Arrays & Pointers

CS 3410: Computer System Organization and Programming

Spring 2025

Administrivia

- **Assignments:**
  - **A0: Infrastructure** due tonight
    - Slip days aren't tracked
  - **A1: printf** due last night; late due date Sat. (2/1)
    - Slip days *are* tracked
  - **A0/A1 Survey** out now, due Sat.
  - **A2: Minifloat** out today!
    - Due Wed. (2/5)
- **Online Exercises (E0-E4)** due Wed. (2/5)
- **Week 2 TMQ** due Fri. (1/31)

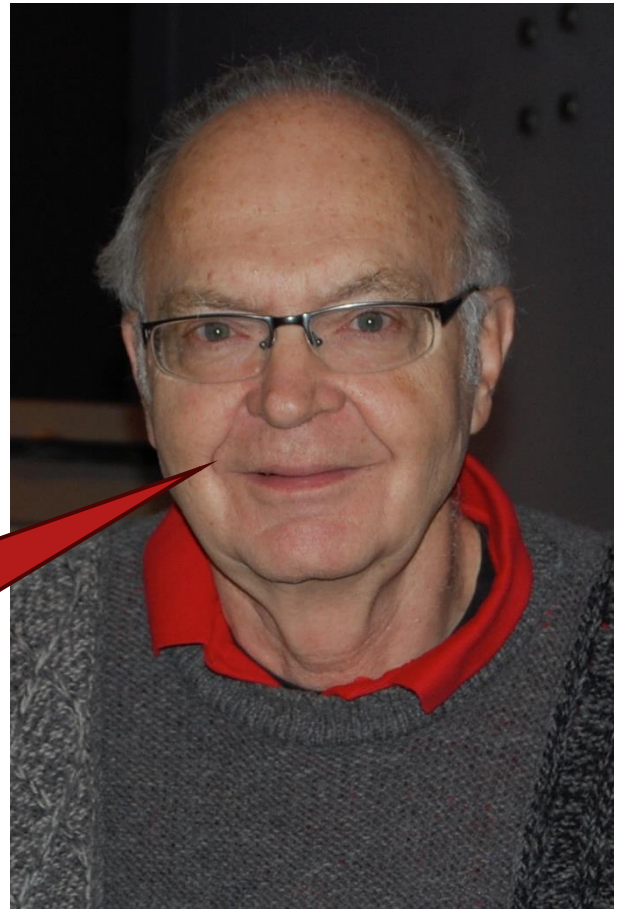
Bit Packing

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

int main() {
    uint32_t bits = 0x41040000;
    uint32_t mantissa = bits & 0x007fffff; // mask to isolate mantissa
    uint32_t exponent = (bits & 0x7f800000) >> 23; // bit and bit shift
    uint32_t sign = (bits & 80000000) >> 31; // mask and bit shift

    printf("s = %b, e = %b, g = %b \n", sign, exponent, mantissa);
    return 0;
}
```

Today's Plan

- Arrays
- **Pointers**: C's Central Construct
  - Mental model of memory
  - Pointers as addresses
  - Pointers as references
  - Pointer Arithmetic
  - Arrays as Pointers
  - Fun Pointer Tricks

Donald Knuth



I do consider assignment statements and **pointer variables** to be among computer science's "most valuable treasures".

Arrays

Arrays

- **An array is a sequence of same-type values that are consecutive in memory**
- Fixed-size
  - C does not know the size of an array!

```
// Declaration
int my_array[4];


