CS 240 Stage 2

Hardware-Software Interface

Memory addressing, C language, pointers

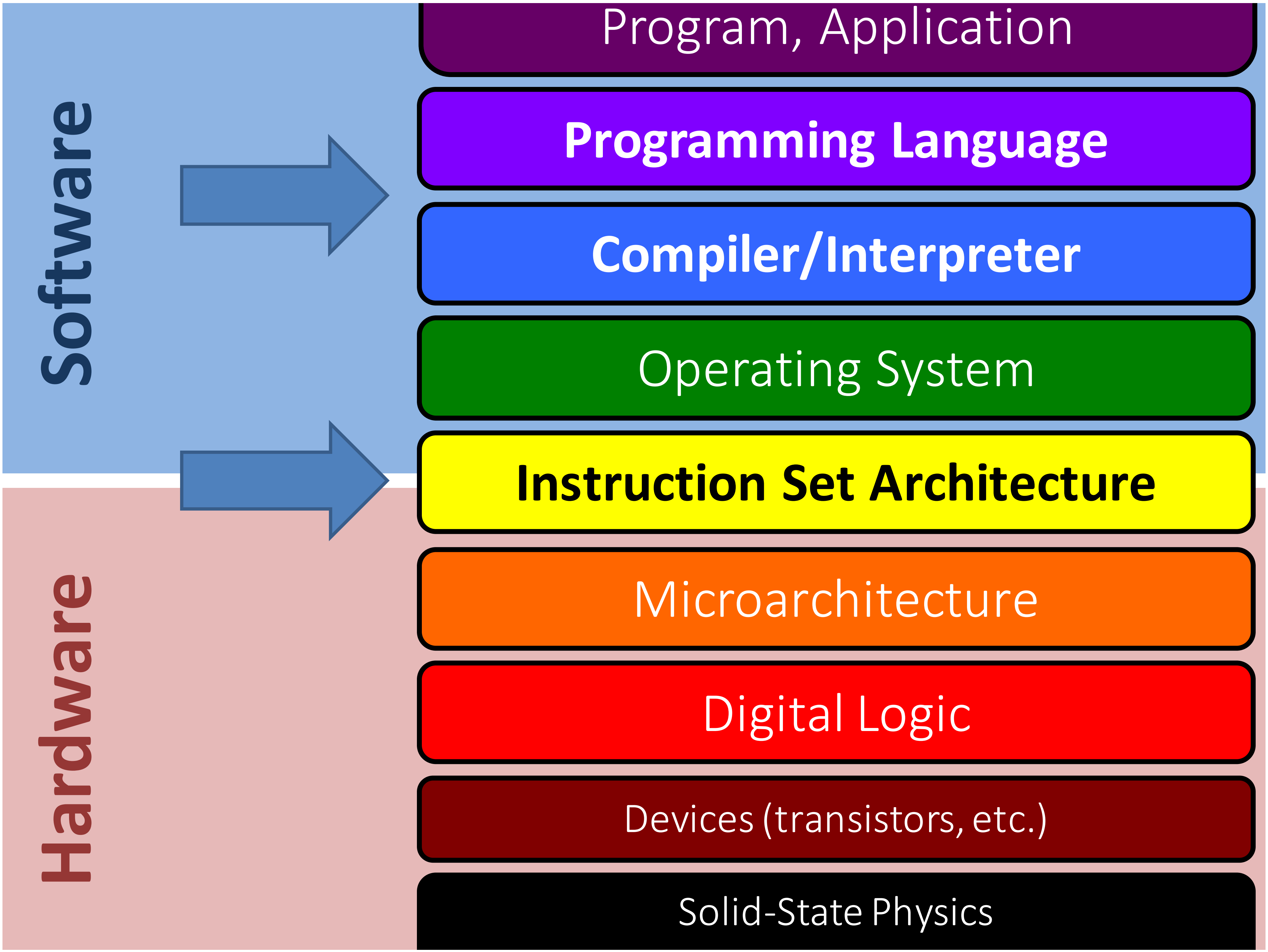
Assertions, debugging

Machine code, assembly language, program translation

Control flow

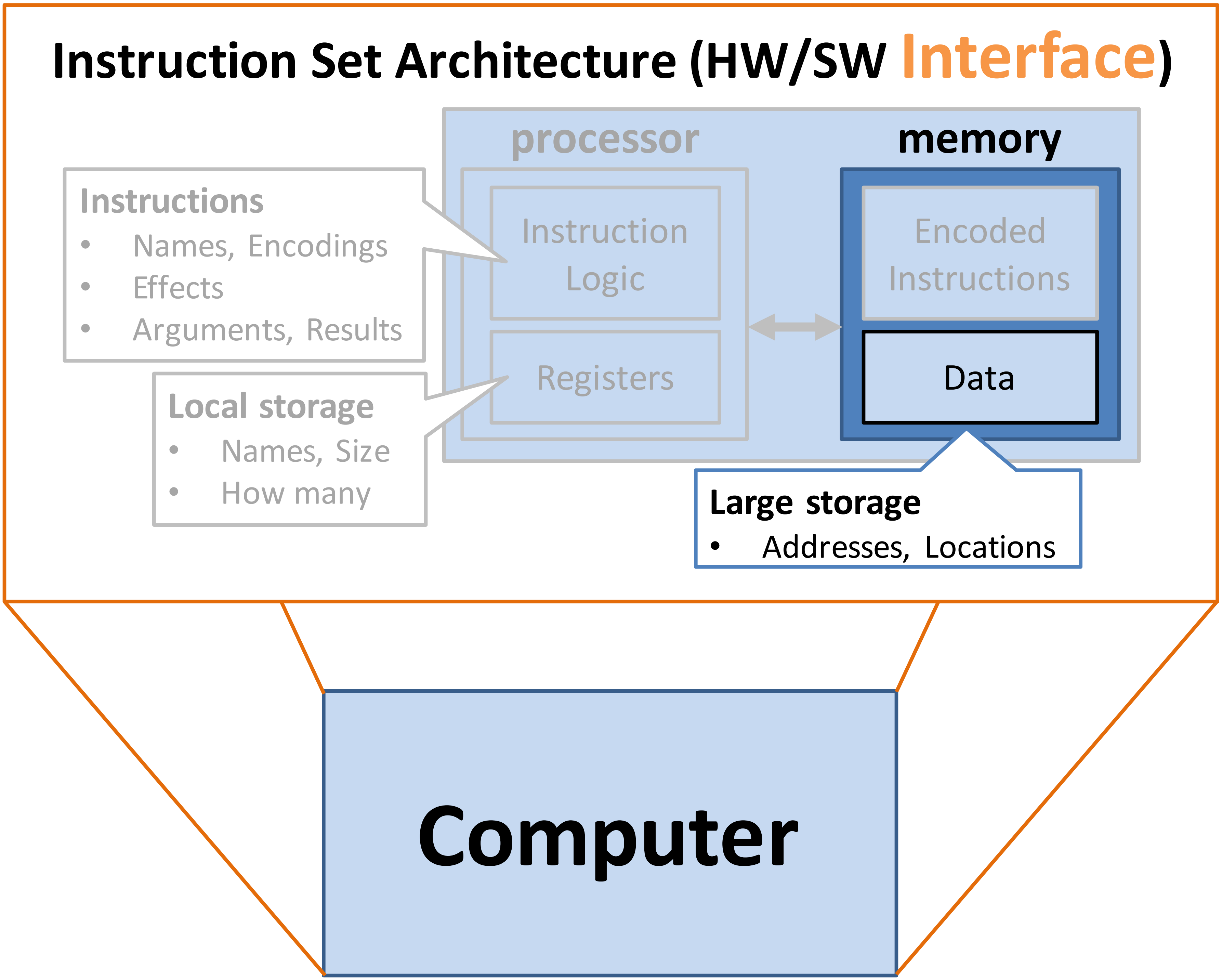
Procedures, stacks

Data layout, security, linking and loading

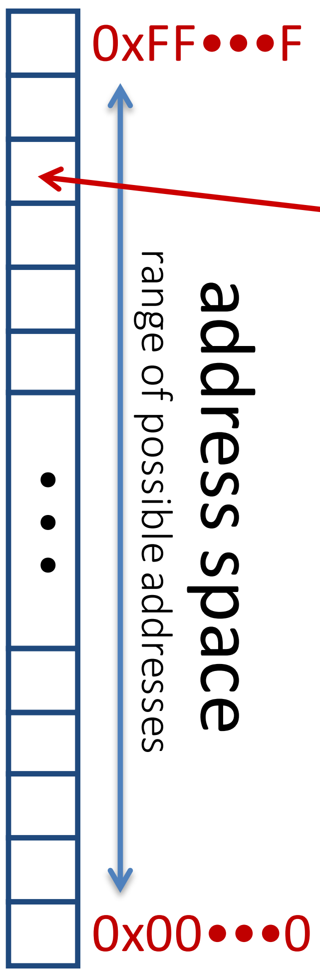

Software
Hardware
Program, Application
Programming Language
Compiler/Interpreter
Operating System
Instruction Set Architecture
Microarchitecture
Digital Logic
Devices (transistors, etc.)
Solid-State Physics

Programming with Memory

via C, pointers, and arrays


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



Fixed-length ordered sequence of cells

Cell = location = element

- Addressed by a unique numerical address
- Holds one byte
- Can be read and written by program

Address = index

- Unsigned number
- Represented by one word
- Can be computed and stored

multi-byte values in memory

Use *N* contiguous byte locations to store an *N*-byte value.

