CS 240 Stage 2

Hardware-Software Interface

Memory addressing, C language, pointers

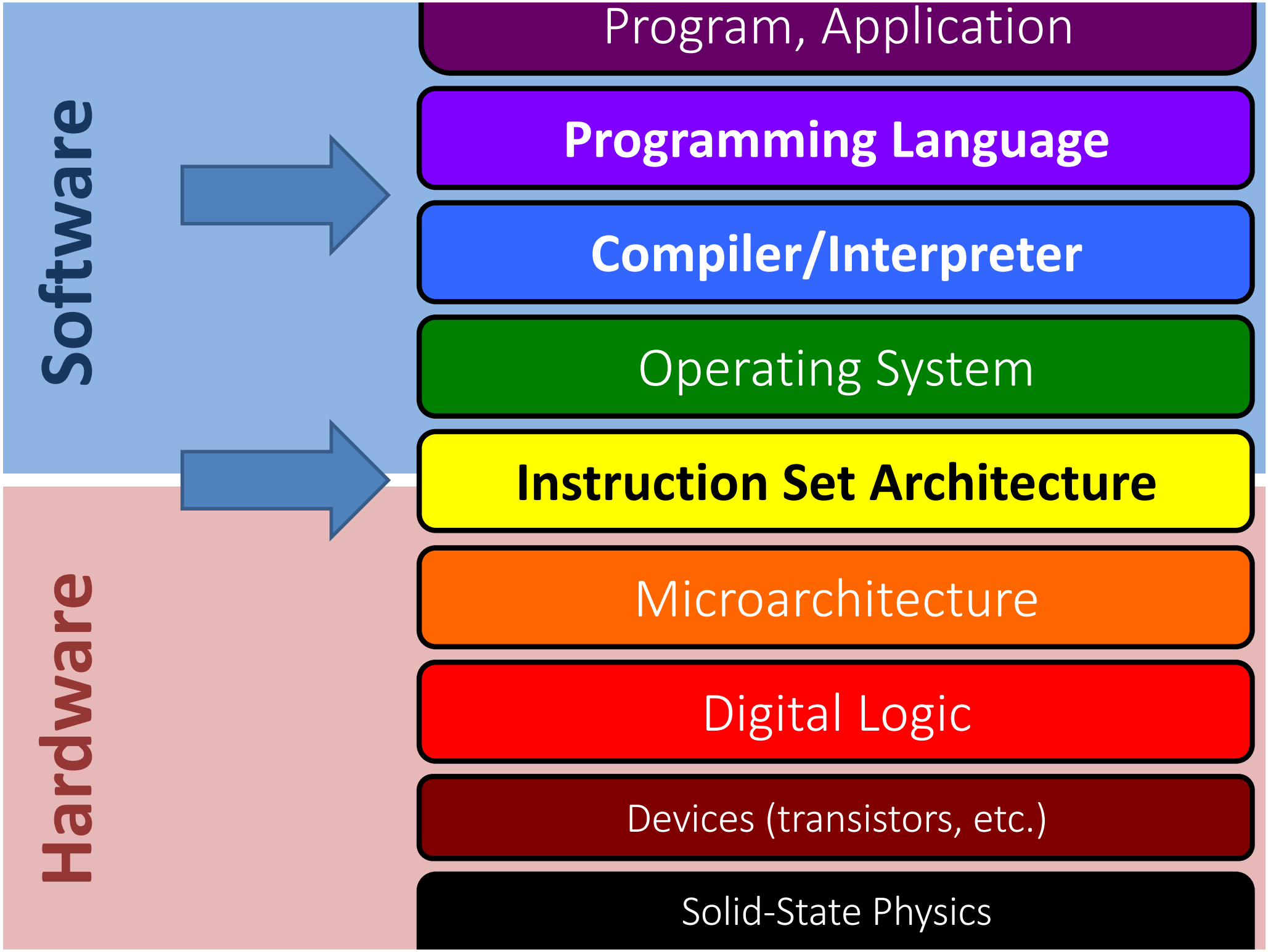
Assertions, debugging

Machine code, assembly language, program translation

Control flow

Procedures, stacks

Data layout, security, linking and loading

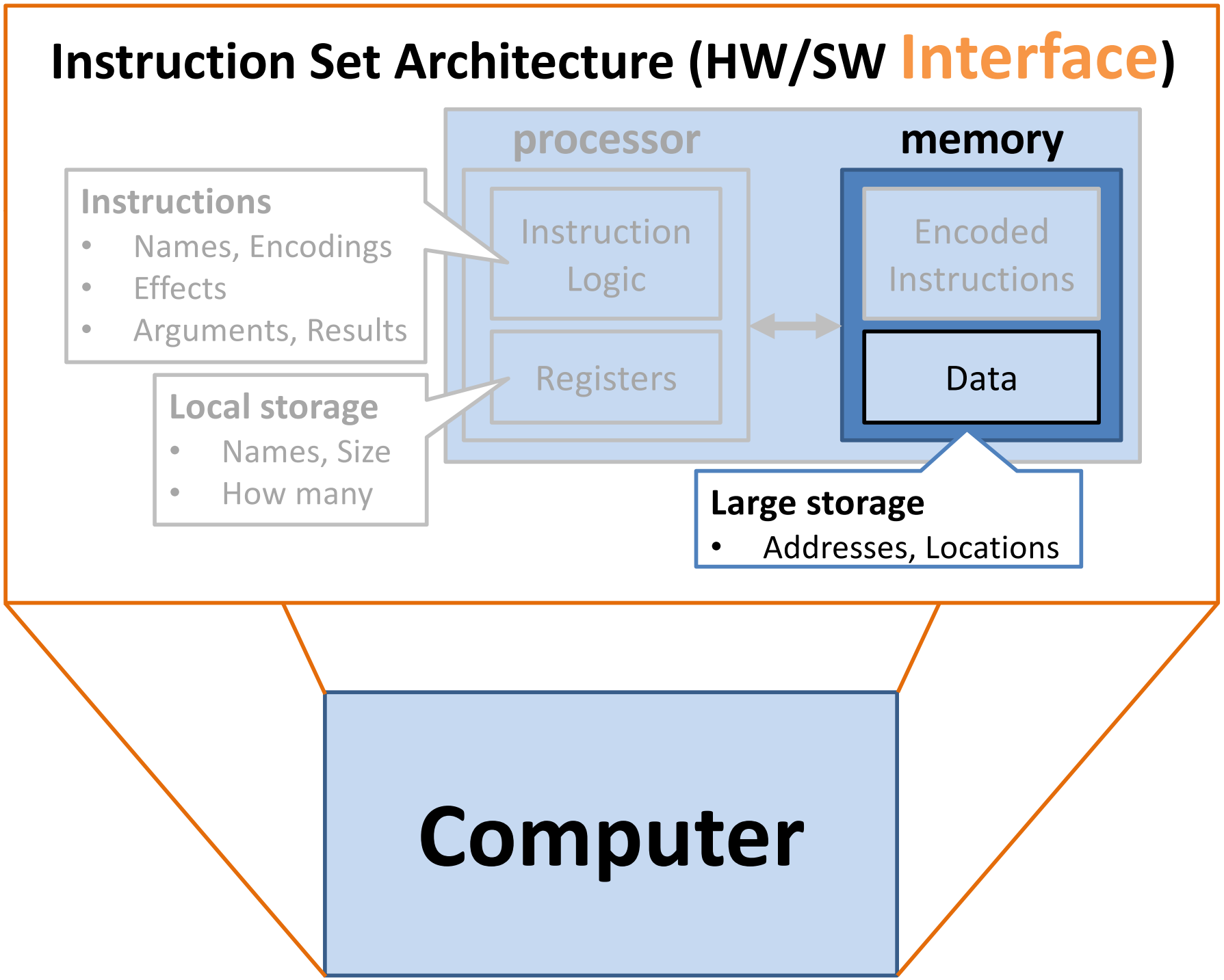

Software
Hardware
Program, Application
Programming Language
Compiler/Interpreter
Operating System
Instruction Set Architecture
Microarchitecture
Digital Logic
Devices (transistors, etc.)
Solid-State Physics

Programming with Memory