// Declaration & Initialization
int my_array[4] = {42, 3, -19, 71};
int my_array[4] = {0};
int my_array[]  = {42, 3, -19, 71};
```

Demo: Arrays

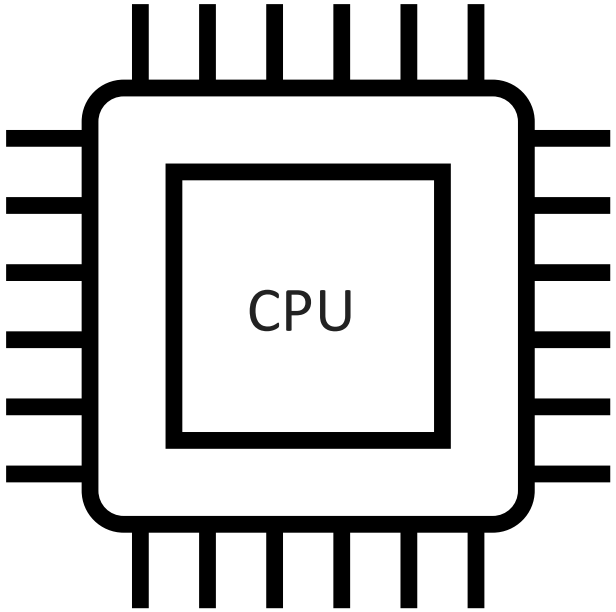
```
1  #include <stdio.h>
2
3  int main() {
4    int courses[7] = {1110, 1111, 2110,
5                      2112, 2800, 3110, 3410};
6    int course_total = 0;
7    for (int i = 0; i < 7; ++i) {
8      course_total += courses[i];
9    }
10    printf("the average course is CS %d\n",
11           course_total / 7);
12    return 0;
13 }
```

Pointers

But first, memory!

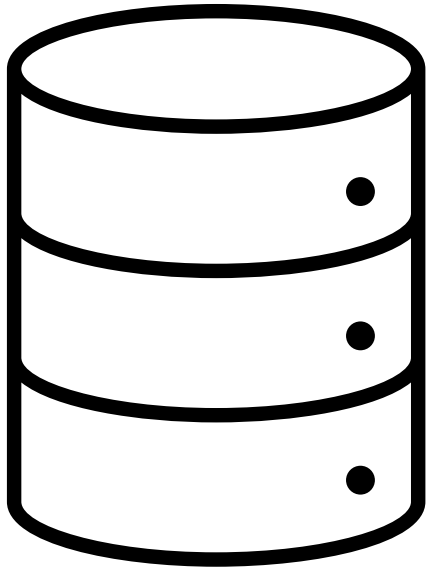
Simplified Computer Architecture

Processor



- ✓ Runs code; does computations
- ✗ Doesn't remember anything

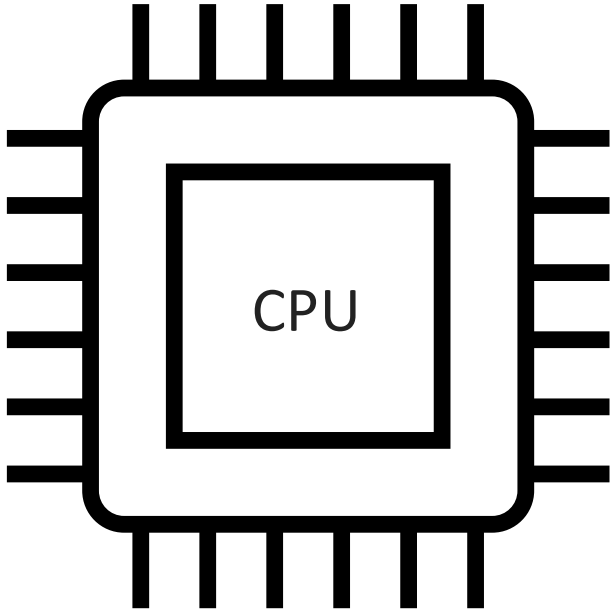
Memory



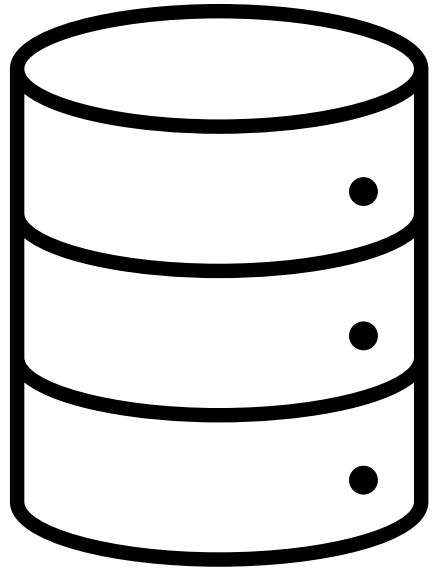
- ✗ Can't compute anything
- ✓ Stores data

A Mental Model of Memory

Processor



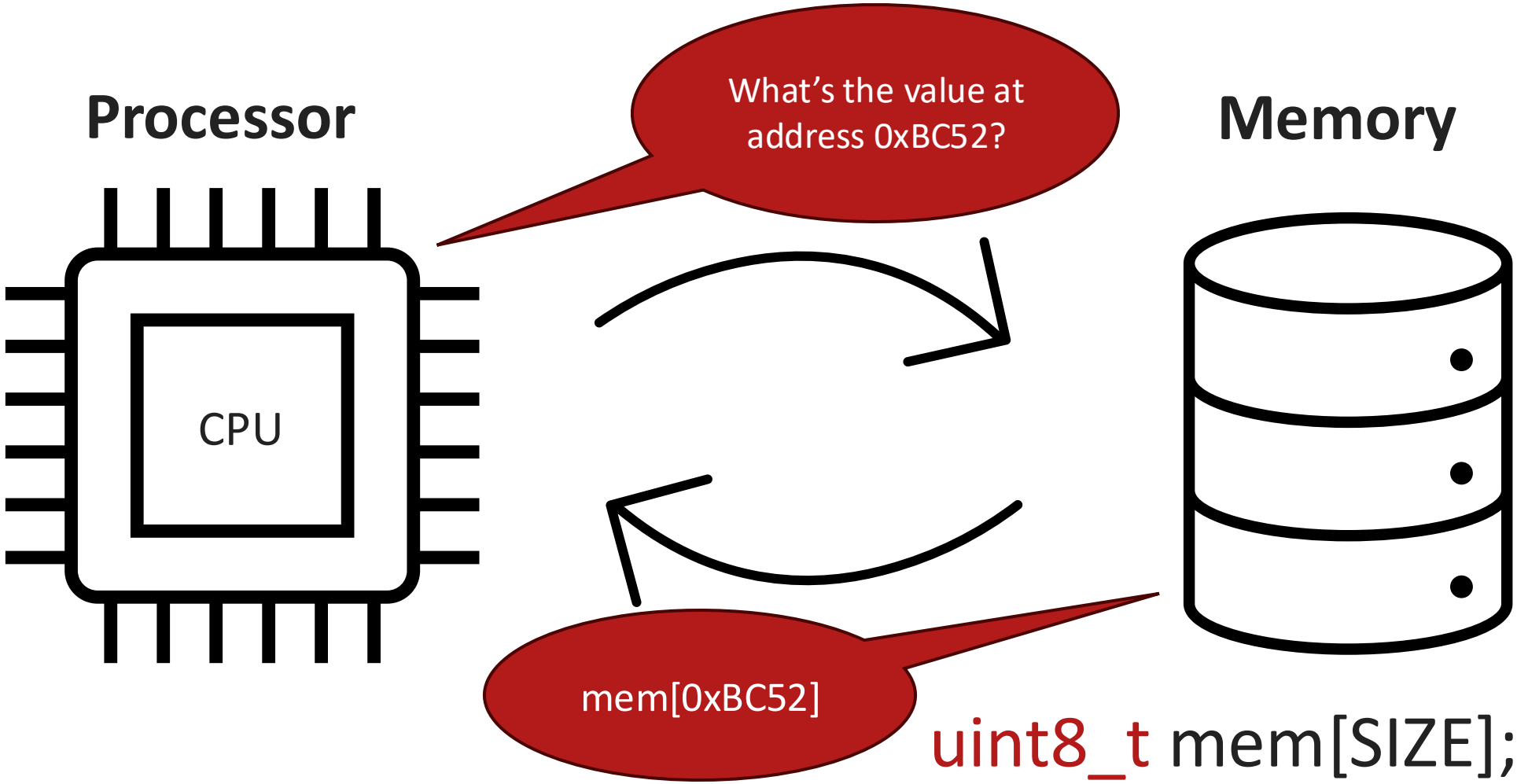
Memory



uint8_t mem[SIZE];

$$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34}$$
