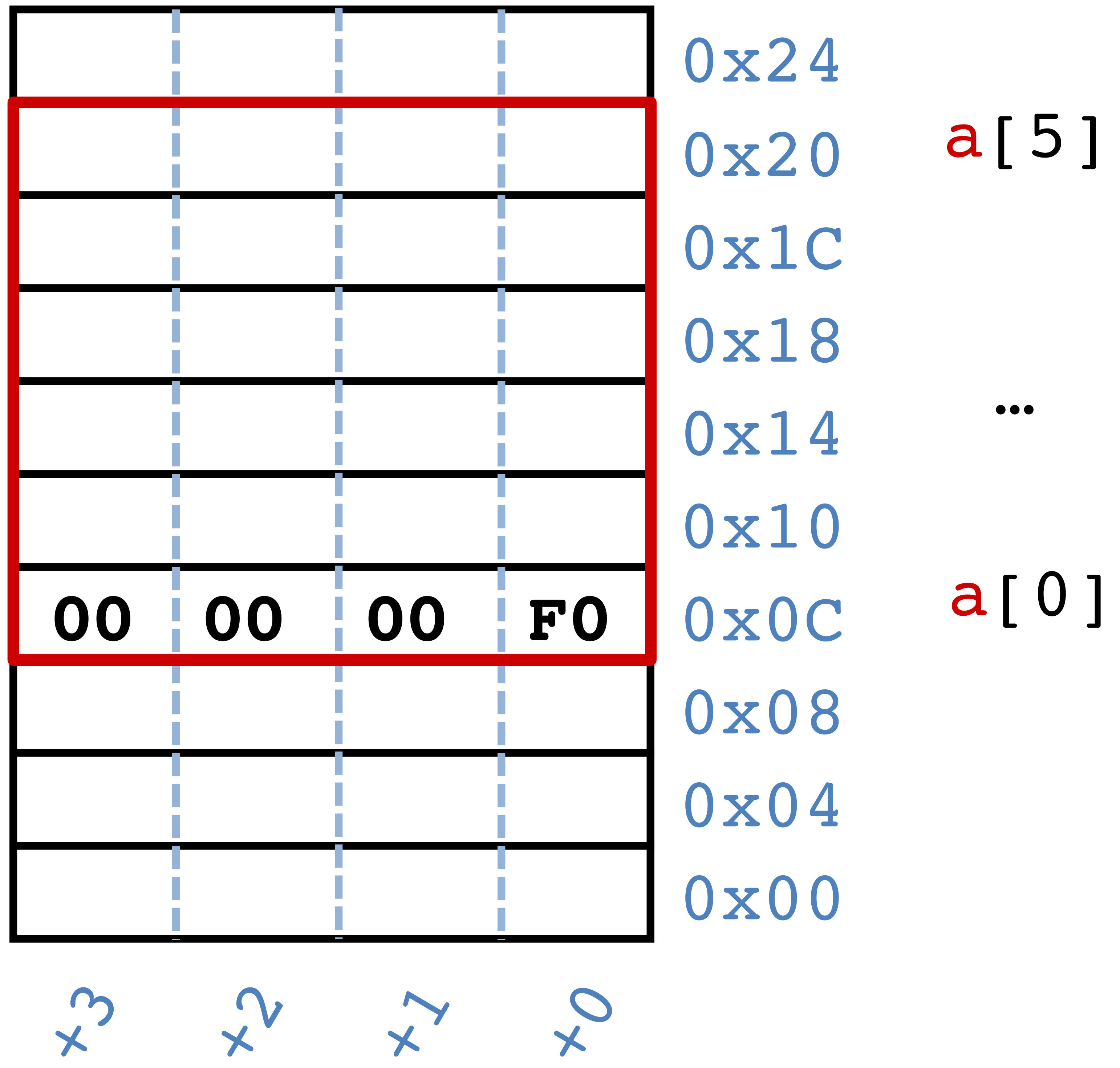
Declaration: `int a[6];`

Indexing: `a[0] = 0xf0;`

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.



C: Arrays

Arrays are adjacent memory locations storing the same type of data.

a is a name for the array's base address, can be used as an *immutable* pointer.

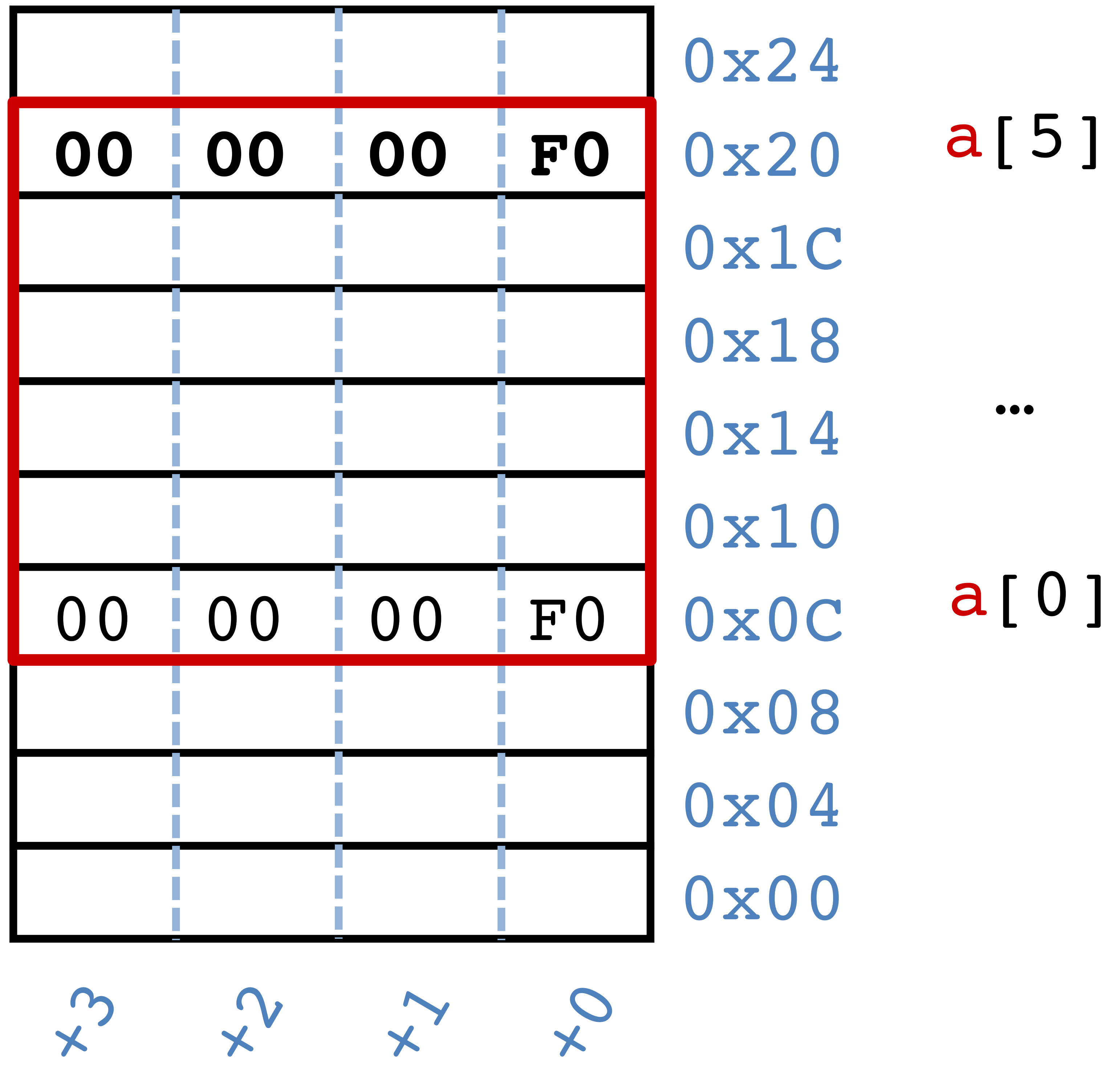
Address of `a[i]` is base address `a` plus `i` times element size in bytes.