Alignment

Data of size *N bytes* stored at *A* only if *A* mod *N* = 0

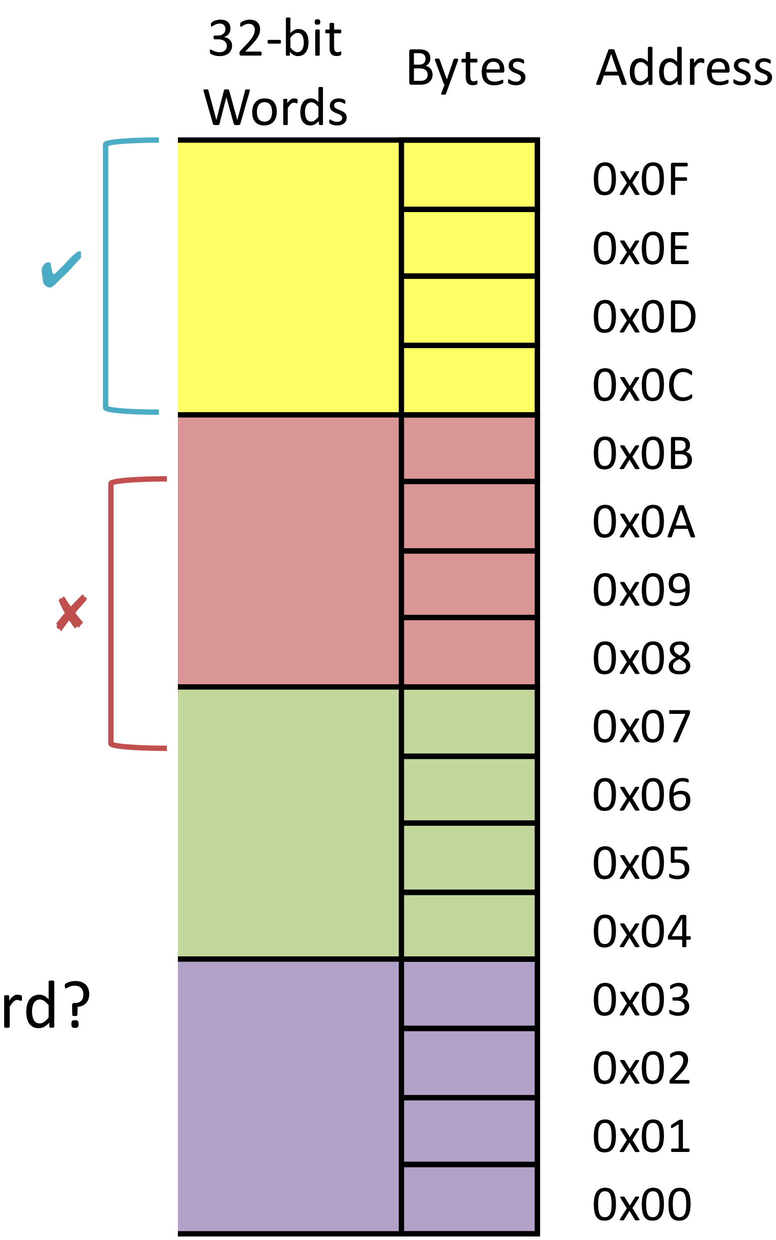
N is a power of 2

Recommended (x86) or required

Why?

Byte ordering:

Which byte is "first" in a multi-byte word?



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

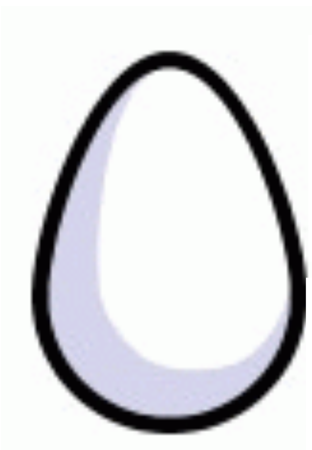
$$\underbrace{\text{31 30 29 28 27 26 25 24}}_{\text{most significant byte}}\ \text{23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8}\ \underbrace{\text{7 6 5 4 3 2 1 0}}_{\text{least significant byte}}$$

word in positional hexadecimal notation

2A	B6	00	0B

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

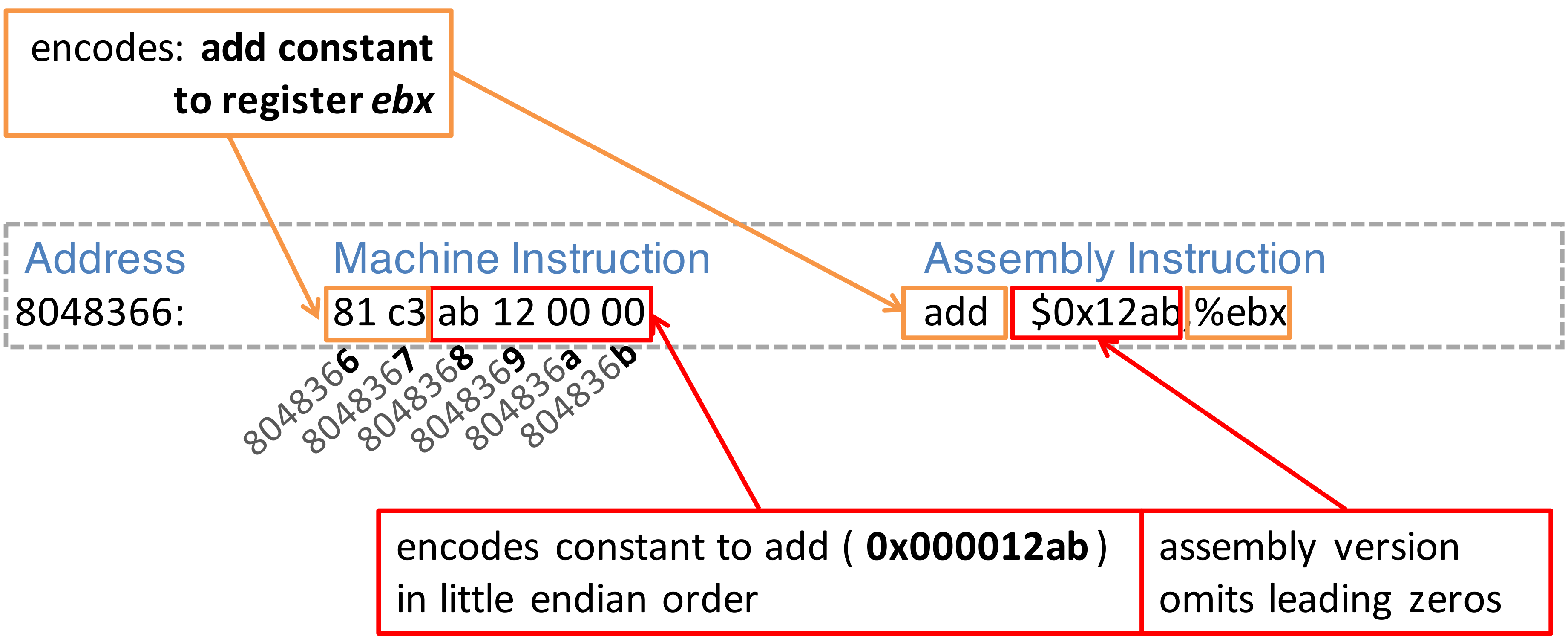
- low order byte at low address, high order byte at high address
- used by **x86**