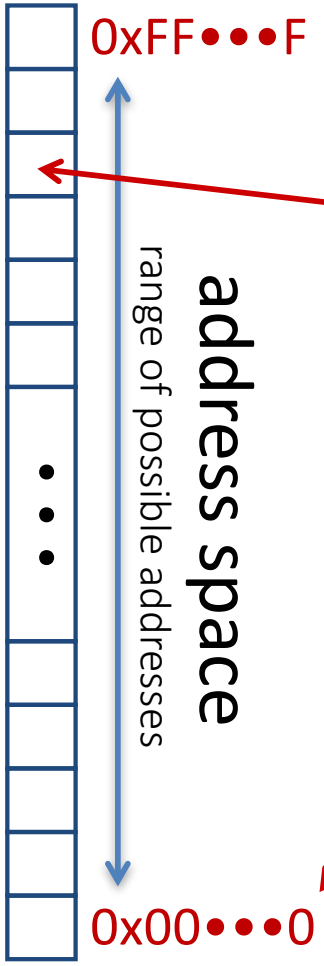
via C, pointers, and arrays

Why not just registers?

- Represent larger structures
- Computable addressing
- Indirection


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

byte-addressable memory = mutable byte array



Cell / location = element

- Addressed by unique numerical address
- Holds one byte
- Readable and writable

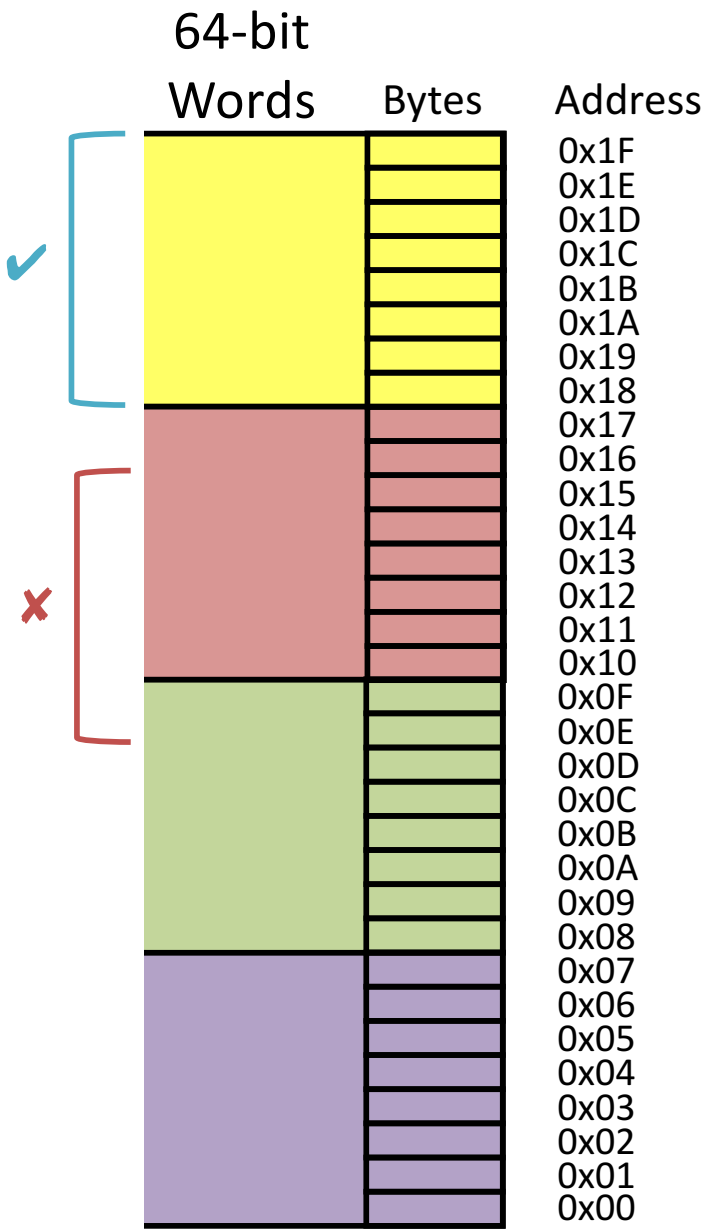
Address = index

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

multi-byte values in memory

Store across contiguous byte locations.