Declaration:

```c
int a[6];
```

Indexing:

```c
a[0] = 0xf0;
a[5] = a[0];
```



C: Arrays

Declaration: `int a[6];`

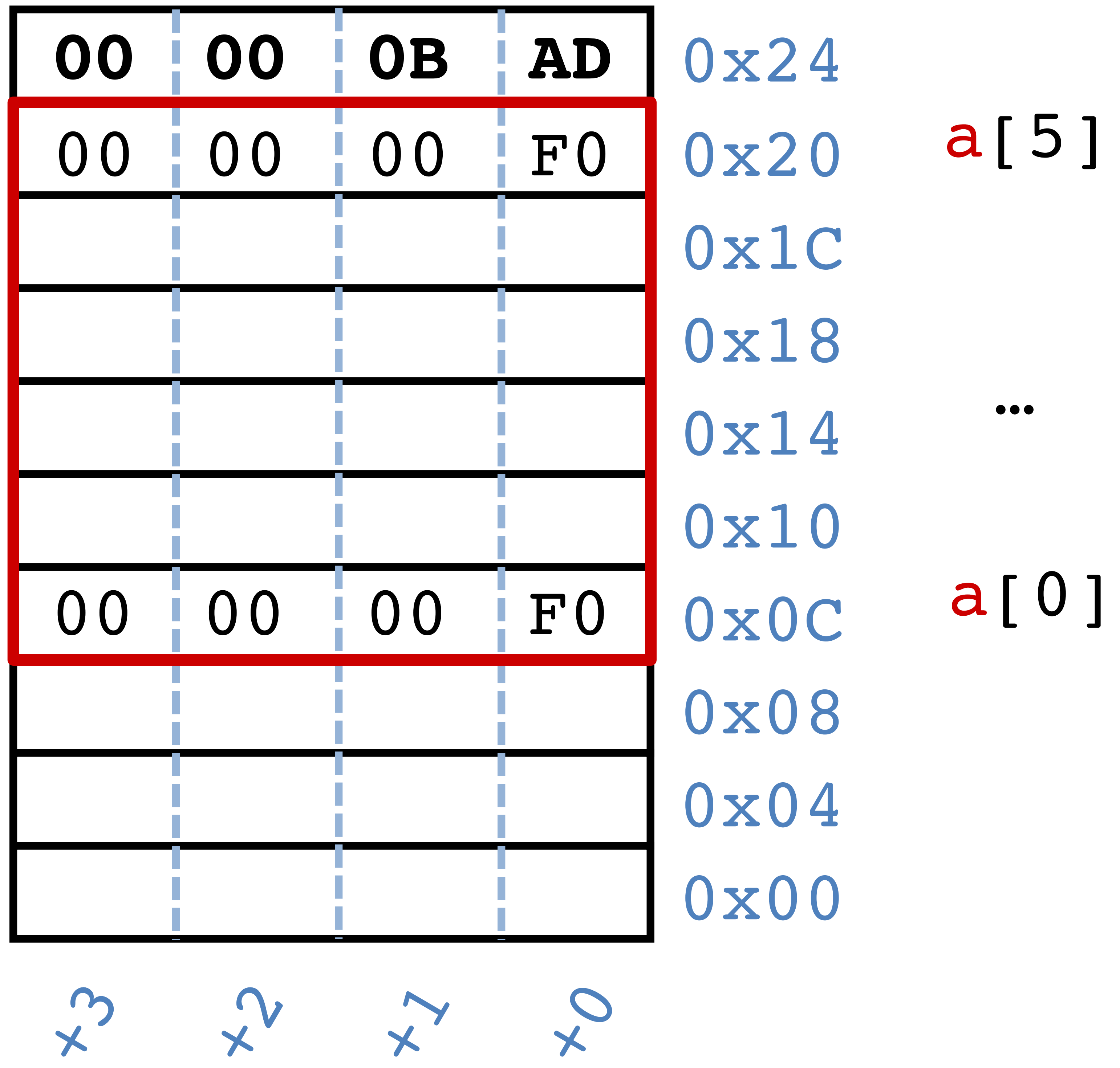
Indexing:
```
a[0] = 0xf0;
a[5] = a[0];
```

No bounds check: `a[6] = 0xBAD;`

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.



C: Arrays

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.

Declaration:

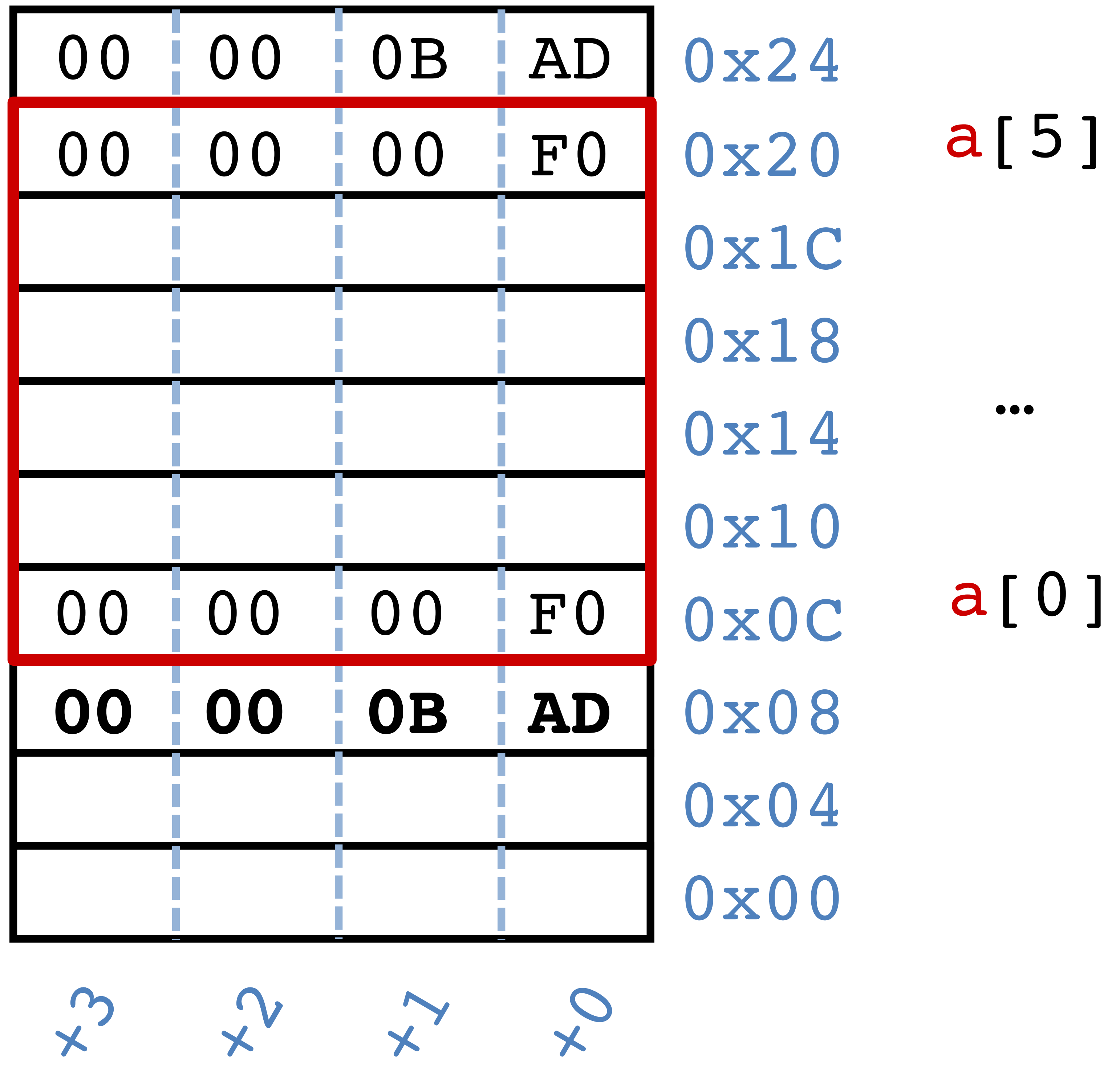
```
int a[6];
```

Indexing:

```
a[0] = 0xf0;
a[5] = a[0];
```

No bounds check:

```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```



C: Arrays

Arrays are adjacent memory locations storing the same type of data.

a is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address a plus `i` times element size in bytes.

Declaration:

```
int a[6];
```

Indexing:

```
a[0] = 0xf0;
a[5] = a[0];
```

No bounds check:

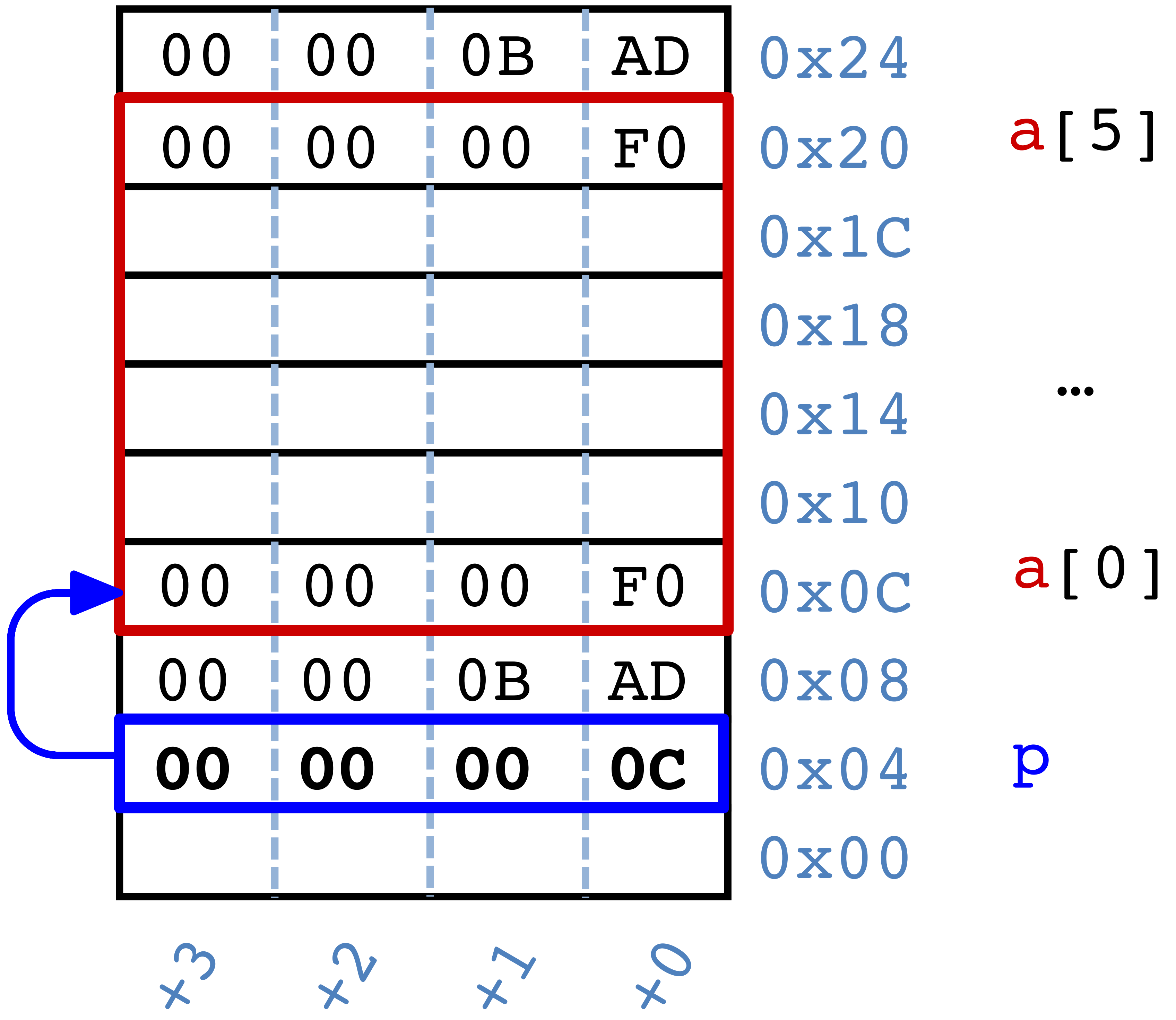
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:

```
int* p;
```

equivalent

```
p = a;
p = &a[0];
```



C: Arrays

Arrays are adjacent memory locations storing the same type of data.

a is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.

Declaration:

```
int a[6];
```

Indexing:

```
a[0] = 0xf0;
a[5] = a[0];
```

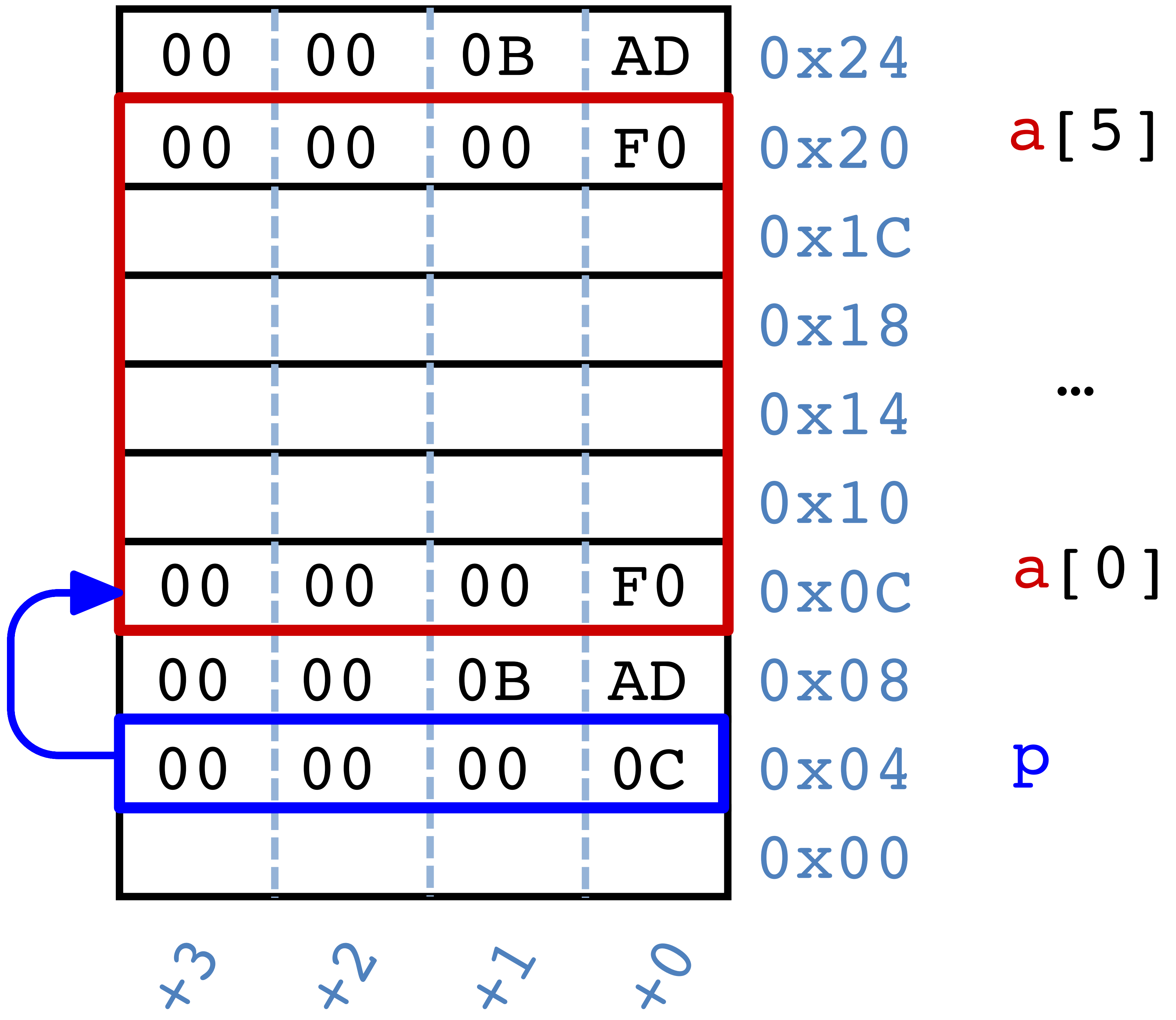
No bounds check:

```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:

```
int* p;
p = a;
p = &a[0];
*p = 0xA;
```

equivalent



C: Arrays

Arrays are adjacent memory locations storing the same type of data.

a is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.

Declaration:

```
int a[6];
```

Indexing:

```
a[0] = 0xf0;
a[5] = a[0];
```

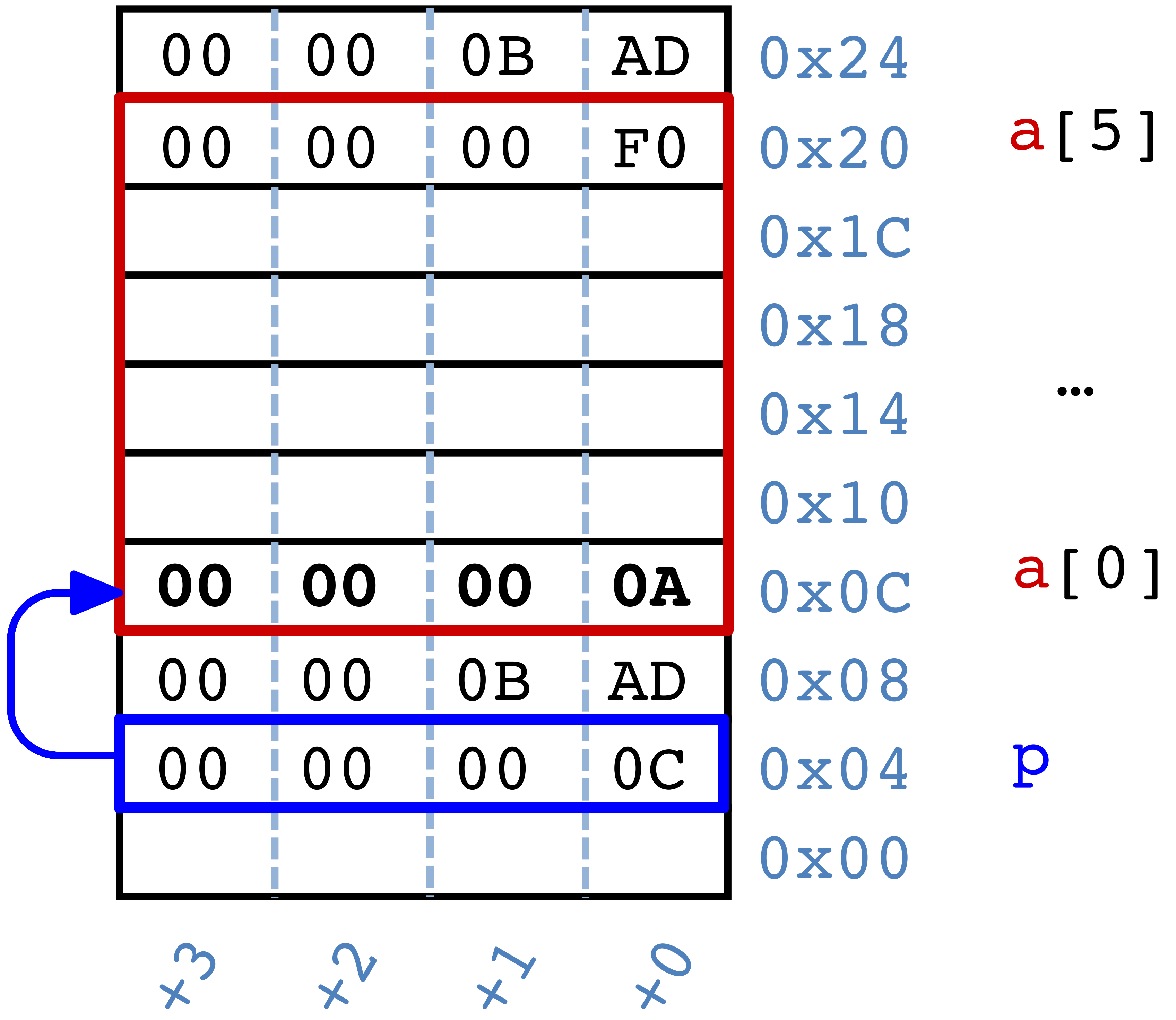
No bounds check:

```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:

```
int* p;
p = a;
p = &a[0];
*p = 0xA;
```

equivalent: `p = a;` / `p = &a[0];`



C: Arrays

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.

Declaration:

```
int a[6];
```

Indexing:

```
a[0] = 0xf0;
a[5] = a[0];
```

No bounds check:

```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:

```
int* p;
```

equivalent

```
p = a;
p = &a[0];
```

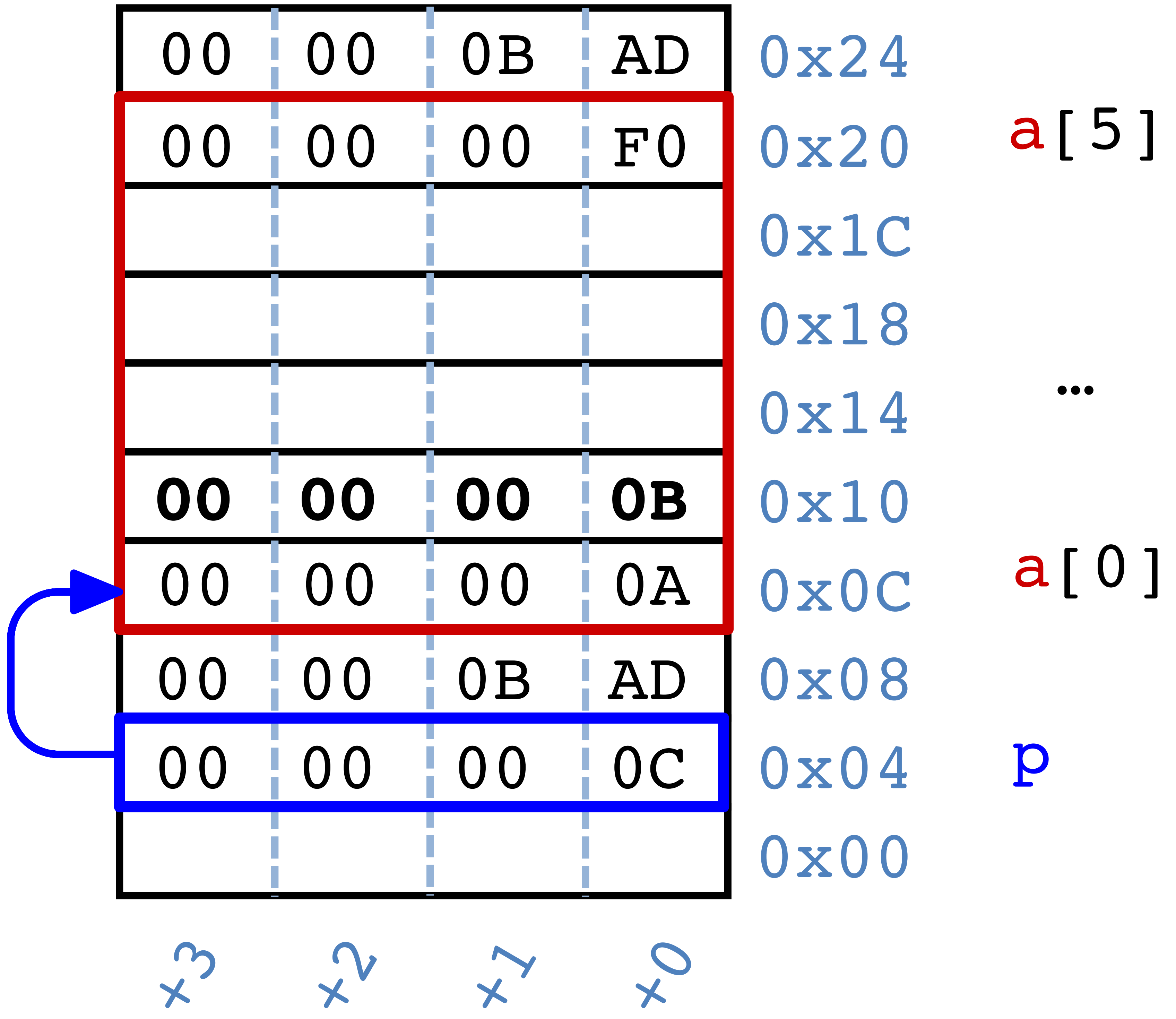
```
*p = 0xA;
```

equivalent

```
p[1] = 0xB;
*(p + 1) = 0xB;
```

array indexing = address arithmetic

Both are scaled by the size of the type.



C: Arrays

Arrays are adjacent memory locations storing the same type of data.

a is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.

Declaration:
```
int a[6];
```

Indexing:
```
a[0] = 0xf0;
a[5] = a[0];
```

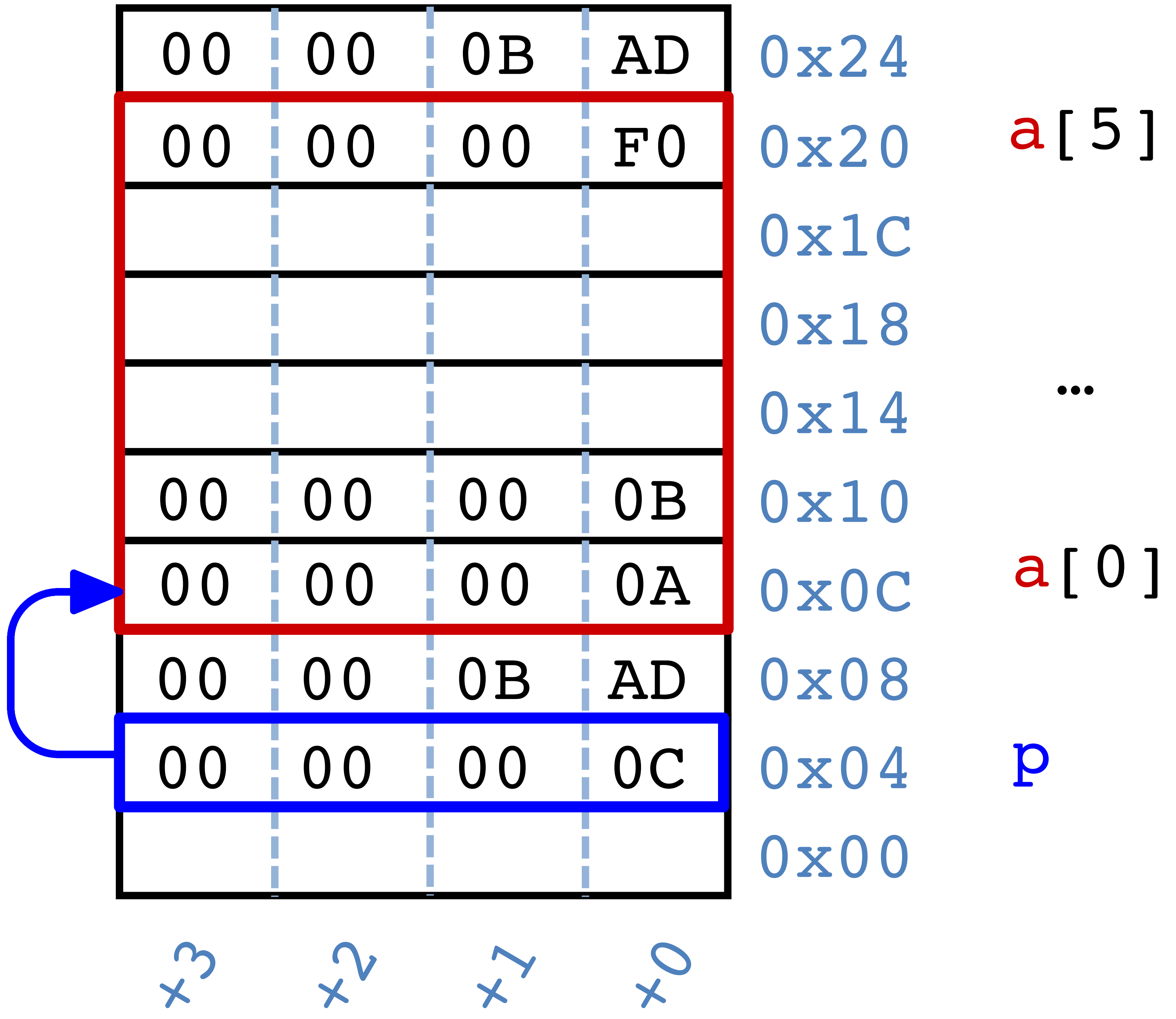
No bounds check:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:
```
int* p;
p = a;
p = &a[0];
*p = 0xA;
```
equivalent: `p = a;` / `p = &a[0];`

```
p[1] = 0xB;
*(p + 1) = 0xB;
p = p + 2;
```
equivalent: `p[1] = 0xB;` / `*(p + 1) = 0xB;`

array indexing = address arithmetic

Both are scaled by the size of the type.