Big Endian: most significant byte first

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

Bit order within bytes is always the same.

Endianness in x86 Machine Code



Data, Addresses, and Pointers

address **= number of a location in memory**

pointer **= data that holds an address**

The number 240 is stored at address 0x20.

$240_{10} = F0_{16} = 0x00\ 00\ 00\ F0$

A **pointer** stored at address 0x08
points to the contents at address 0x20.

A **pointer to a pointer**
is stored at address 0x00.

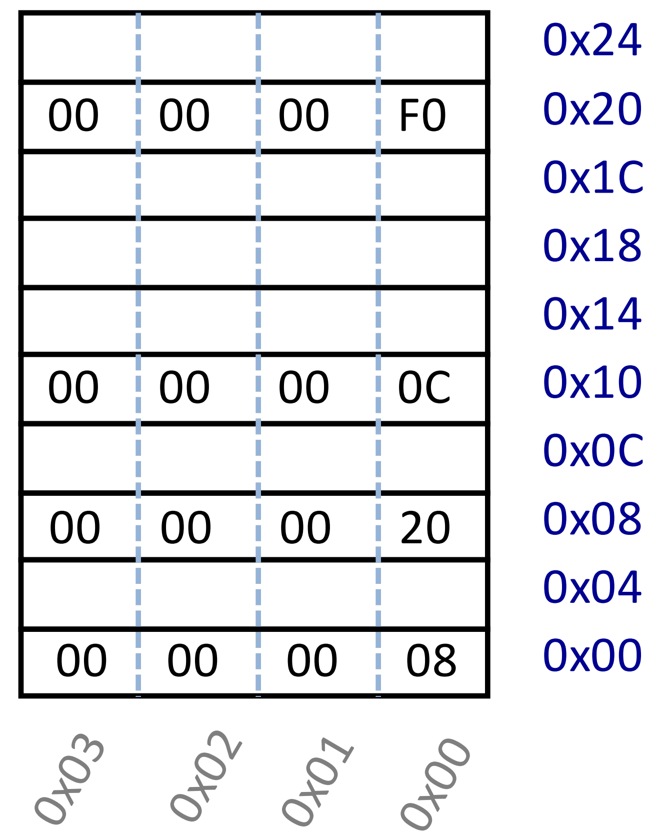
The number 12 is stored at address 0x10.

Is it a pointer?

How do we know values are pointers or not?

How do we manage use of memory?

memory drawn as words

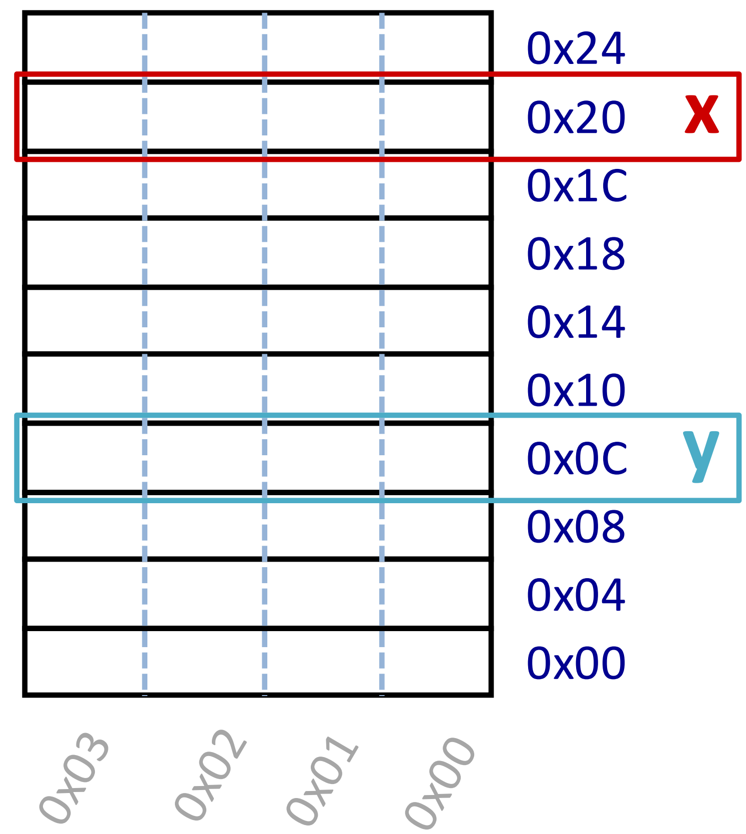


C: variables are memory locations (for now)

Compiler manages the mapping from variable to memory.