$$= 17{,}179{,}869{,}184\text{B}$$

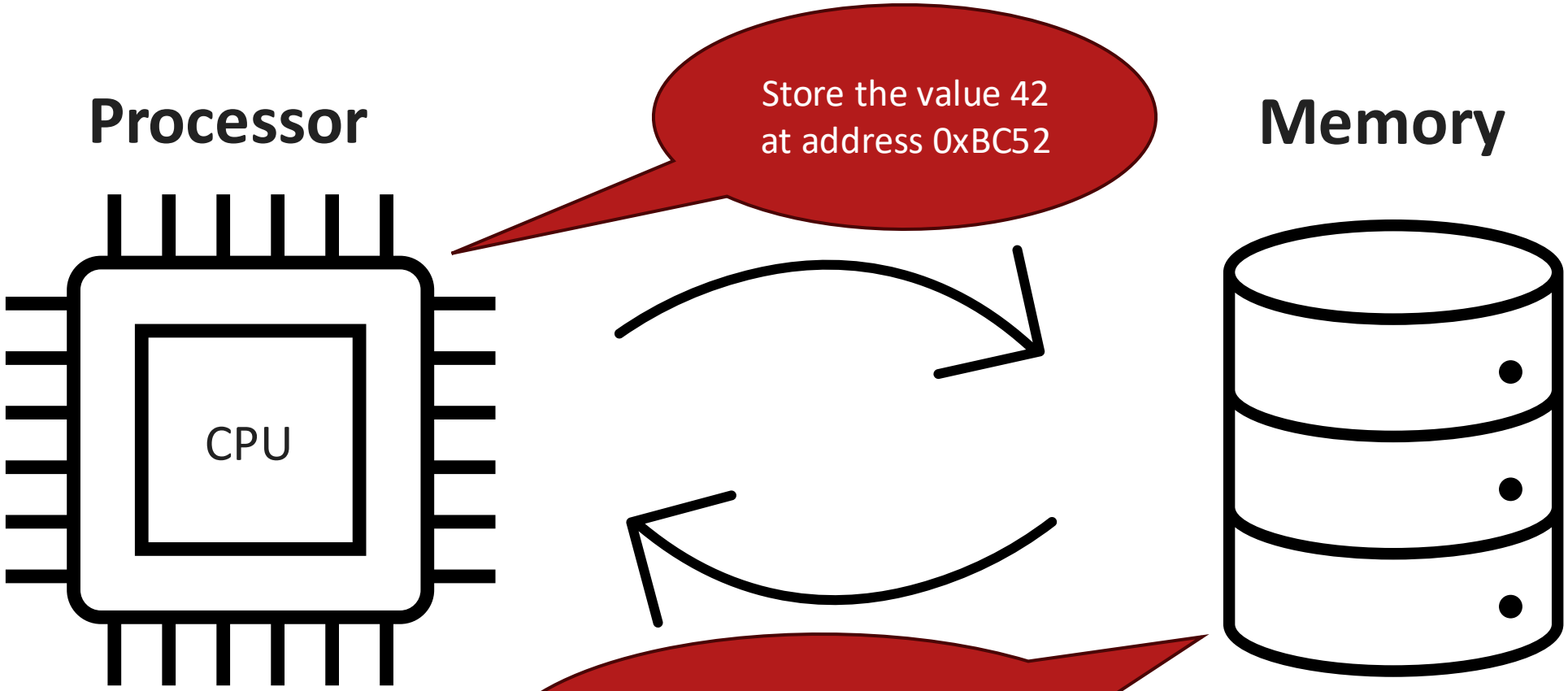
A Mental Model of Memory



uint8_t mem[SIZE];

$$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34}$$
$$= 17{,}179{,}869{,}184\text{B}$$

A Mental Model of Memory



_t mem[SIZE];

$$16\text{GB} = 16 \times 1024^3 = 2^4 \times 2^{30} = 2^{34}$$
$$= 17{,}179{,}869{,}184\text{B}$$

Loading a Single Byte

$load_1(0xBC52)$:

uint8_t mem[SIZE]

Address	Value (uint8_t)
…	…
0xBC52	0xBF
…	…
0x000F	0x02
…	
0x0003	0xEA
0x0002	0x51
0x0001	0xB2
0x0000	0x07

Loading Multiple Bytes

$load_4(0x0000)$

=

uint8_t mem[SIZE]

Address	Value (uint8_t)
…	…
0xBC52	0xBF
…	…
0x000F	0x02
…	
0x0003	0xEA
0x0002	0x51
0x0001	0xB2
0x0000	0x07

https://pollev.com/zacharysusag306

Loading Multiple Bytes

uint8_t mem[SIZE]

Address	Value (uint8_t)
…	…
0xBC52	0xBF
…	…
0x000F	0x02
…	
0x0003	0xEA
0x0002	0x51
0x0001	0xB2
0x0000	0x07

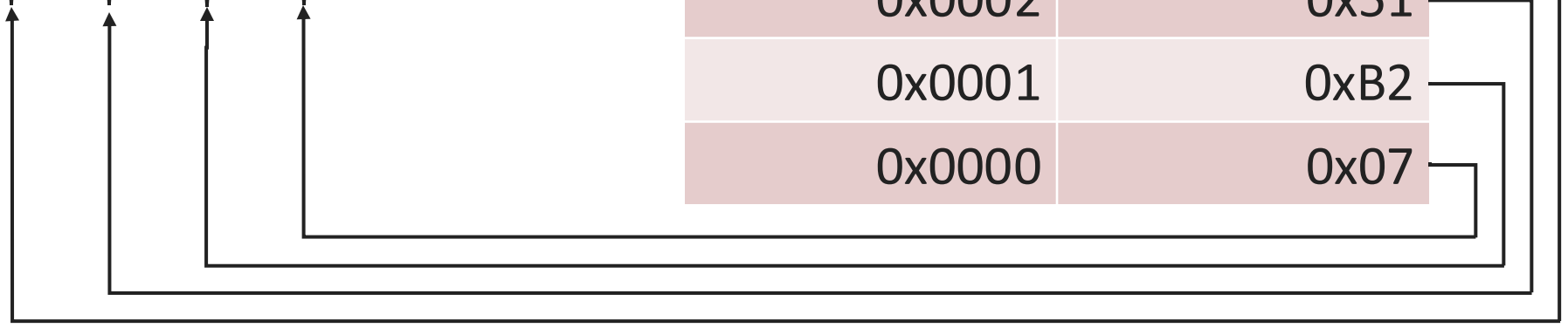
Little-Endian

***Least** significant byte at the **smallest** address*

$$\text{load}_4(0x0000)$$