C: Arrays

Arrays are adjacent memory locations storing the same type of data.

`a` is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.

Declaration:
```
int a[6];
```

Indexing:
```
a[0] = 0xf0;
a[5] = a[0];
```

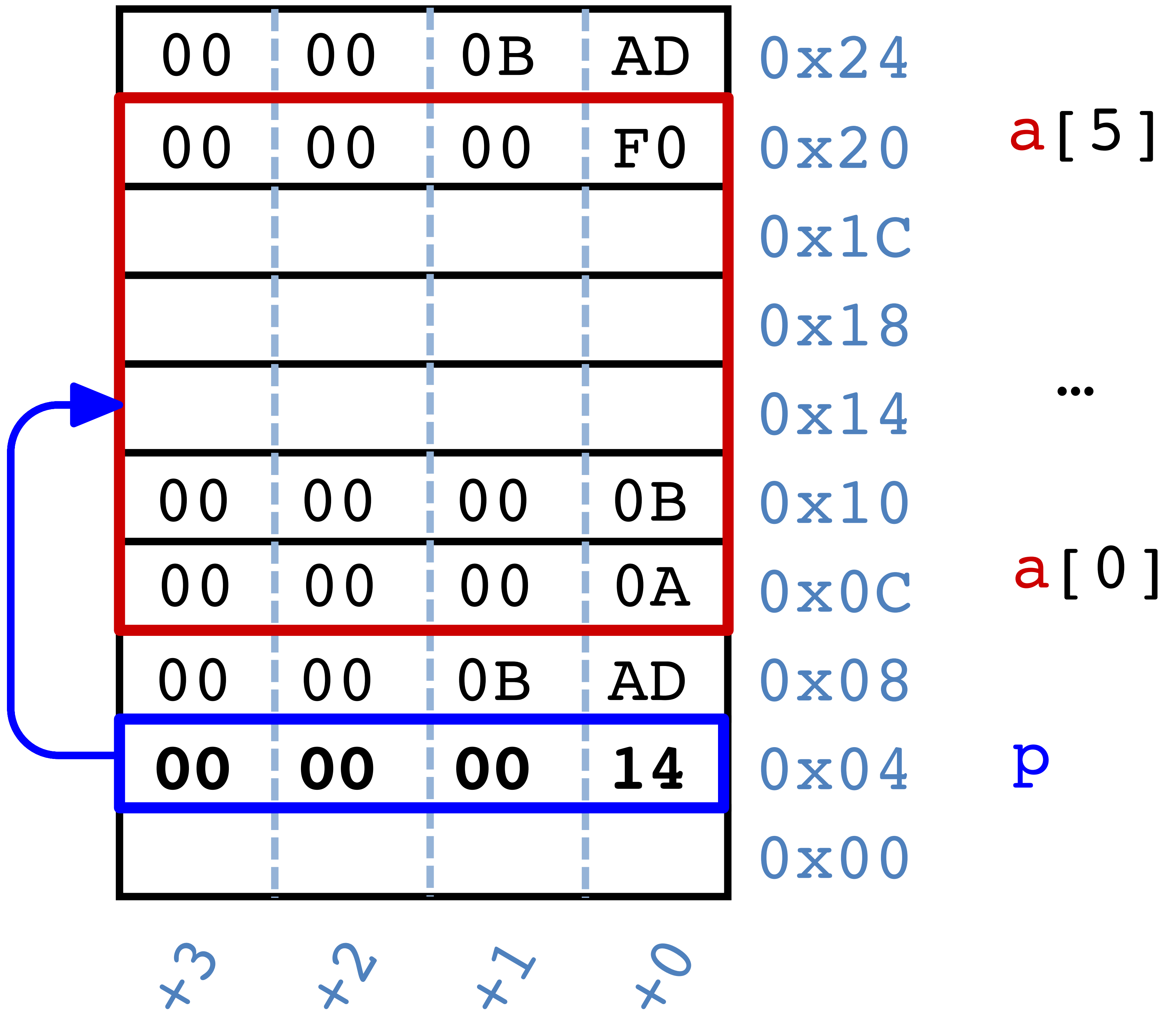
No bounds check:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:
```
int* p;
```
equivalent {
```
p = a;
p = &a[0];
```
```
*p = 0xA;
```

equivalent {
```
p[1] = 0xB;
*(p + 1) = 0xB;
```
```
p = p + 2;
```

array indexing = address arithmetic

Both are scaled by the size of the type.



C: Arrays

Arrays are adjacent memory locations storing the same type of data.

a is a name for the array's base address, can be used as an *immutable* pointer.

Address of `a[i]` is base address `a` plus `i` times element size in bytes.

Declaration:
```
int a[6];
```

Indexing:
```
a[0] = 0xf0;
a[5] = a[0];
```

No bounds check:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

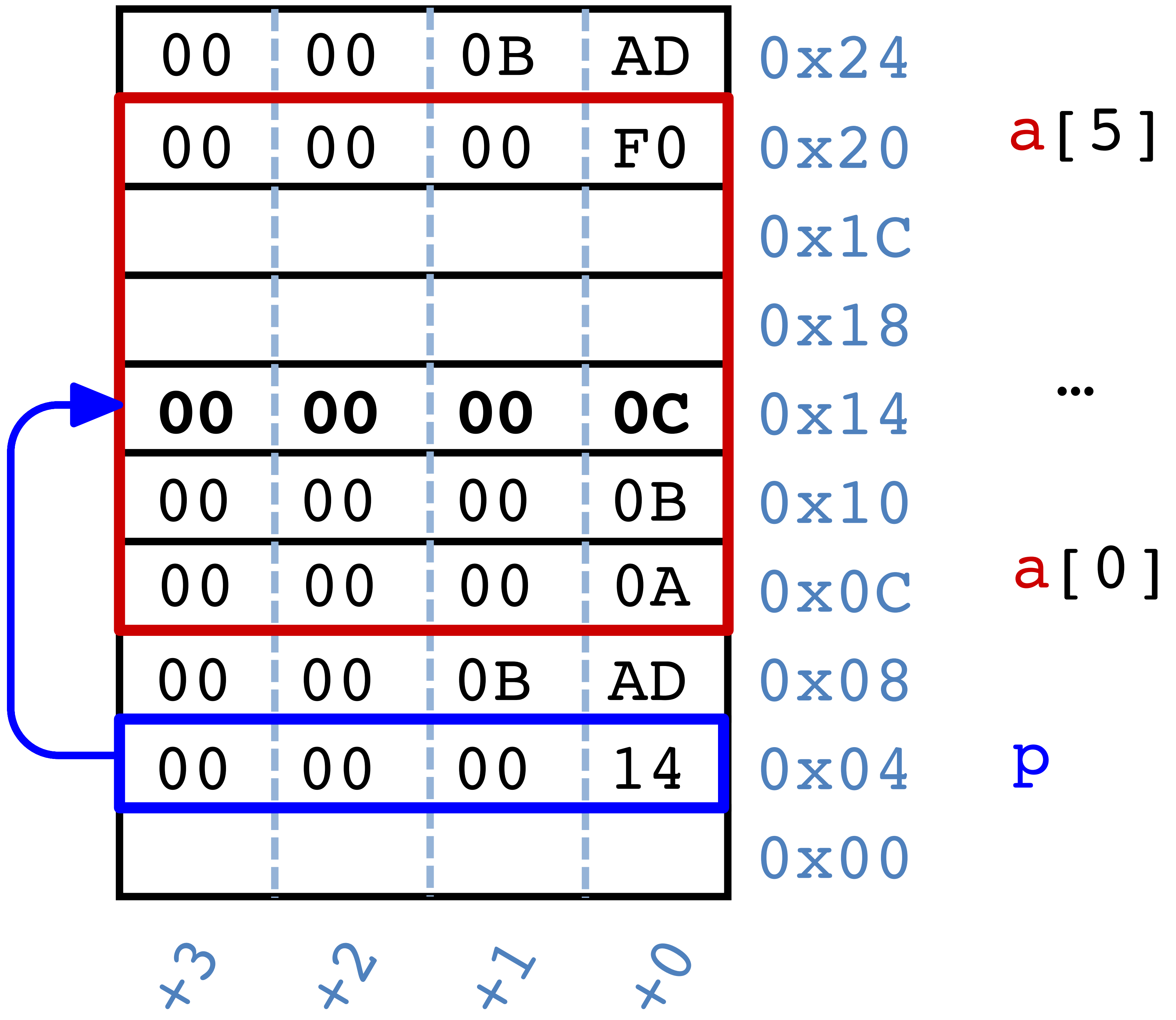
Pointers:
```
int* p;
```
equivalent
```
p = a;
p = &a[0];
```
```
*p = 0xA;
```