Declarations do not initialize!

```
int x; // x stored at 0x20
int y; // y stored 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: Types determine sizes

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: Addresses and Pointers

& = 'address of'
** = 'contents at address' or 'dereference'*

```
int* p;
```

Declare a variable, `p`, of type int* that is a pointer to (*i.e.*, holds the address of) an int in memory. (Does not initialize anything.)

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

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

```
p = &x;
```

Set the variable `p` to hold the **address of** `x`. Now, "*p points to x.*"

"Dereference p."

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

Set `y` to hold:
1 plus the contents of memory at the address *held by* `p`.
Because `p` points to `x`, this is equivalent to `y=1+x;`

C: Addresses and Pointers

& = ‘address of’
* = ‘contents at address’
or ‘dereference’

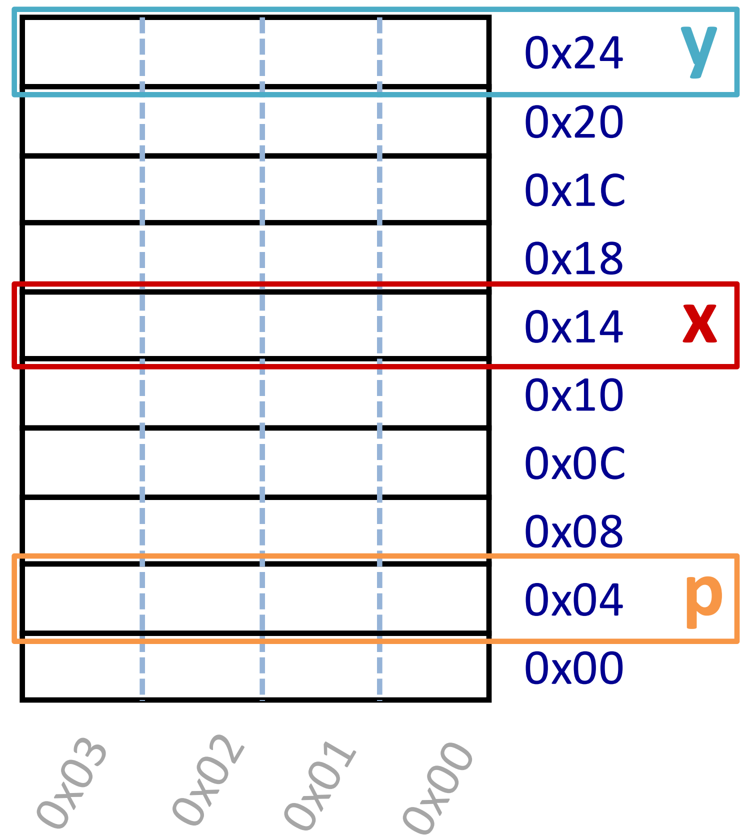
Left-hand-side = right-hand-side;

RHS must provide a *value.*

LHS must provide a *storage location*.

Store RHS value in LHS location.

```
int* p;       // p stored at 0x04
int x = 5;    // x stored at 0x14
int y = 2;    // y stored 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 Types

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.

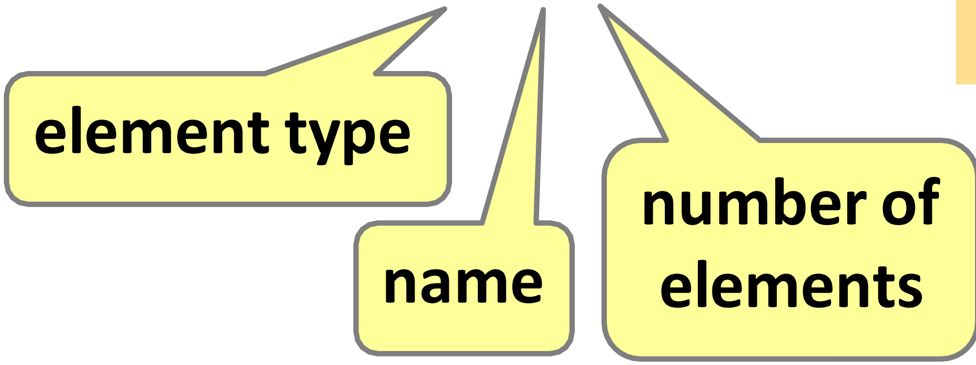
`int* a, b;` means `int *a, b;` means `int* a; int b;`