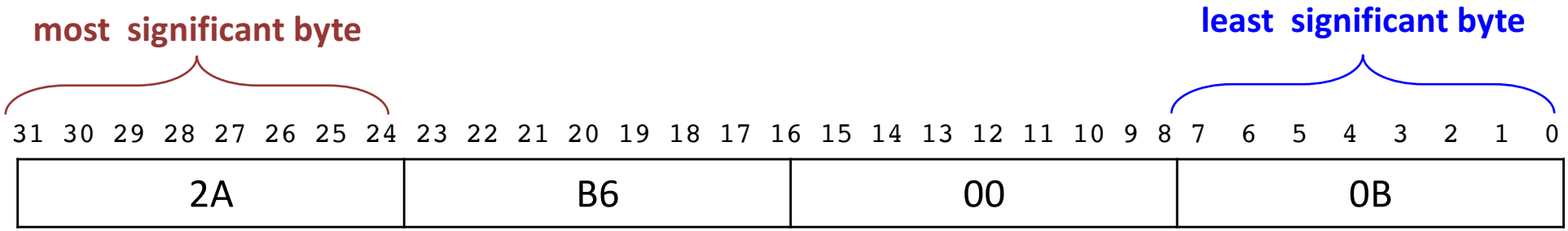
Alignment (Why?)



Bit order within byte always same.

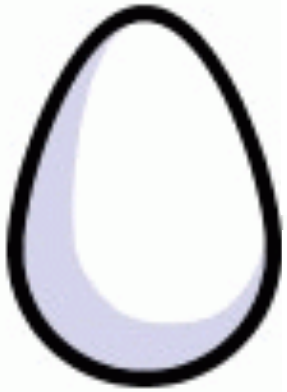
Byte ordering within larger value?

Endianness: To store a multi-byte value in memory, which byte is stored first (at a lower address)?



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

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



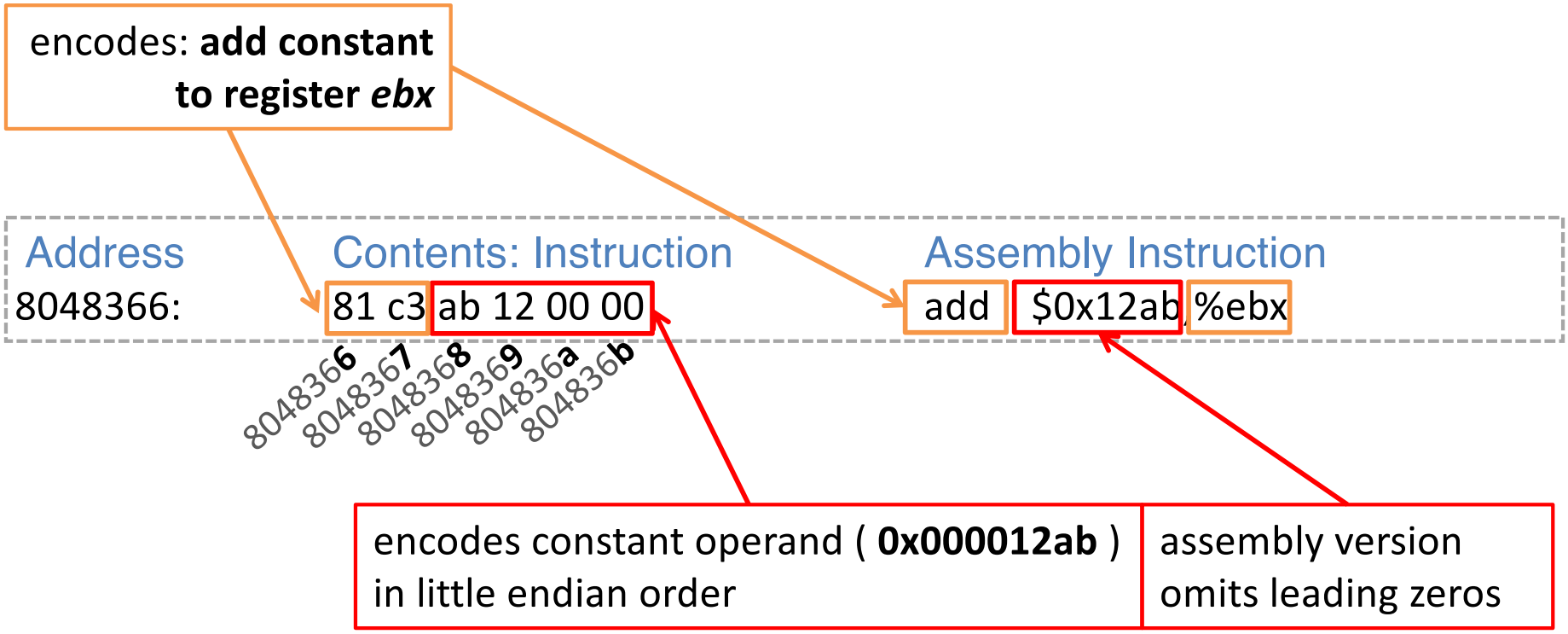
Little Endian: least significant byte first