equivalent
```
p[1] = 0xB;
*(p + 1) = 0xB;
```
```
p = p + 2;
```

array indexing = address arithmetic

Both are scaled by the size of the type.

```
*p = a[1] + 1;
```



C: Array allocation

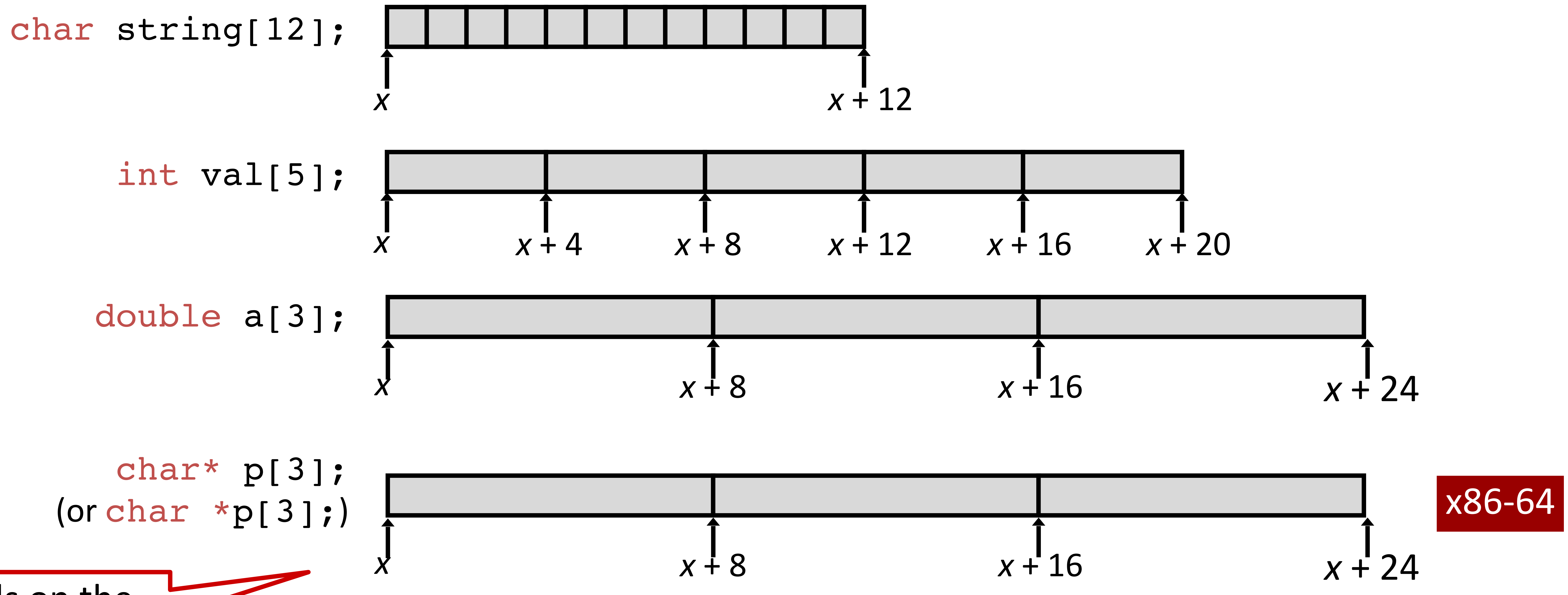
Basic Principle

`T A[N];`

Array of length *N* with elements of type *T* and name *A*

Contiguous block of *N**`sizeof(T)` bytes of memory

Use `sizeof` to determine proper size in C.



size depends on the machine word size

C: Array access

ex

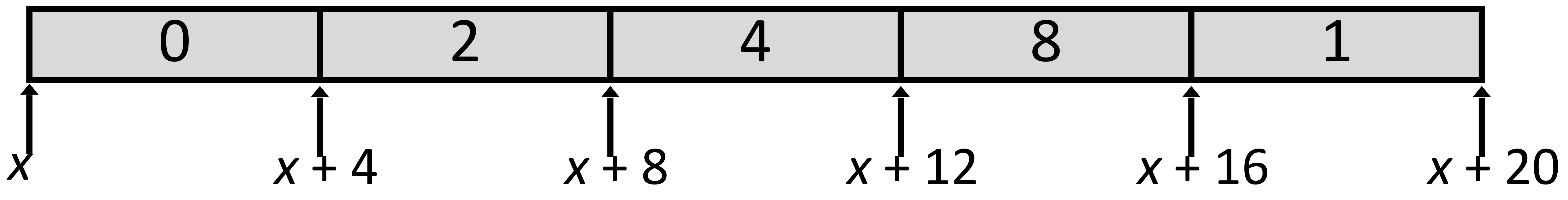
Basic Principle

`T A[N];`

Array of length *N* with elements of type *T* and name *A*

Identifier *A* has type *T**

`int val[5];`



Expression	Type	Value
`val[4]`	`int`	1
`val`	`int *`	
`val+1`	`int *`	
`&val[2]`	`int *`	
`val[5]`	`int`	
`*(val+1)`	`int`	
`val + i`	`int *`	

Representing strings

A C-style string is represented by an array of bytes (`char`).

- Elements are one-byte ASCII codes for each character.
- ASCII = American Standard Code for Information Interchange

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

C: Null-terminated strings

ex

C strings: arrays of ASCII characters ending with *null character*.

Why?

0x57	0x65	0x6C	0x6C	0x65	0x73	0x6C	0x65	0x79	0x20	0x43	0x53	0x00
'W'	'e'	'l'	'l'	'e'	's'	'l'	'e'	'y'	' '	'C'	'S'	'\0'

Does Endianness matter for strings?

```
int string_length(char str[]) {



}
```

C: * *and* [ ]

ex

C programmers often use * where you might expect []:

e.g., char*:

- pointer to a char
- pointer to the first char in a string of unknown length

```
int strcmp(char* a, char* b);
```

C: 0 vs. `'\0'` vs. NULL

`0`

Name:	zero
Type:	`int`
Size:	4 bytes
Value:	`0x00000000`
Usage:	The integer zero.

`'\0'`

Name:	null character
Type:	`char`
Size:	1 byte
Value:	`0x00`
Usage:	Terminator for C strings.

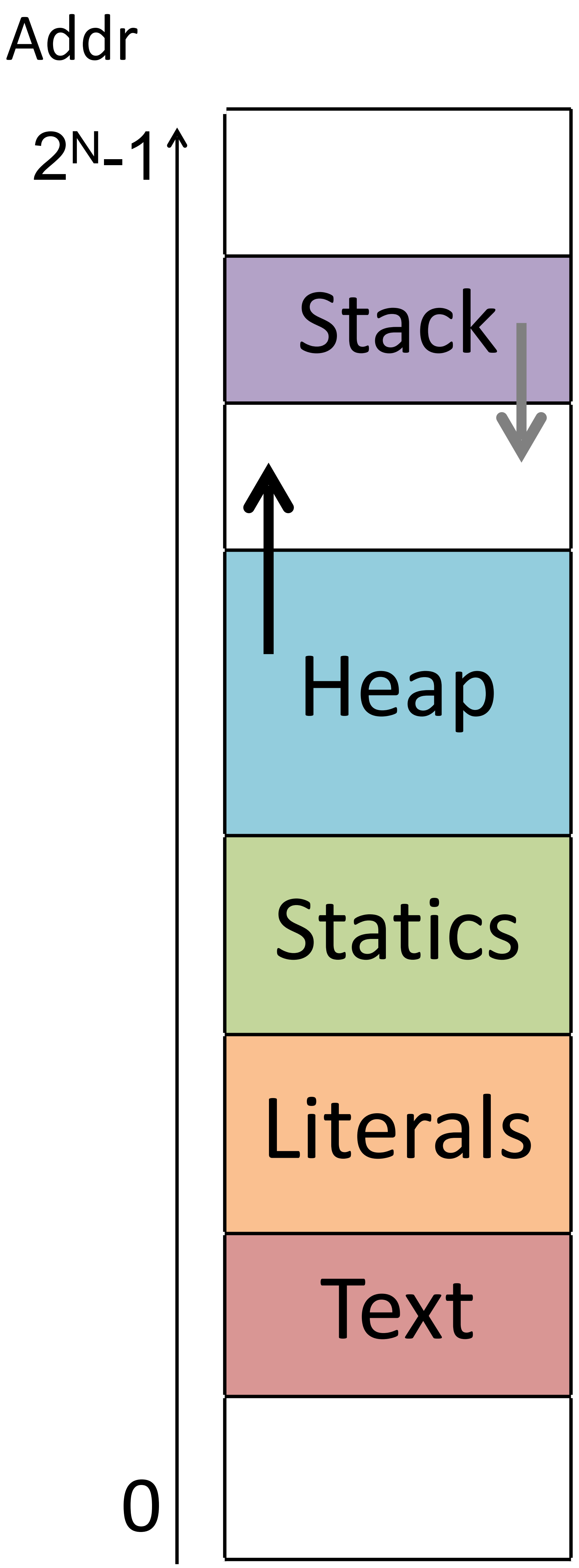
`NULL`

Name:	null pointer / null reference / null address
Type:	`void*`
Size:	1 word (= 8 bytes on a 64-bit architecture)
Value:	`0x0000000000000000`
Usage:	The absence of a pointer where one is expected.
	Address 0 is inaccessible, so `*NULL` is invalid; it crashes.