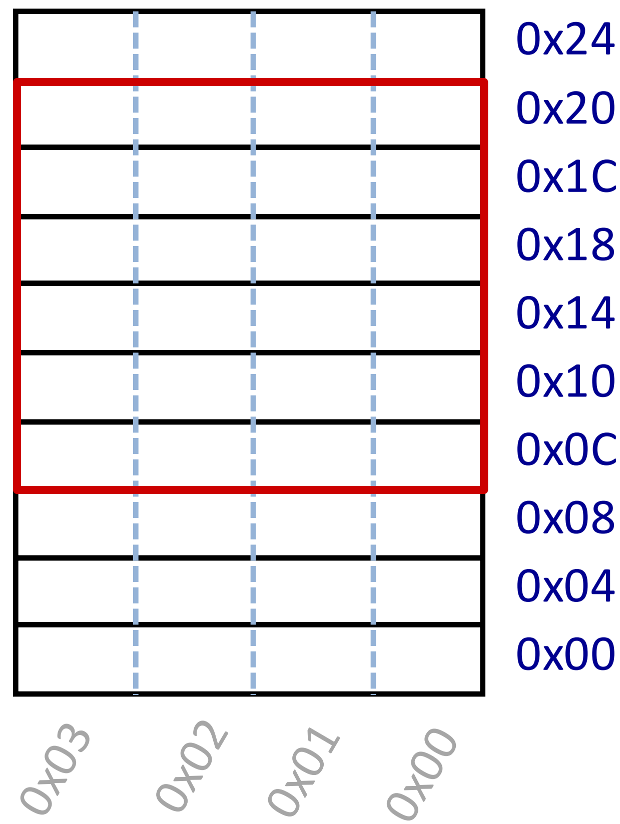
C: Arrays

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

a is a name for the array's address, *not a pointer to the array.*

Declaration: `int a[6];`





C: Arrays

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

a is a name for the array's address, not a pointer to the array.

The address of **a[i]** is address of **a[0]** 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;
```

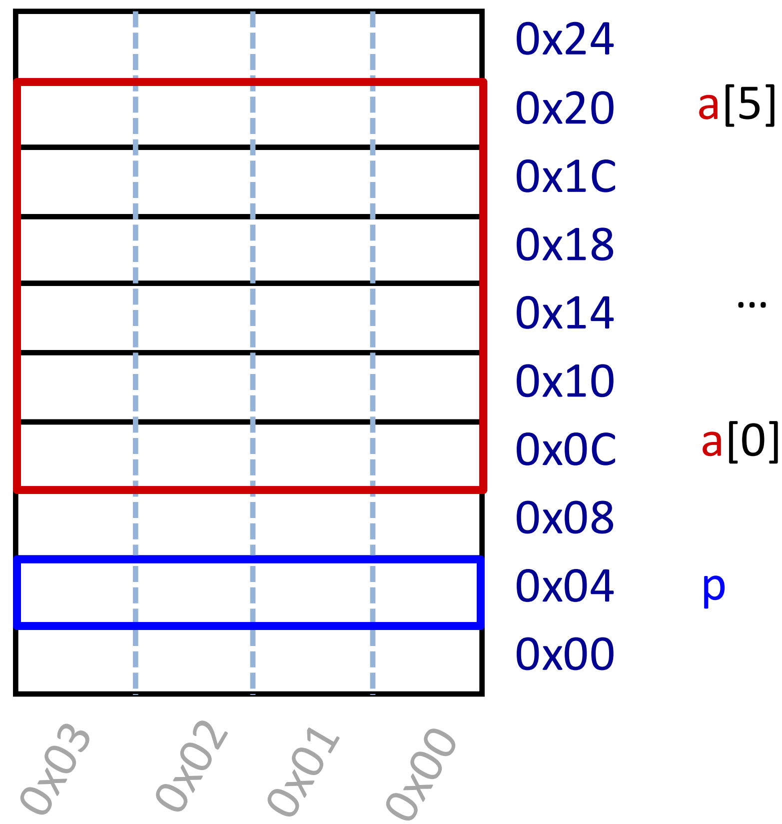
Pointers:
```
int* p;
```
equivalent {
```
p = a;
p = &a[0];
```
```
*p = 0xA;
```

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

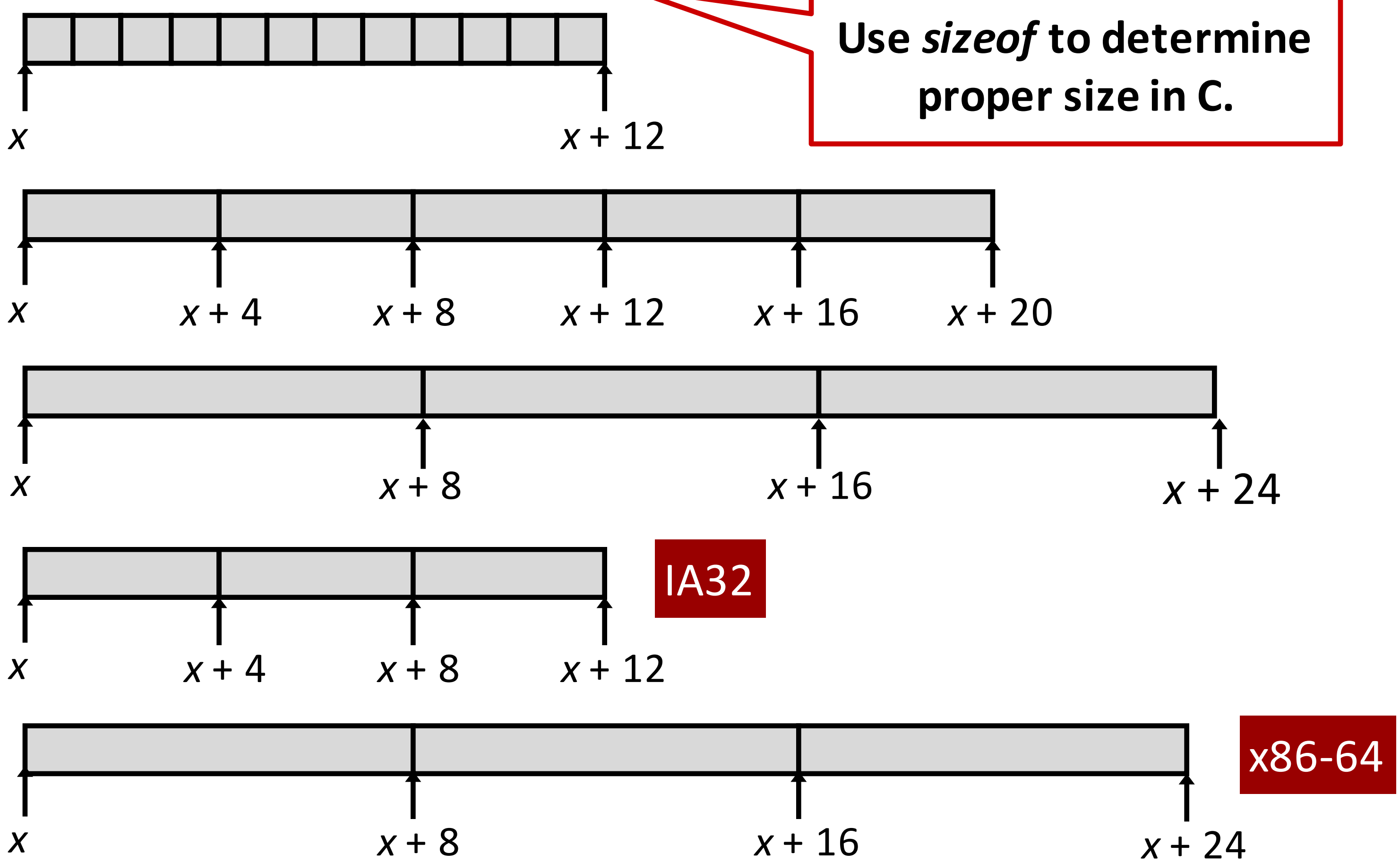
Use *sizeof* to determine proper size in C.

`char string[12];`

`int val[5];`

`double a[3];`

`char* p[3];`
(or `char *p[3];`)



C: Array Access

ex

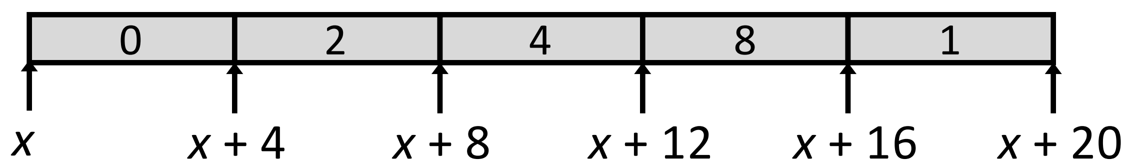
Basic Principle

$T \quad A[N];$

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

Identifier A can be used as a pointer to array element 0: A has type $T*$