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

Big Endian: most significant byte first

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

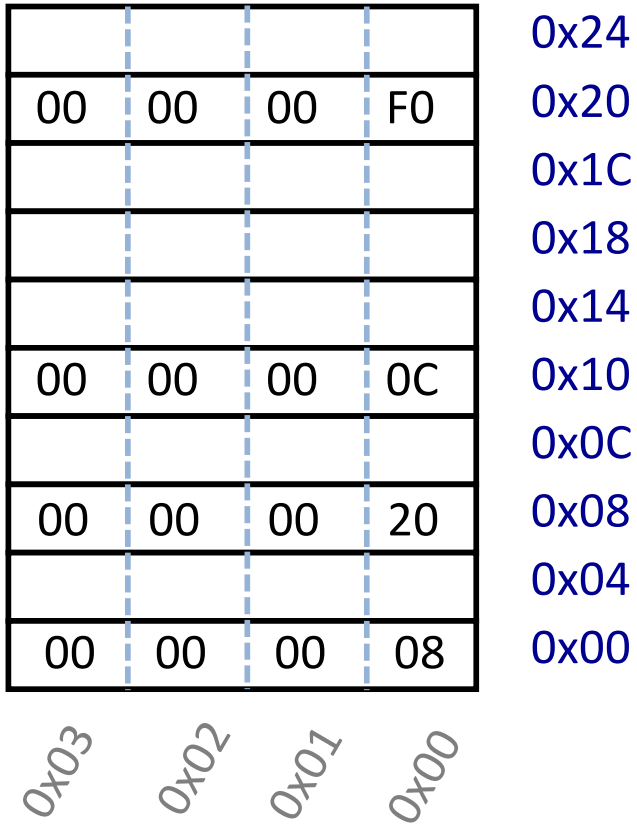
Endianness in Machine Code



Data, Addresses, and Pointers

address = index of a cell in memory

pointer = address represented as data



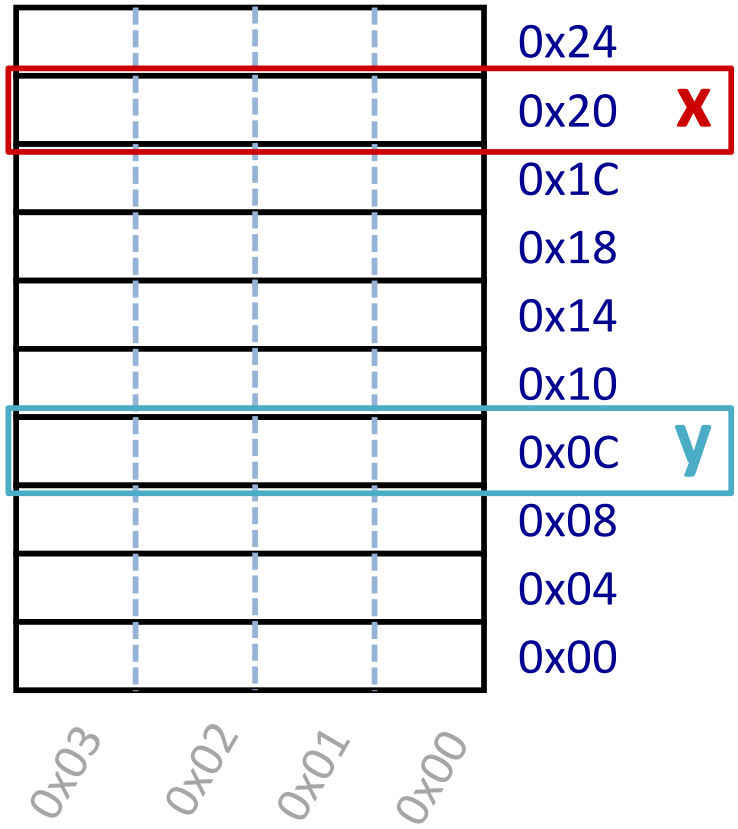
memory drawn as 32-bit values,
little endian order

C: variables are memory locations (for now)

Compiler maps variable → memory location.

Declarations do not initialize!

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

x = 0; // store 0 at 0x20

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

// load the contents at 0x0C,
// add 3, and store sum at 0x20
x = y + 3;
```



C: Address and Pointer Primitives

address = index of a cell/location in memory

pointer = address represented as data

Expressions using addresses and pointers:

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

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

a.k.a. "dereference ___"

Pointer types:

___* address of a memory location holding a ___

C: Address and Pointer Example

& = address of
** = contents at*

```
int* p;


int x = 5;
int y = 2;


p = &x;



y = 1 + *p;
```

C: Address and Pointer Example

& = address of
** = contents at*

Declare a variable, p

```
int* p;
```

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

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

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

Get the address of the memory location representing x

```
p = &x;
```

... and store it in p. Now, "*p points to x.*"

Add 1 to the contents of memory at the address stored in p

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

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

C: Address and Pointer Example

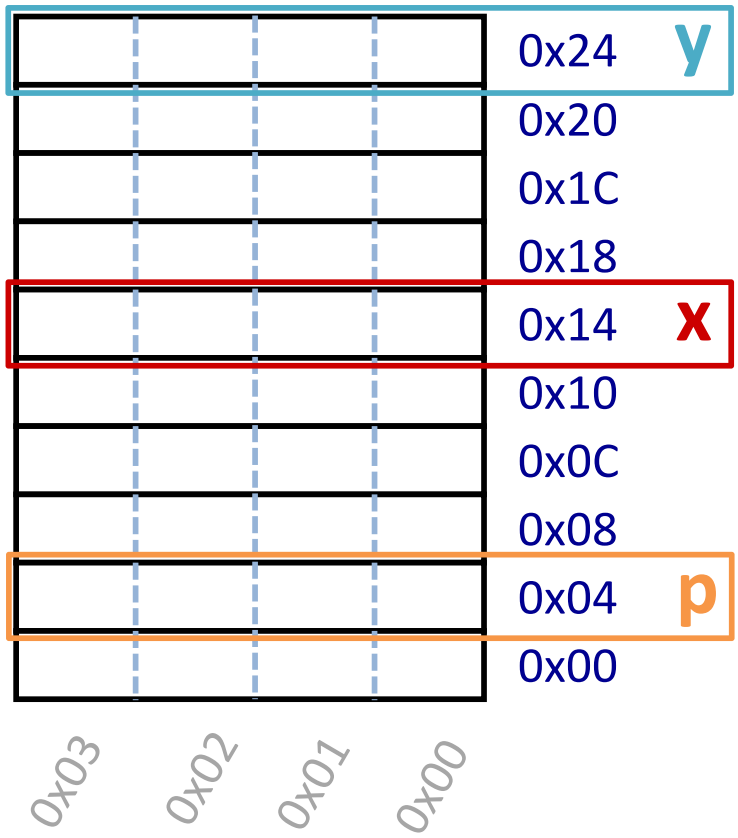
& = address of
** = contents at*

C assignment:

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

location value

```
int* p;      // p: 0x04
int x = 5;   // x: 0x14, store 5 at 0x14
int y = 2;   // y: 0x24, store 2 at 0x24
p = &x;      // store 0x14 at 0x04
// load the contents at 0x04  (0x14)
// load the contents at 0x14  (0x5)
// add 1 and store sum at 0x24
y = 1 + *p;
// load the contents at 0x04  (0x14)
// store 0xF0 (240) at 0x14
*p = 240;
```



C: Pointer Type Syntax

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

The following are equivalent:

```
int* ptr;
```

I prefer this

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