$$=$$

$$0x\ \underbrace{EA}\ \underbrace{51}\ \underbrace{B2}\ \underbrace{07}$$



Loading Multiple Bytes

Big-Endian

***Most** significant byte at the **smallest** address*

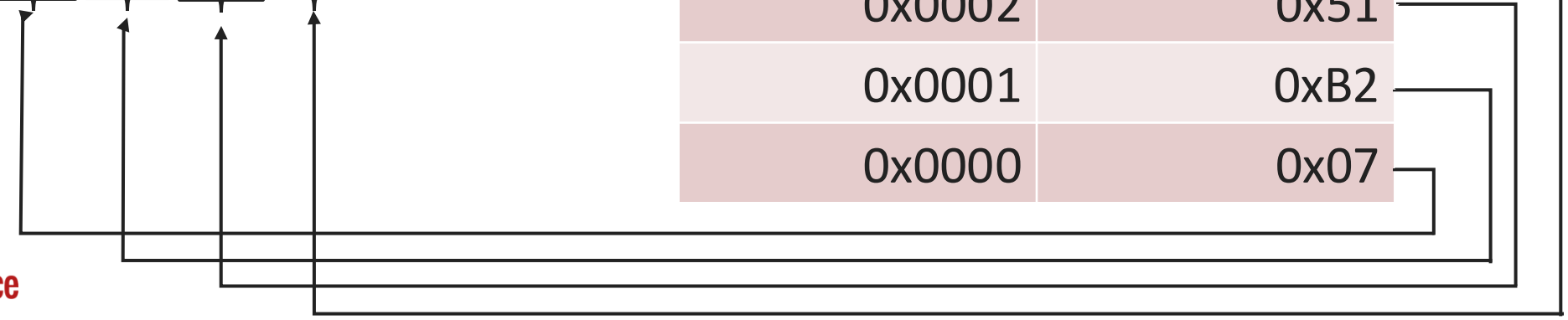
$$\text{load}_4(0x0000)$$

$$=$$

$$0x\ 07\ B2\ 51\ EA$$

uint8_t mem[SIZE]

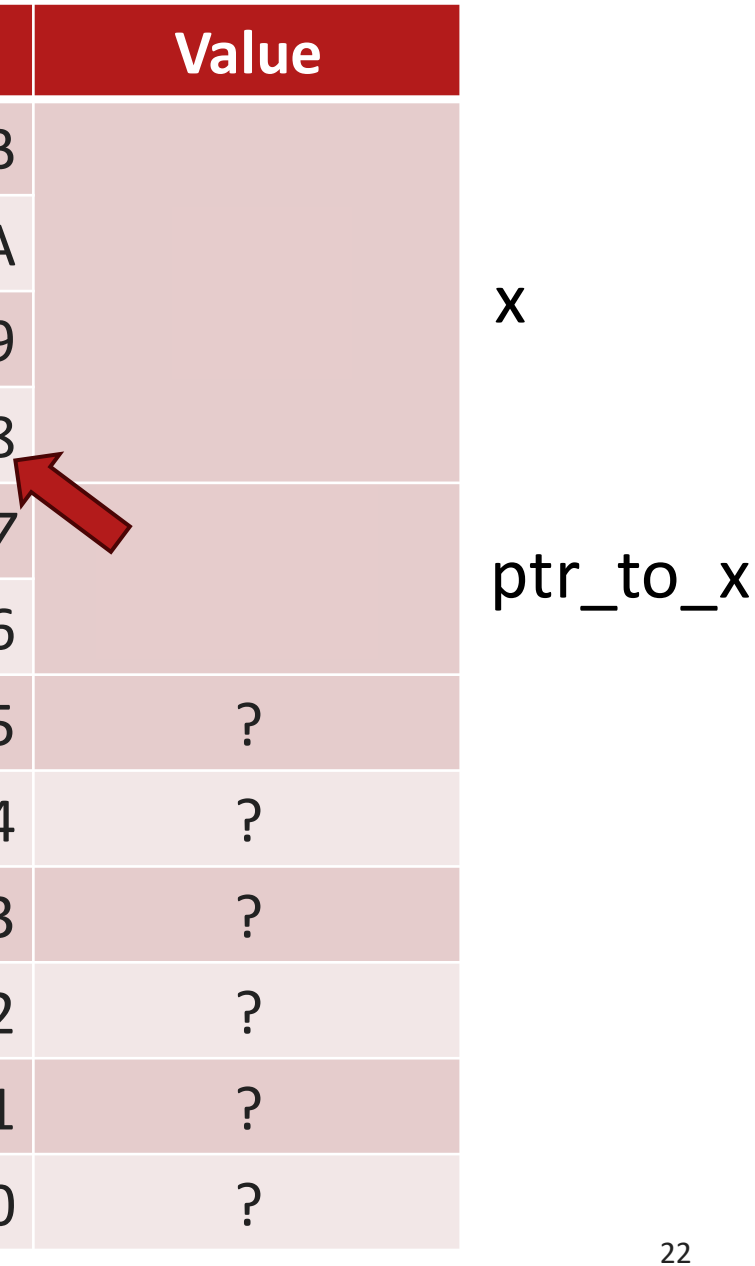
Address	Value (uint8_t)
…	…
0xBC52	0xBF
…	…
0x000F	0x02
…	
0x0003	0xEA
0x0002	0x51
0x0001	0xB2
0x0000	0x07



A Pointer is An Address

- In C, all data "lives" in memory
  - ⇒ every variable *has an address*
- & is the "reference-of" operator
  - Gets a pointer (i.e., address) to a variable

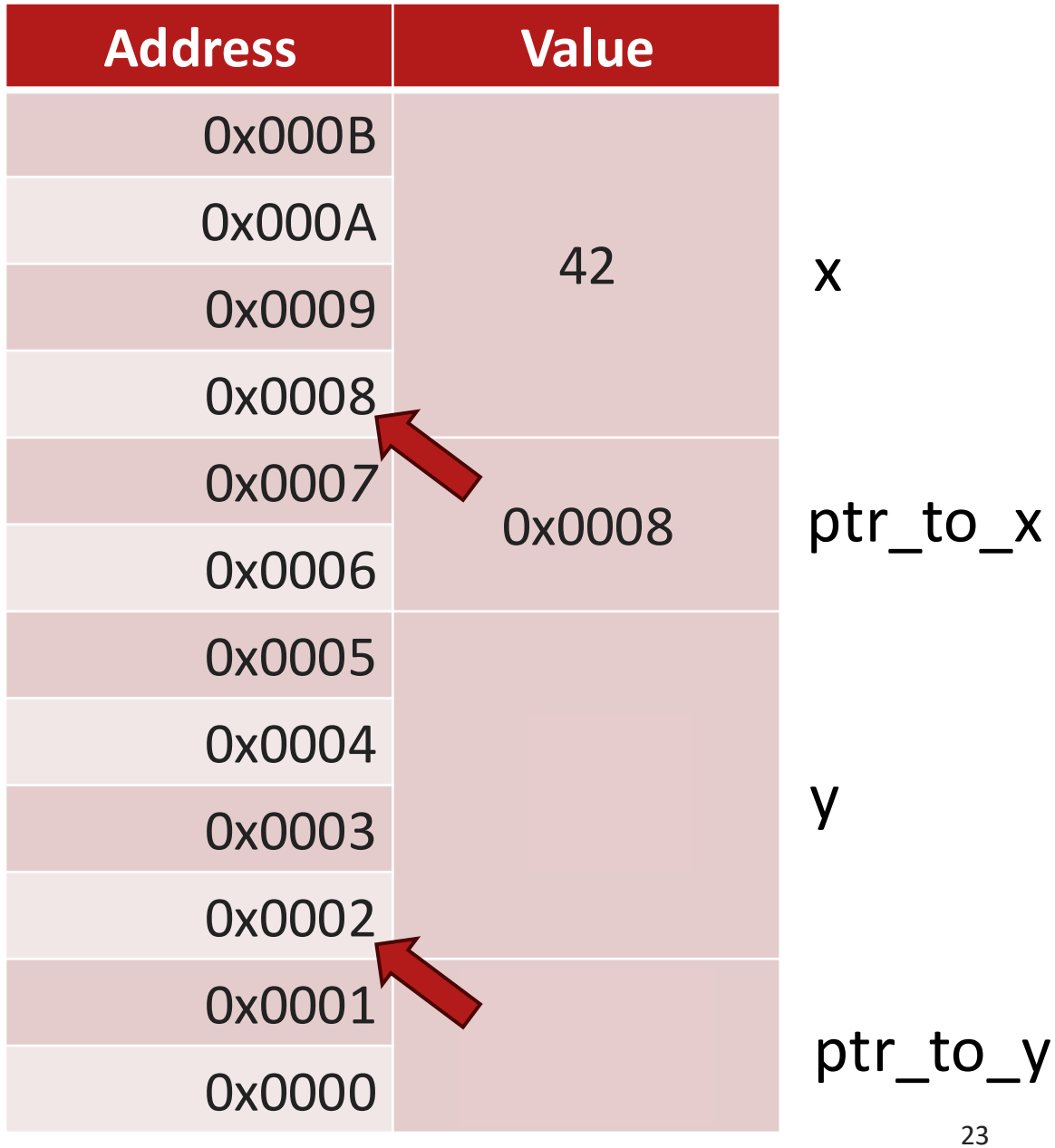
```
1  int main() {
2    int x = 42;
3    int *ptr_to_x = &x;
4    printf("x = %d is at %p\n",
5           x, ptr_to_x);
6
7
8
9
10   return 0;
11 }
```



A Pointer is An Address

- In C, all data "lives" in memory
  - ⇒ every variable *has an address*
- & is the "reference-of" operator
  - Gets a pointer (i.e., address) to a variable