Is it important/necessary to encode the null character or the null pointer as 0x0?

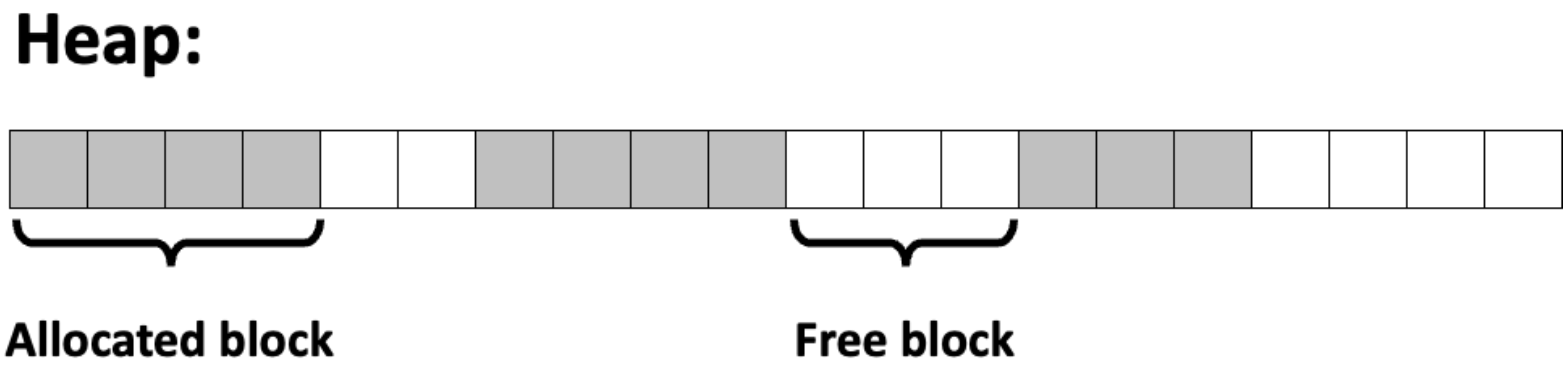
What happens if a programmer mixes up these "zeroey" values?

Memory address-space layout



Perm	Contents	Managed by	Initialized
RW	Procedure context	Compiler	Run time
RW	Dynamic data structures	Programmer, malloc/free, new/GC	Run time
RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
R	String literals	Compiler/ Assembler/Linker	Startup
X	Instructions	Compiler/ Assembler/Linker	Startup

C: Dynamic memory allocation in the heap



Managed by memory allocator:

pointer to newly allocated block
of at least that size

number of contiguous bytes required

```
void* malloc(size_t size);

void free(void* ptr);
```

pointer to allocated block to free

C: standard memory allocator

```
#include <stdlib.h>      // include C standard library
```

```
void* malloc(size_t size)
```

Allocates a memory block of at least `size` bytes and returns its address.

If memory error (e.g., allocator has no space left), returns `NULL`.

Rules:

Check for error result.

Cast result to relevant pointer type.

Use sizeof(...) to determine size.

```
void free(void* ptr)
```

Deallocates the block referenced by `ptr`,
making its space available for new allocations.

`ptr` **must** be a `malloc` result that has not yet been `freed`.

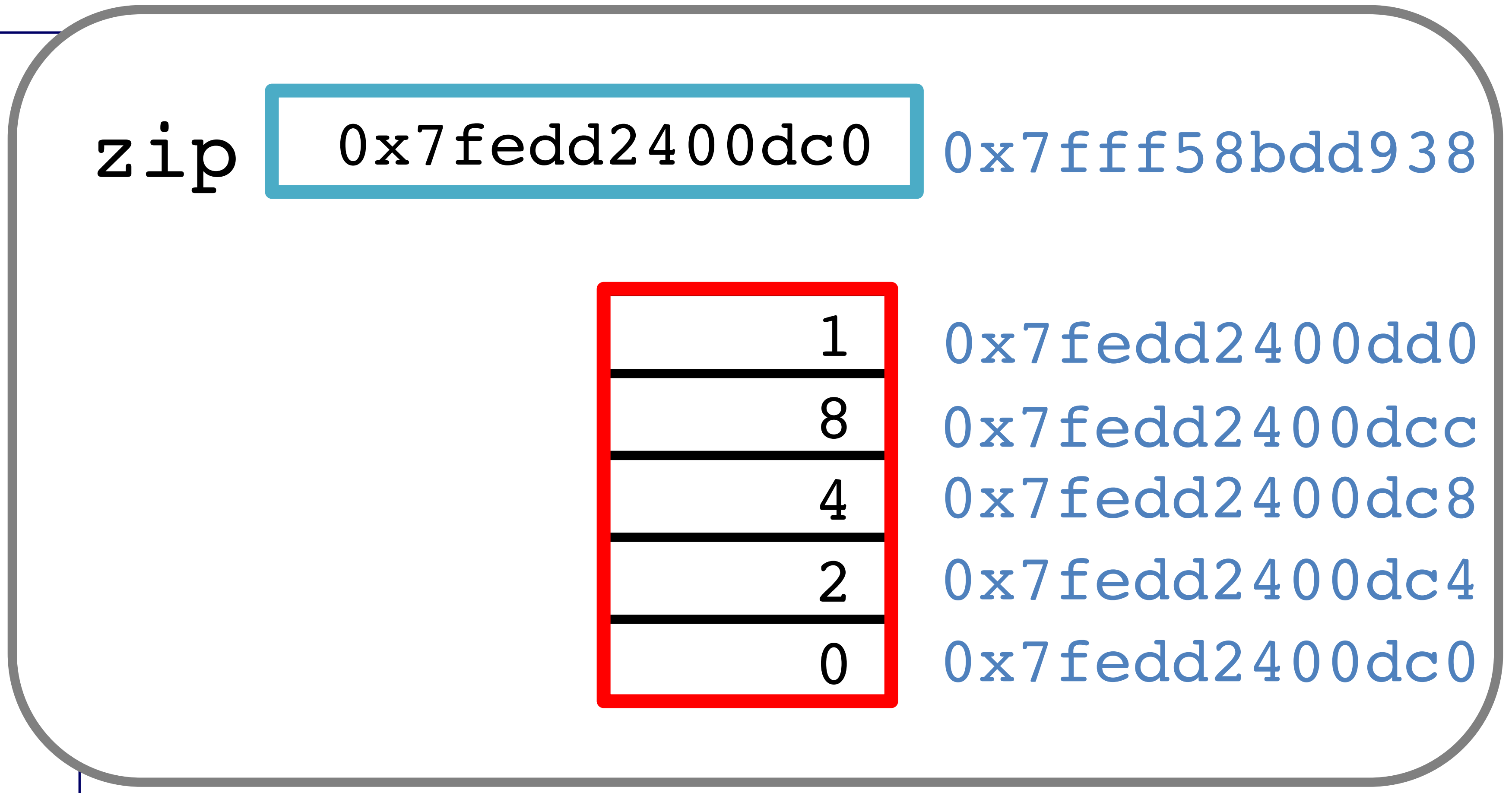
Rules:

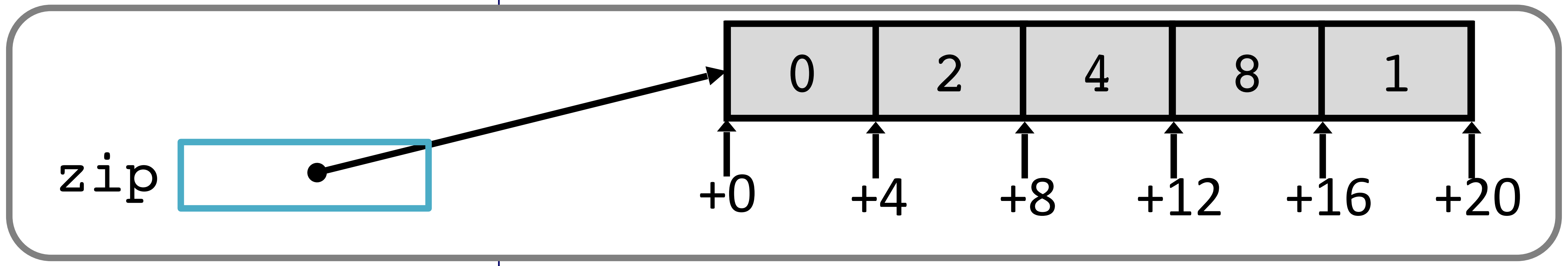
`ptr` must be a `malloc` result that has not yet been freed.