```
int * ptr;
```

```
int *ptr;
```

more common C style

I see: "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

Declaration: `int a[6];`

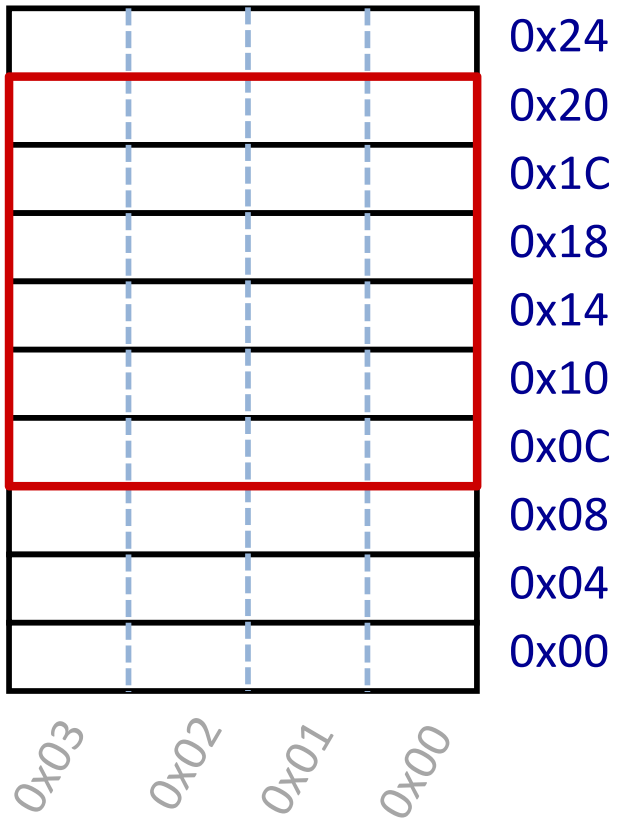
element type

name

number of elements

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.



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;
```

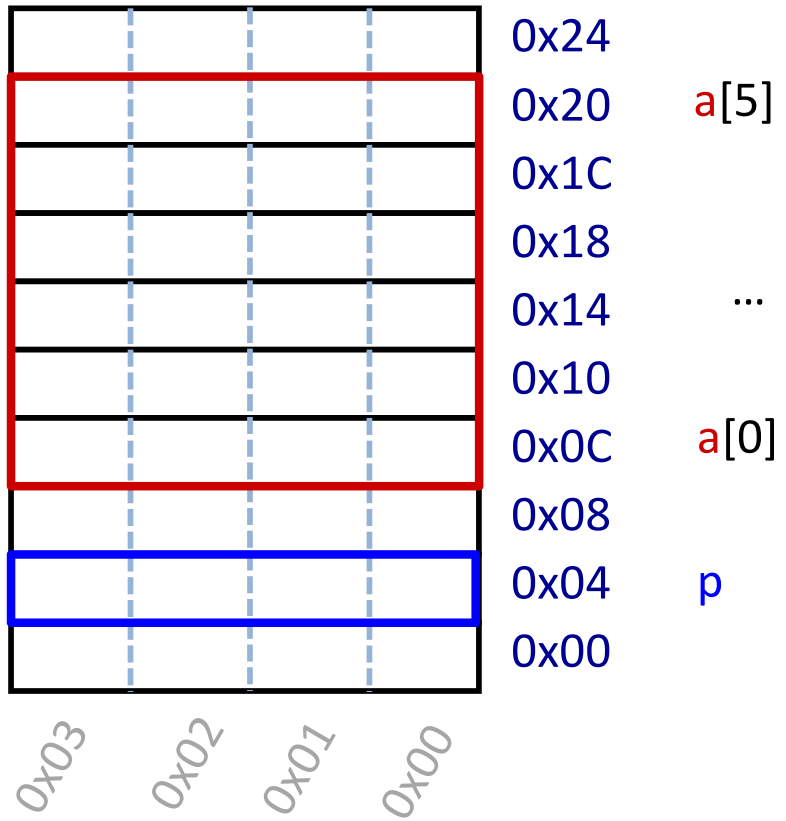
equivalent

```
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;
```