```
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.

Why?

0x48	0x61	0x72	0x72	0x79	0x20	0x50	0x6F	0x74	0x74	0x65	0x72	0x00
'H'	'a'	'r'	'r'	'y'	' '	'P'	'o'	't'	't'	'e'	'r'	'\0'

Does Endianness matter for strings?

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



}
```

C: * *and* [ ]

ex

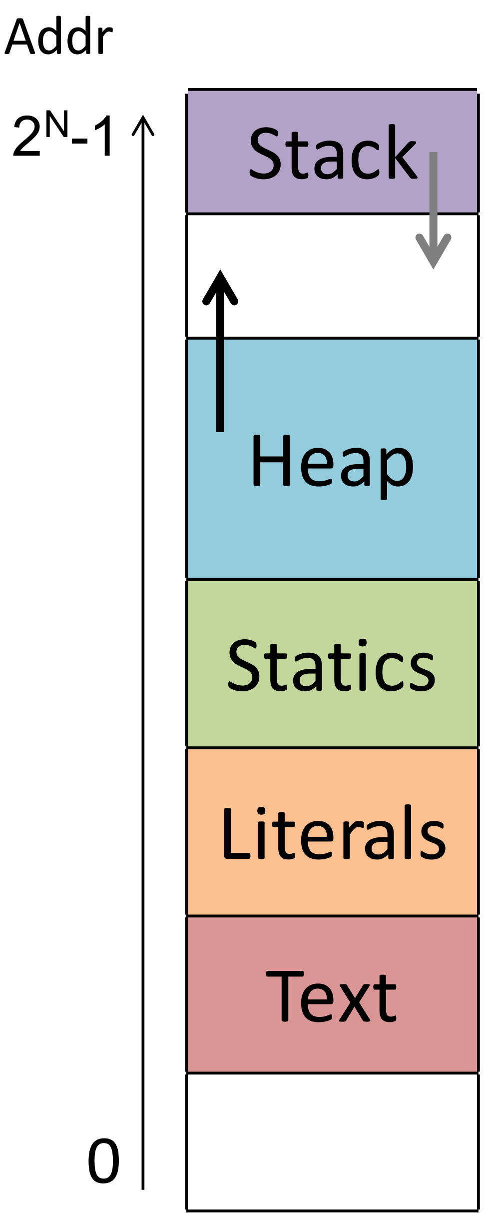
- array name == address of 0^{th} element
- array indexing == pointer arithmetic

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

- e.g.: char* is a:
  - 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