Do not use `*ptr` after freeing.

C: Dynamic array allocation

```
#define ZIP_LENGTH 5
int* zip = (int*)malloc(sizeof(int)*ZIP_LENGTH);
if (zip == NULL) {    // if error occurred
  perror("malloc");   // print error message
  exit(0);            // end the program
}

zip[0] = 0;
zip[1] = 2;
zip[2] = 4;
zip[3] = 8;
zip[4] = 1;

printf("zip is");
for (int i = 0; i < ZIP_LENGTH; i++) {
   printf(" %d", zip[i]);
}
printf("\n");

free(zip);
```



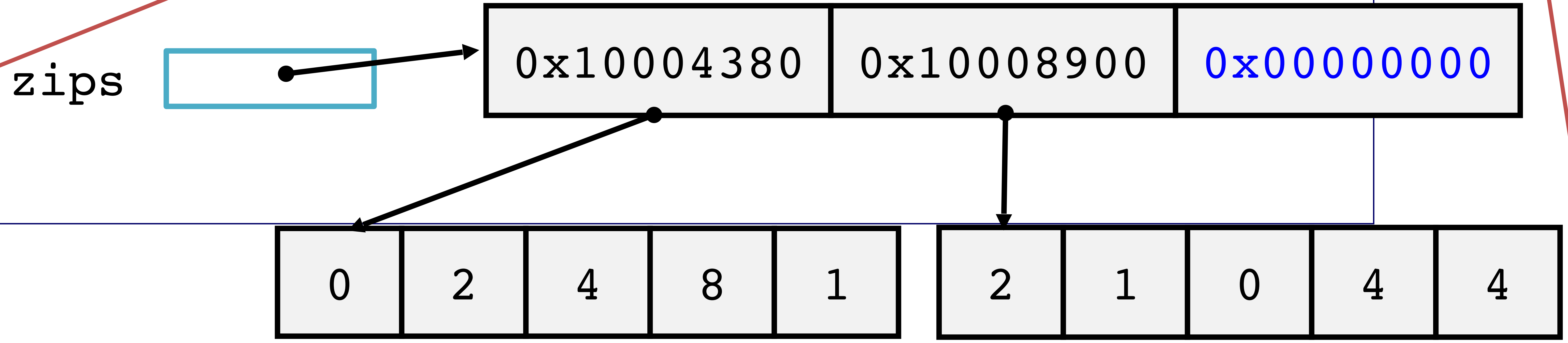


C: Array of pointers to arrays of `int`s

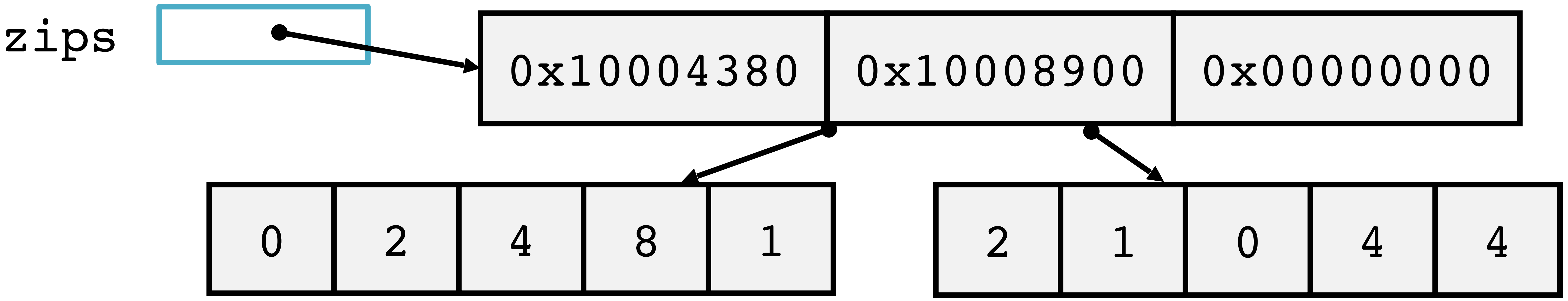
```c
int** zips = (int**)malloc(sizeof(int*) * 3);

zips[0] = (int*)malloc(sizeof(int)*5);
int* zip0 = zips[0];
zip0[0] = 0;
zips[0][1] = 2;
zips[0][2] = 4;
zips[0][3] = 8;
zips[0][4] = 1;

zips[1] = (int*)malloc(sizeof(int)*5);
zips[1][0] = 2;
zips[1][1] = 1;
zips[1][2] = 0;
zips[1][3] = 4;
zips[1][4] = 4;

zips[2] = NULL;
```

Why no NULL?

Why terminate with NULL?



Zip code




```
// return a count of all zips that end with digit endNum
int zipCount(int* zips[], int endNum) {




}
```

`scanf` reads formatted input

```
int val;

...

scanf("%d", &val);
```

Declared, but not initialized. Holds anything.

Read one `int` in decimal$_{10}$ format **from input.**

Store it in memory at this address.

Store in memory at the address given by the **address of `val`**: store input `@ 0x7F…F38`.

		0x7FFFFFFFFFFFF3C
val	CA FE 12 34	0x7FFFFFFFFFFFF38
		0x7FFFFFFFFFFFF34

C: Classic bug using `scanf`

!!!

```
int val;

...

scanf("%d",  val);
```

Declared, but not initialized.
Holds anything.

Read one `int` in decimal$_{10}$ format **from input.**

Store it in memory at this address.

Store in memory at the address given by the **contents of `val`** (implicitly cast as a pointer): store input @ `0xBAD4FACE`.

		0x7FFFFFFFFFFFFF3C
val	BA D4 FA CE	0x7FFFFFFFFFFFFF38
		0x7FFFFFFFFFFFFF34
	...	...
	CA FE 12 34	0x00000000BAD4FACE

Best case: 🤦 **crash immediately** with segmentation fault/bus error.

Bad case: 🤬 **silently corrupt data** stored @ `0xBAD4FACE`, **fail to store input in `val`**, *and keep going.*

Worst case: 💻🔥🧨🚀 program does literally anything.

C: Memory error messages

11: **segmentation fault** ("**segfault**", SIGSEGV)
accessing address outside legal area of memory

10: **bus error** (SIGBUS)
accessing misaligned or other problematic address

More to come on debugging!

http://xkcd.com/371/

C: Why?

Why learn C?

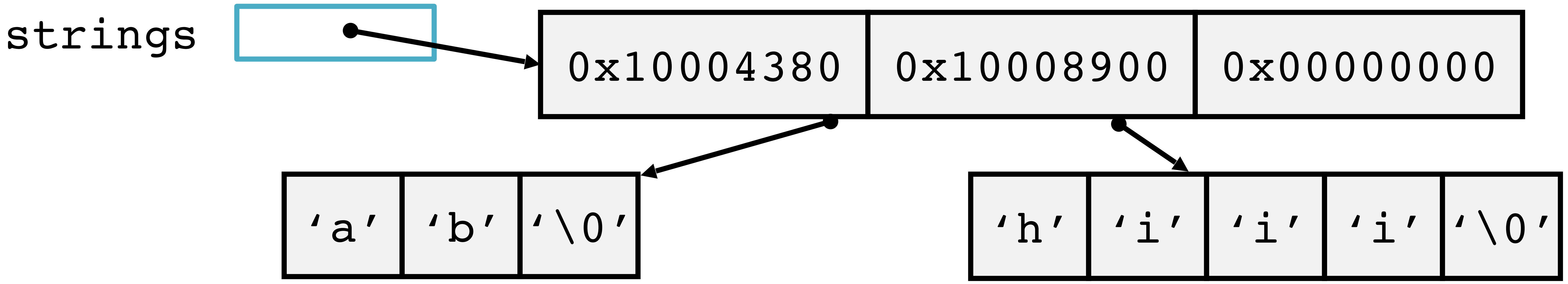
- Think like actual computer (abstraction close to machine level) without dealing with machine code.
- Understand just how much Your Favorite Language provides.
- Understand just how much Your Favorite Language might cost.
- Classic.
- Still (more) widely used (than it should be).
- Pitfalls still fuel devastating reliability and security failures today.

Why not use C?

- Probably not the right language for your next personal project.
- It "gets out of the programmer's way" … even when the programmer is unwittingly running toward a cliff.
- Advances in programming language design since the 70's have produced languages that fix C's problems while keeping strengths.

Group example: longest string starts with

ex



output: 'h'

```
// Return the starting character of the longest string in the
// null-terminated strings array.
// You can use: int strlen(char *str)
char longest_string_starts_with(char ** strings) {



}
```