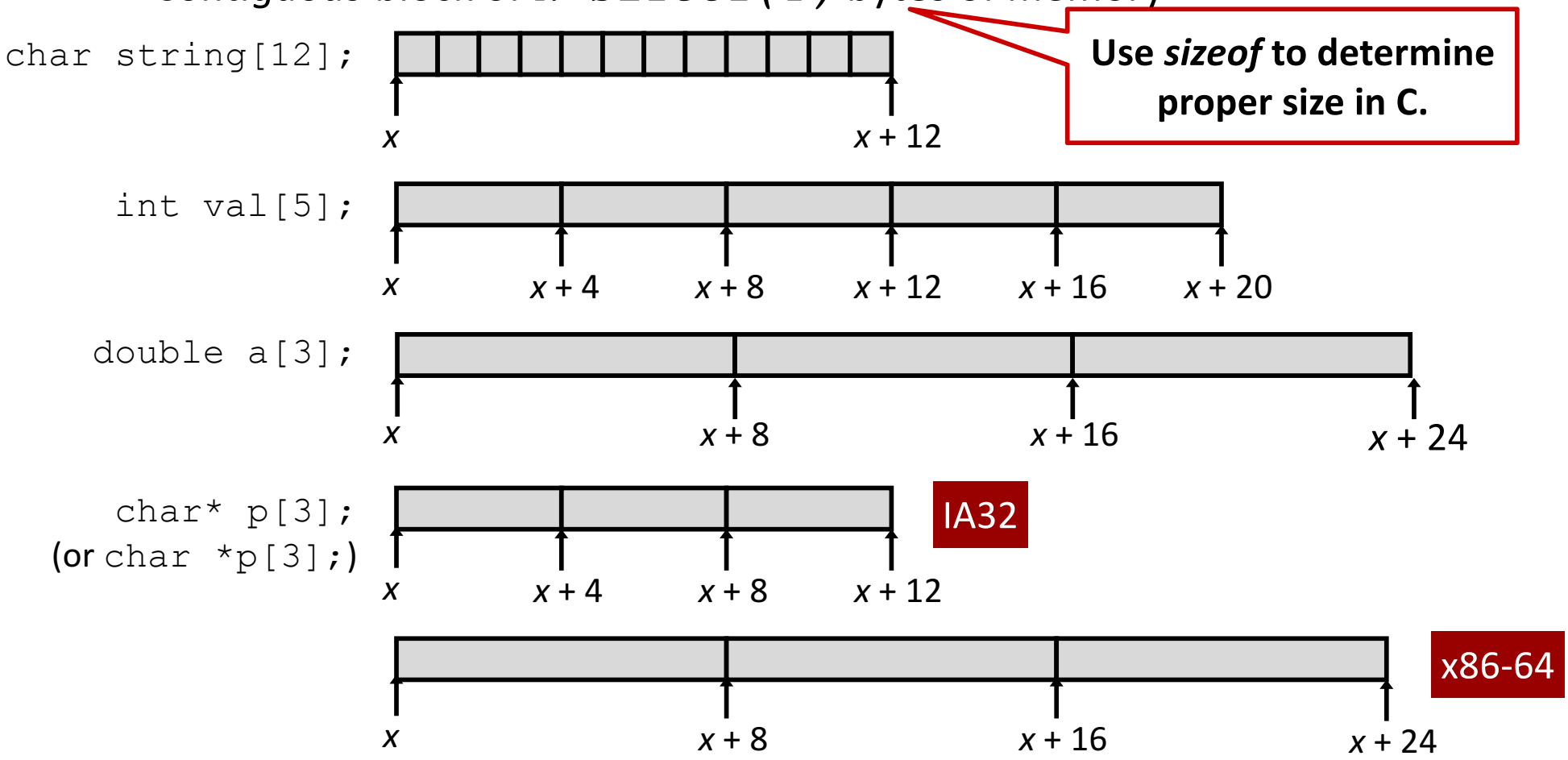
C: Array Allocation

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



C: Array Access

ex

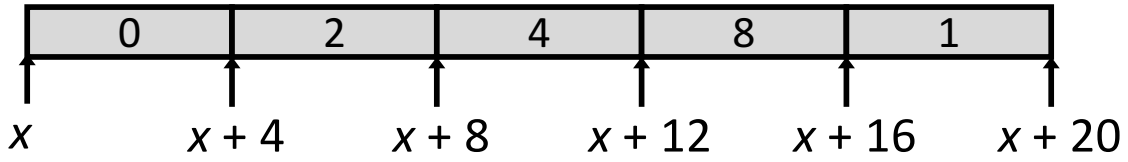
Basic Principle

T *A*[*N*];

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

Identifier *A* has type

```
int val[5];
```

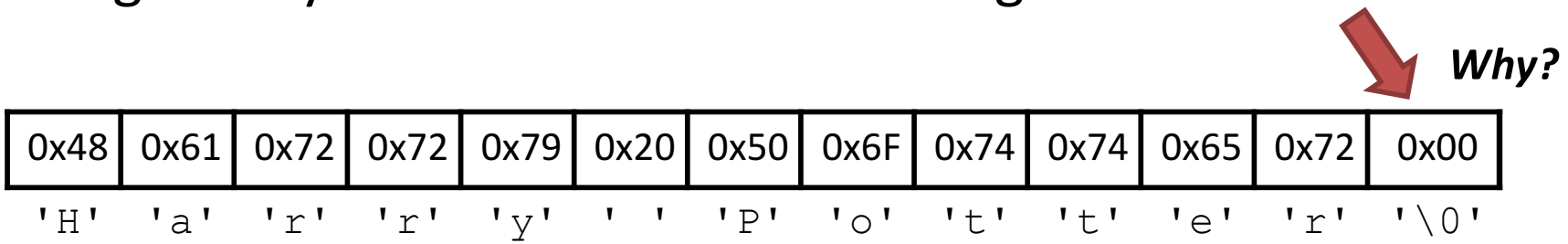


Reference	Type	Value
val[4]	int	
val	int *	
val+1	int *	
&val[2]	int *	
val[5]	int	
*(val+1)	int	
val + i	int *	

C: Null-terminated strings

ex

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



Does Endianness matter for strings?

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



}
```

C: * *and* [ ]

ex

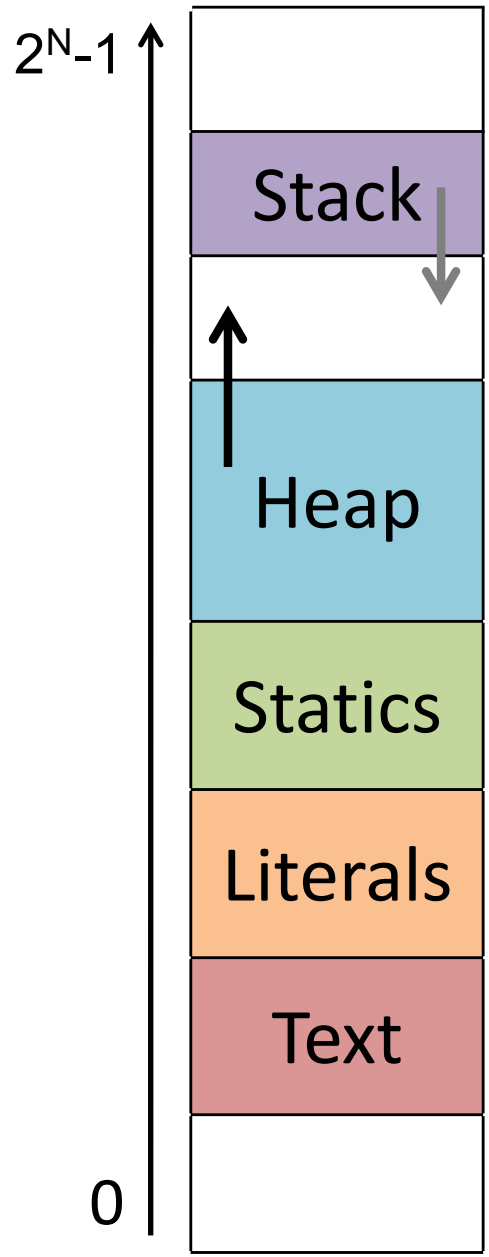
C programmers often use * where you might expect []:

e.g., char*:

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

```
int strcmp(char* a, char* b);
int string_length(char* str) {
  // Try with pointer arithmetic, but no array indexing.