```
1  int main() {
2    int x = 42;
3    int *ptr_to_x = &x;
4    printf("x = %d is at %p\n",
5           x, ptr_to_x);
6    int y = 5;
7    int *ptr_to_y = &y;
8    printf("y = %d is at %p\n",
9           y, ptr_to_y);
10    return 0;
11 }
```



Pointer Types

- **Pointers are just addresses to memory**
  - Our RISC-V 64 architecture is 64-bit
- The pointer type tells you the type of the value which it points at
  - Pointer to an integer might be int*
  - Pointer to a floating-point value might be float*
  - Pointer to a character value might be char*
- Pointer declaration is whitespace insensitive

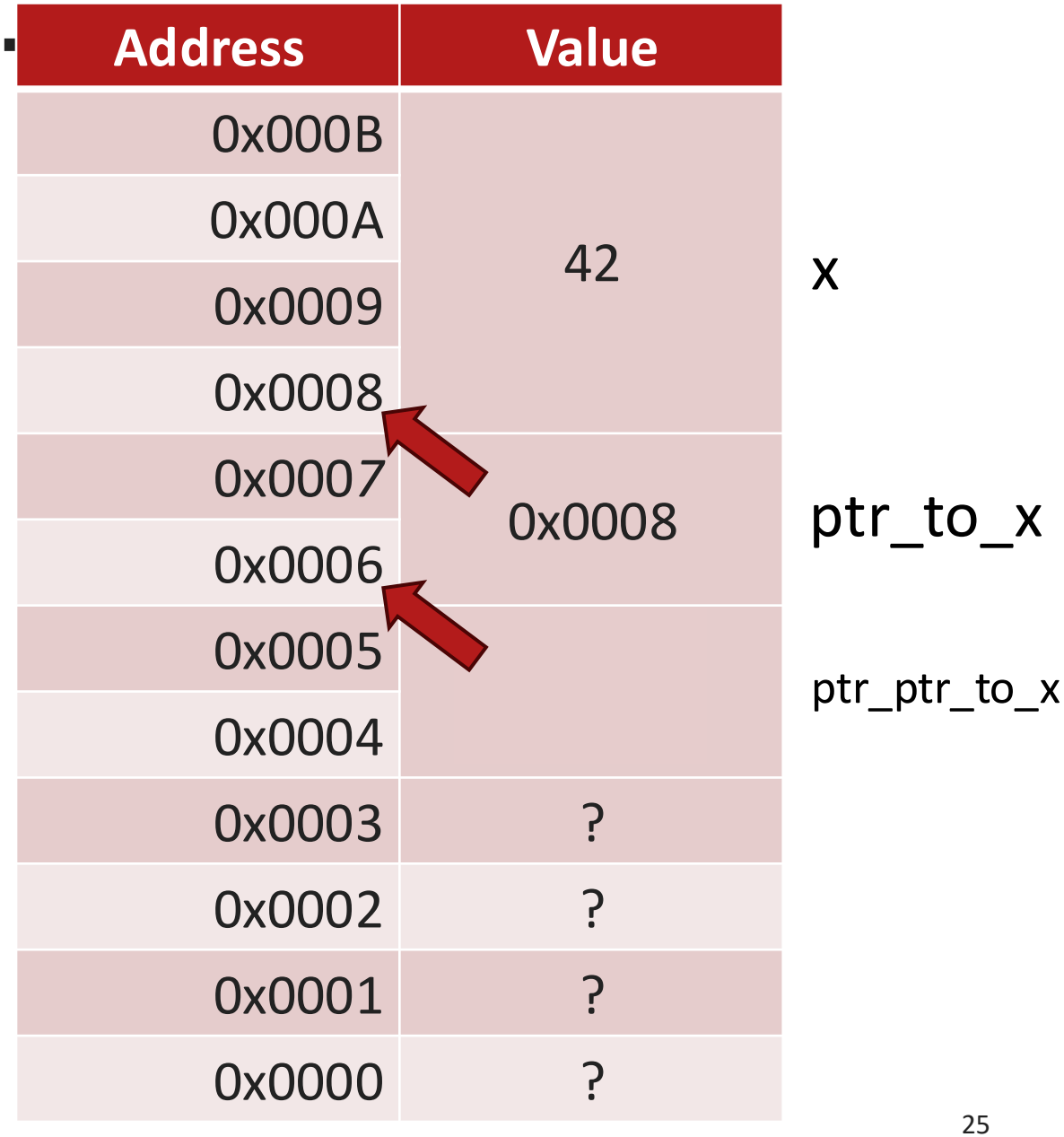
```
int* x;
int *x;
int * x;
```

} *All still pointers to an* int!

A pointer to a pointer to…

- Even pointers live in memory!