```
#include <stdlib.h>
```

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

Successful:

Returns a pointer to a memory block of at least **`size`** bytes (typically) aligned to 8-byte boundary

If **`size == 0`**, returns NULL

Unsuccessful: returns NULL and sets **`errno`**

```
void free(void* p)
```

Returns the block pointed at by **`p`** to pool of available memory

`p` must come from a previous call to **`malloc`**

```c
void foo(int n, int m) {
  // allocate a block of n ints
  int* p = (int *)malloc(n * sizeof(int));
  if (p == NULL) {
    perror("malloc");  // print an error message
    exit(0);
  }
  for (int i=0; i<n; i++) { p[i] = i; }

  free(p); // return p to available memory pool
}
```

malloc rules:

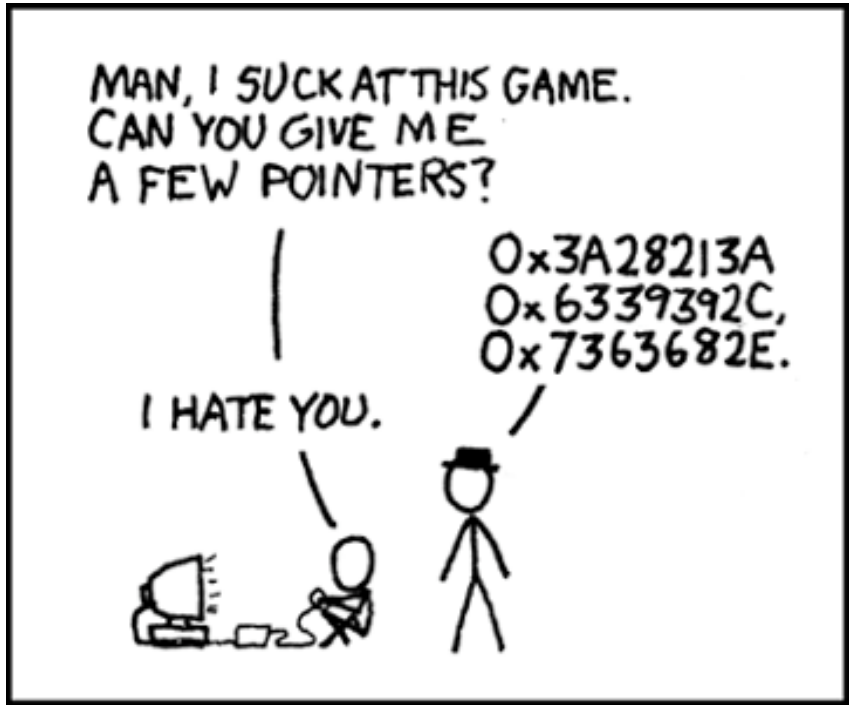
cast result to proper pointer type

Use **sizeof(...)** to determine size

free rules:

Free only objects acquired from malloc, and only once.

Do not use an object after freeing it.



http://xkcd.com/138/

C: Memory-Related Perils and Pitfalls

!!!

Terrible things to do with pointers, part 1.

Dereferencing bad pointers

See later exercises for:

- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks

C: scanf reads formatted input

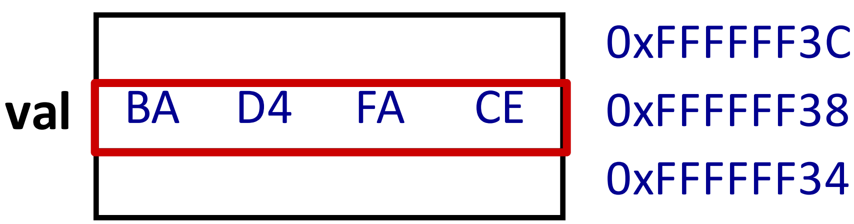
```
int val;

...

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

!!!

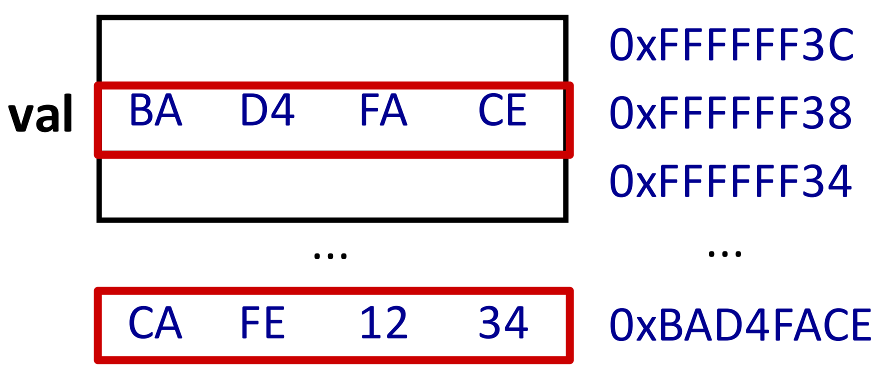
```
int val;
...
scanf("%d",  val);
```

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

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 very close to machine level.
- 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 other programming languages since then fix a lot of C's problems while keeping strengths.