}
```

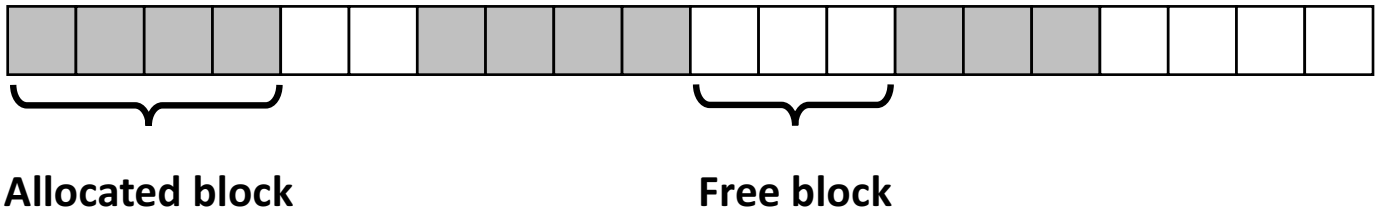
Memory Layout



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

Heap:



Managed by memory allocator:

pointer to newly allocated block of at least that size

number of contiguous bytes required

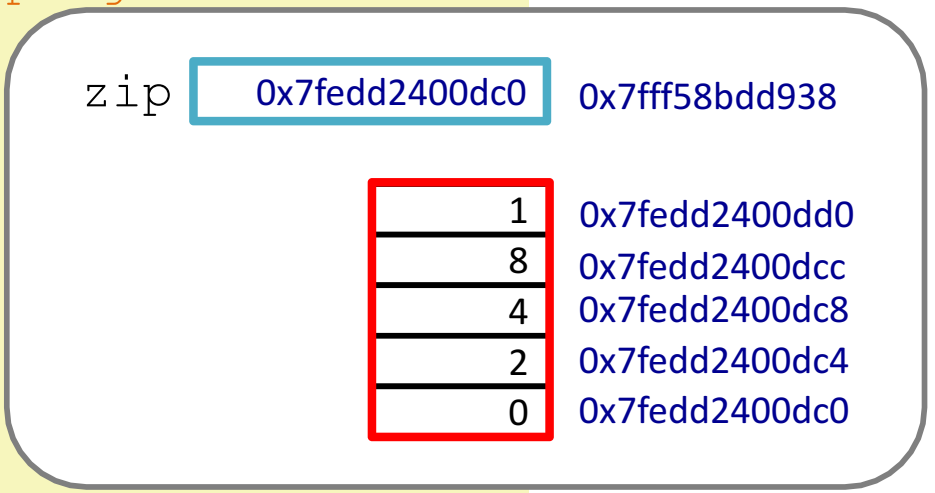
```
void* malloc(size_t size);
```

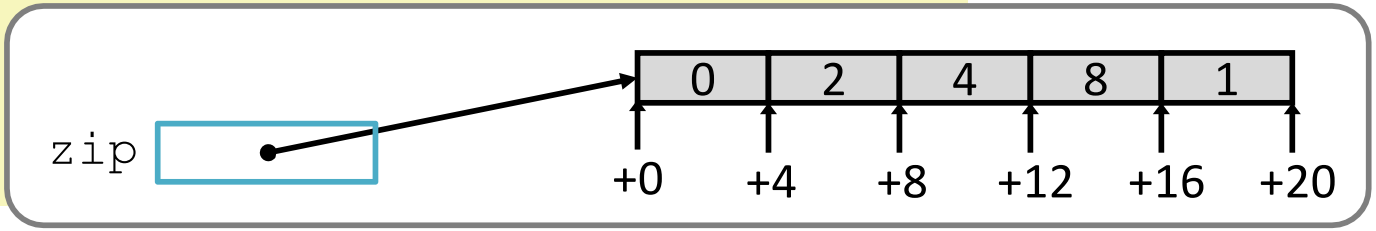
pointer to allocated block to free

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

C: Dynamic array allocation

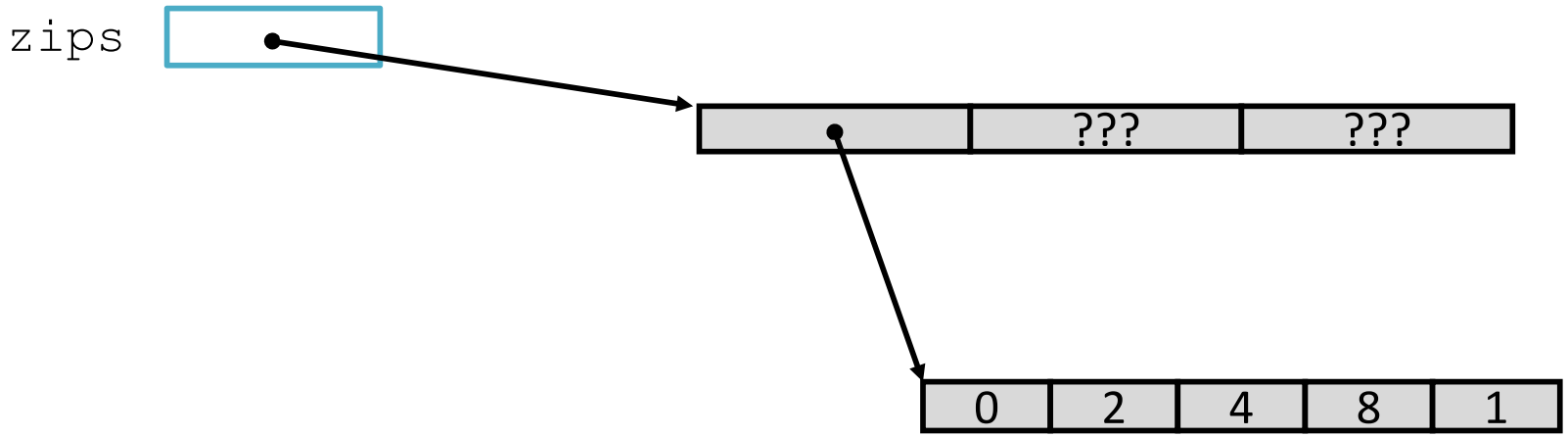
```
#define ZIP_LENGTH 5
int* zip = (int*)malloc(sizeof(int)*ZIP_LENGTH);
if (zip == NULL) {    // if error occurred
  perror("malloc");   // print error message
  exit(0);            // end the program
}