```
1  int main() {
2    int x = 42;
3    int *ptr_to_x = &x;
4    int
5
6
7
8
9
10    return 0;
11 }
```



Pointers are References

- Pointers are *useful* because they are **references**
- * is the *dereference* operator
  - Used for loading and storing

```
1  int main() {
2    int x = 42;
3    int *ptr_to_x = &x;
4    int x_copy = *ptr_to_x;
5    *ptr = 5;
6
7
8
9
10   return 0;
11 }
```

Address	Value	
0x000B	5	x
0x000A		
0x0009		
0x0008		
0x0007		ptr_to_x
0x0006		
0x0005		x_copy
0x0004		
0x0003		
0x0002		
0x0001	?	
0x0000	?	

Demo: Pointers as References

```c
#include <stdio.h>

int main() {
  int x = 34;
  int y = 10;

  int *ptr = &x;

  printf("0: x = %d and y = %d and ptr = %p\n", x, y, ptr);
   *ptr = 41;
   printf("1: x = %d and y = %d and ptr = %p\n", x, y, ptr);
   ptr = &y;
   printf("2: x = %d and y = %d and ptr = %p\n", x, y, ptr);
   *ptr = 20;
   printf("3: x = %d and y = %d and ptr = %p\n", x, y, ptr);

   return 0;
}
```

Poll Everywhere

What are the values of:

1. a
2. b
3. *p
4. *q
5. **r

```
1  int main() {
2    uint8_t a = 0;
3    uint8_t b = 1;
4    uint8_t *p = &a;
5    uint8_t *q = &b;
6    uint8_t **r = &p;
7    **r = 10;
8    *r = q;
9    *p = 11;
10    return 0;
11 }
```

https://pollev.com/zacharysusag306

Arrays as Pointers

An array is a sequence of same-type values that are consecutive in memory.