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);
```

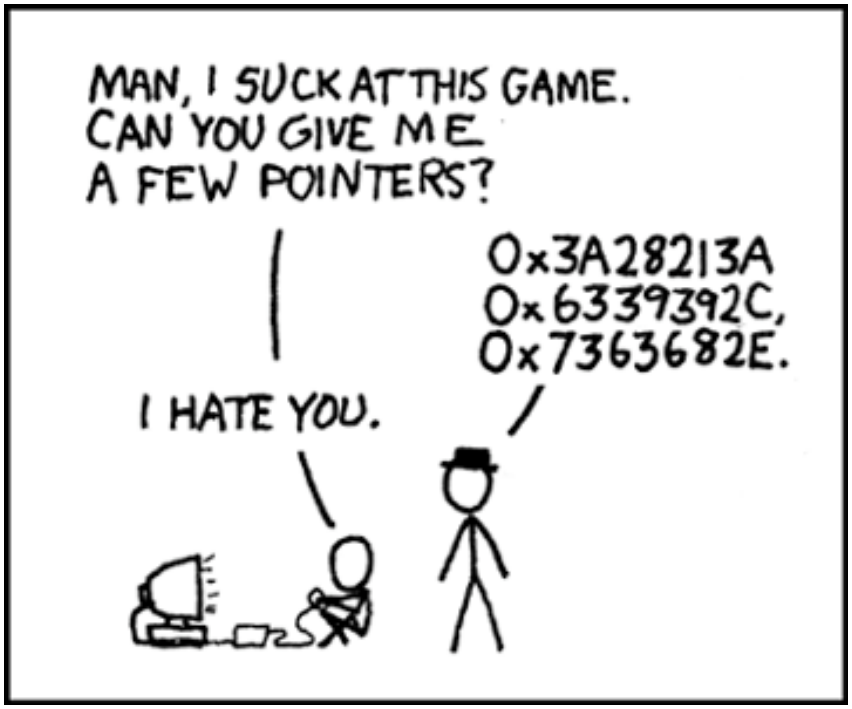




C: Arrays of pointers to arrays of …

```
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;
```





http://xkcd.com/138/

C: scanf reads formatted input

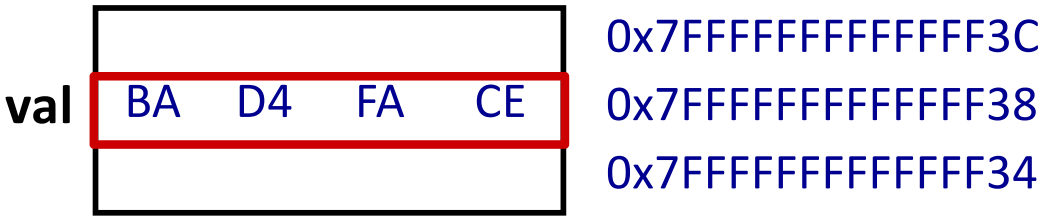
```
int val;

...

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

Declared, but not initialized – holds anything.

Read one int from input.

Store it in memory at this address.

i.e., store it in memory at the address **where the contents of val is stored**: store into memory at 0xFFFFFF38.



C: classic bug using scanf

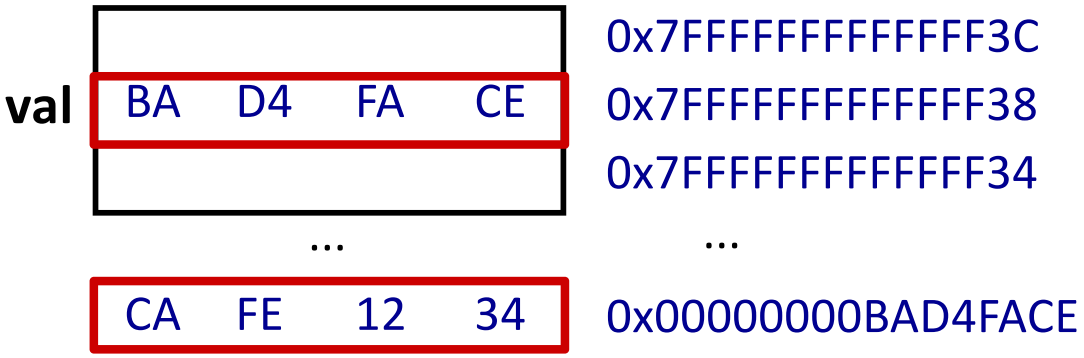
```
int val;

...

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

Declared, but not initialized – holds anything.

Read one int from input.

Store it in memory at this address.

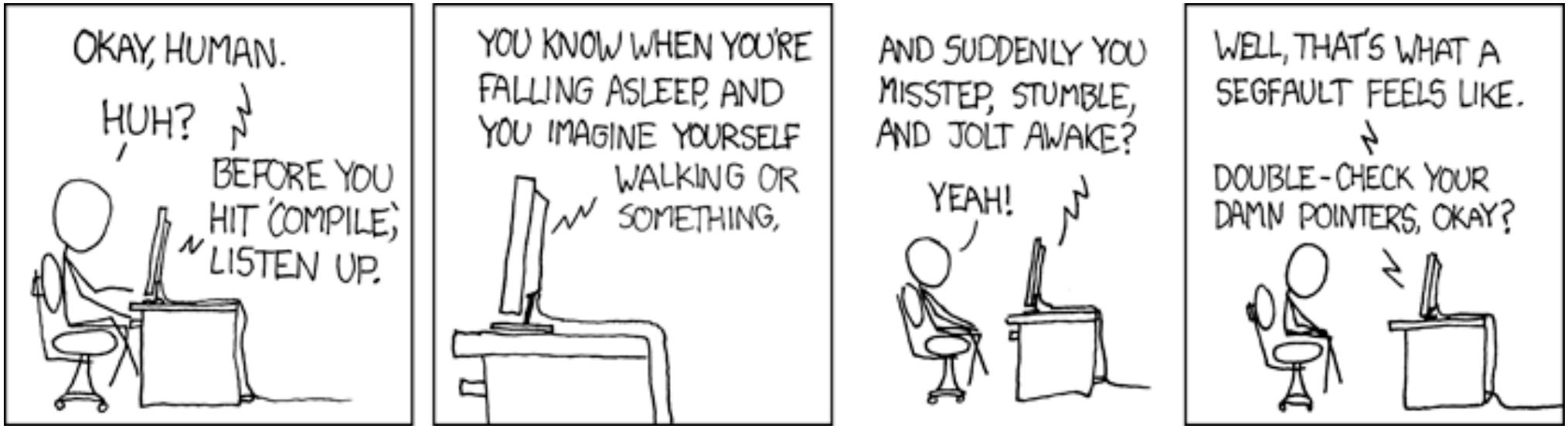
i.e., store it in memory at the address **given by the contents of val**:
store into memory at 0xBAD4FACE.



Best case: segmentation fault, or bus error, crash.

Bad case: silently corrupt data stored at address 0xBAD4FACE, and val still holds 0xBAD4FACE.
Worst case: arbitrary corruption

C: memory error messages



http://xkcd.com/371/

11: segmentation fault ("segfault", SIGSEGV)

accessing address outside legal area of memory

10: bus error

accessing misaligned or other problematic address

More to come on debugging!

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.
- Many advances in programming language design since then have produced languages that fix C's problems while keeping strengths.