```
1  int main() {
2    int arr[3] = {42, -839, 1000};
3
4    printf("first  element is at %p\n",
5           &arr[0]);
6    printf("second element is at %p\n",
7           &arr[1]);
8    printf("third  element is at %p\n",
9           &arr[2]);
10    return 0;
11 }
```

Address	Value	
0x000B		
0x000A		arr[2]
0x0009		
0x0008		
0x0007		
0x0006		arr[1]
0x0005		
0x0004		
0x0003		
0x0002		arr[0]
0x0001		
0x0000		

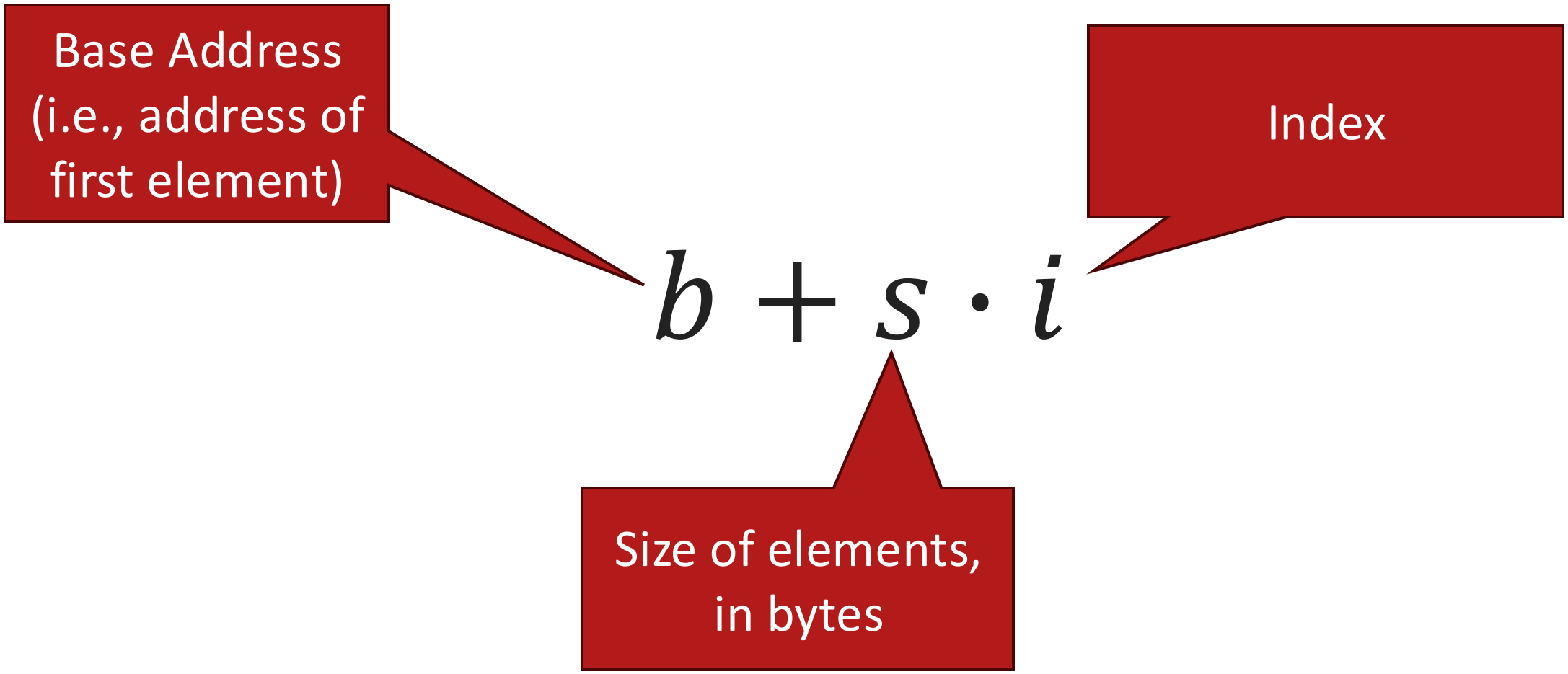
Arrays as Pointers

An array is a sequence of same-type values that are consecutive in memory.

```
1  int main() {
2    int arr[3] = {42, -839, 1000};
3
4    printf("first  element is at %p\n",
5           &arr[0]);
6    printf("second element is at %p\n",
7           &arr[1]);
8    printf("third  element is at %p\n",
9           &arr[2]);
10    return 0;
11 }
```

Address	Value	
0x000B	1000	arr[2]
0x000A		
0x0009		
0x0008		
0x0007	-839	arr[1]
0x0006		
0x0005		
0x0004		
0x0003	42	arr[0]
0x0002		
0x0001		
0x0000		

Formula for address of an element at index i



Arrays as Pointers to the First Element

```
1  #include <stdio.h>
2
3  int main() {
4    int courses[7] = {1110, 1111, 2110, 2112, 2800, 3110, 3410};
5
6    printf("first element is at %p\n", &courses[0]);
7    printf("the array itself is %p\n", courses);
8
9    return 0;
10 }
```

courses and &courses[0] point to the same address!

Passing Arrays to Functions

```
1  int sum_n(int *vals, int count) {
2    int total = 0;
3    for (int i = 0; i < count; ++i) {
4      total += vals[i];
5    }
6    return total;
7  }
8  int main() {
9    int courses[7] = {1110, 1111, 2110, 2112, 2800, 3110, 3410};
10   int sum = sum_n(courses, 7);
11   printf("the average course is CS %d\n",
12          sum / 7);
13   return 0;
14 }
```

- C does not store the length of an array!
  - You must pass the length alongside the array

Pointer Arithmetic

Question:
Can we compute addresses ourselves?

```
1  void experiment(int* courses) {
2    printf("courses     = %p\n", courses);
3    printf("courses + 1 = %p\n", courses + 1);
4  }
5
6  int main() {
7    int courses[7] = {1110, 1111, 2110, 2112, 2800, 3110, 3410};
8    experiment(courses);
9    return 0;
10 }
```

```
$ ./a.out
courses     = 0x1555d56bb0
courses + 1 = 0x1555d56bb4
```

Pointer Arithmetic Rule

- In C, pointer arithmetic "moves" pointers by *element-sized chunks*
  - Element size is determined by pointer type
- courses has type int*
  - Element size is 4 bytes
- **Example:**
  - courses + n adds $4 \times n$ bytes to address of courses

Dereferencing Elements of an Array

```
1  void experiment(int* courses) {
2    printf("courses[0] = %d\n", *(courses + 0));
3    printf("courses[5] = %d\n", *(courses + 5));
4  }
5
6  int main() {
7    int courses[7] = {1110, 1111, 2110, 2112, 2800, 3110, 3410};
8    experiment(courses);
9    return 0;
10 }
```

```
$ ./a.out
courses[0] = 1110
courses[5] = 3110
```

Strings

Strings are Null-Terminated Character Arrays

- Recall that we told you a string has type char* in C
  - Strings are arrays of char values
  - A char is generally 1-byte (8-bits)
- Strings keep track of length by ending with a *null character* (‘\0’)
  - All strings *should* end with a *null character*
- **Example:**
  - “CS3410” = { 'C', 'S', '3', '4', '1', '0', '\0’ }
  - “CS3410” has length 7, not 6!

Demo: Strings

```
1  void print_line(char *s) {
2    for (int i = 0; s[i] != '\0'; ++i)
3    {
4      fputc(s[i], stdout);
5    }
6    fputc('\n', stdout);
7  }
8
9  int main() {
10     char message[7] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
11     print_line(message);
12     return 0;
13   }
```

Fun Pointer Tricks

Pass by Reference

```
1  #include <stdio.h>
2
3  void swap(int x, int y) {
4    int tmp = x;
5    x = y;
6    y = tmp;
7  }
8
9  int main() {
10   int a = 34;
11   int b = 10;
12   printf("a: %d; b: %d\n", a, b);
13   swap(a, b);
14   printf("a: %d; b: %d\n", a, b);
15 }
```

https://pollev.com/zacharysusag306

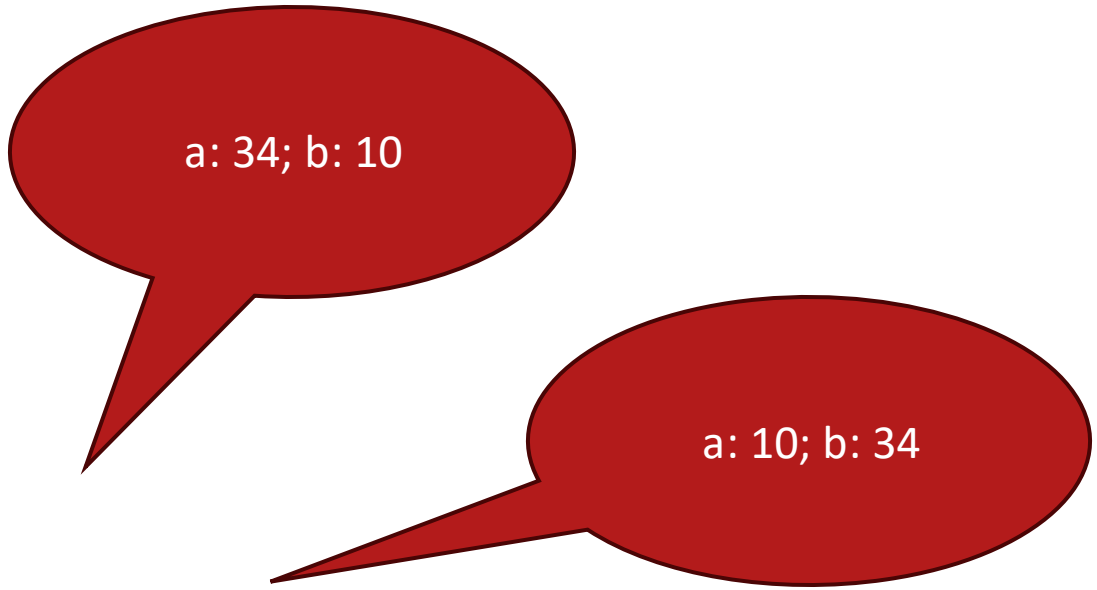
What does the `printf` statement on line 14 print out?

a: 34; b: 10

a: 10; b: 34

Pass by Reference

```
1  #include <stdio.h>
2
3  void swap(int* x, int* y) {
4    int tmp = *x;
5    *x = *y;
6    *y = tmp;
7  }
8
9  int main() {
10   int a = 34;
11   int b = 10;
12   printf("a: %d; b: %d\n", a, b);
13   swap(&a, &b);
14   printf("a: %d; b: %d\n", a, b);
15 }
```



Null Pointers

- Pointers are just integers (i.e., bits!), so what does 0 mean?
- NULL is a pointer with value 0
  - Often used to signal failure
- Be Careful!
  - **NEVER dereference NULL**
  - When in doubt, always check!

Pointers to Anything

- Pointers are just bits!
  - No difference between int*, float*, and char*
- void* is a "pointer to something"

```
1  #include <stdio.h>
2
3  void print_ptr(void* p) {
4    printf("%p\n", p);
5  }
6
7  int main() {
8    int x = 34;
9    float y = 10.0f;
10   print_ptr(&x);
11   print_ptr(&y);
12 }
```