CS 31: Introduction to Computer Systems

15 Storage

03-20-2025

SWARTHMORE COLLEGE

Today

- Accessing *things* via an offset
  - Arrays, Structs, Unions
  - Connect accessing them in C with what we know about assembly

- How complex structures are stored in memory
  - Multi-dimensional arrays & Structs

So Far: One Dimensional Arrays

- We are not restricted to an array of ints..
  How about an array *of* arrays of ints?

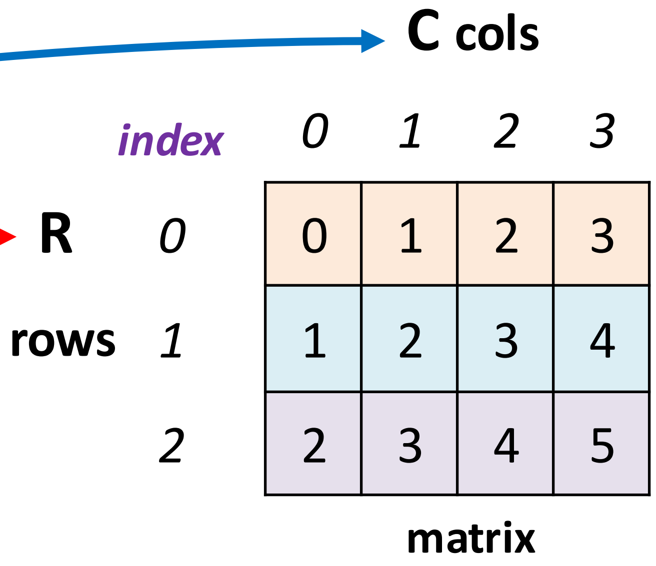
“Give me three sets of four integers”

```
int twodims[3][4];
```

- How should these be organized in memory?

Declaring Static 2D Arrays

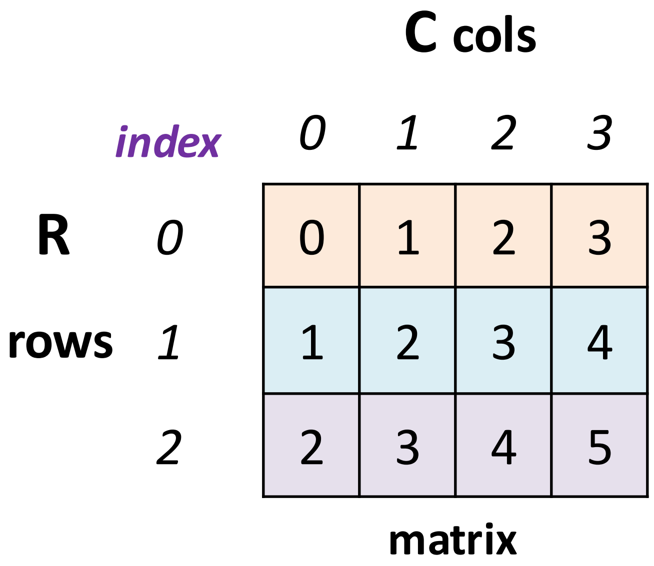
```
#define R 3
#define C 4

int  matrix[R][C] , i, j;

for(i=0; i<R; i++) {
  for(j=0; j<C; j++) {
     matrix[i][j] = i+j;
   }
}
```



index	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5

- Declare with row and column dimension
- Can use matrix[i][j] to index

Memory Layout of Static 2D Arrays



C cols

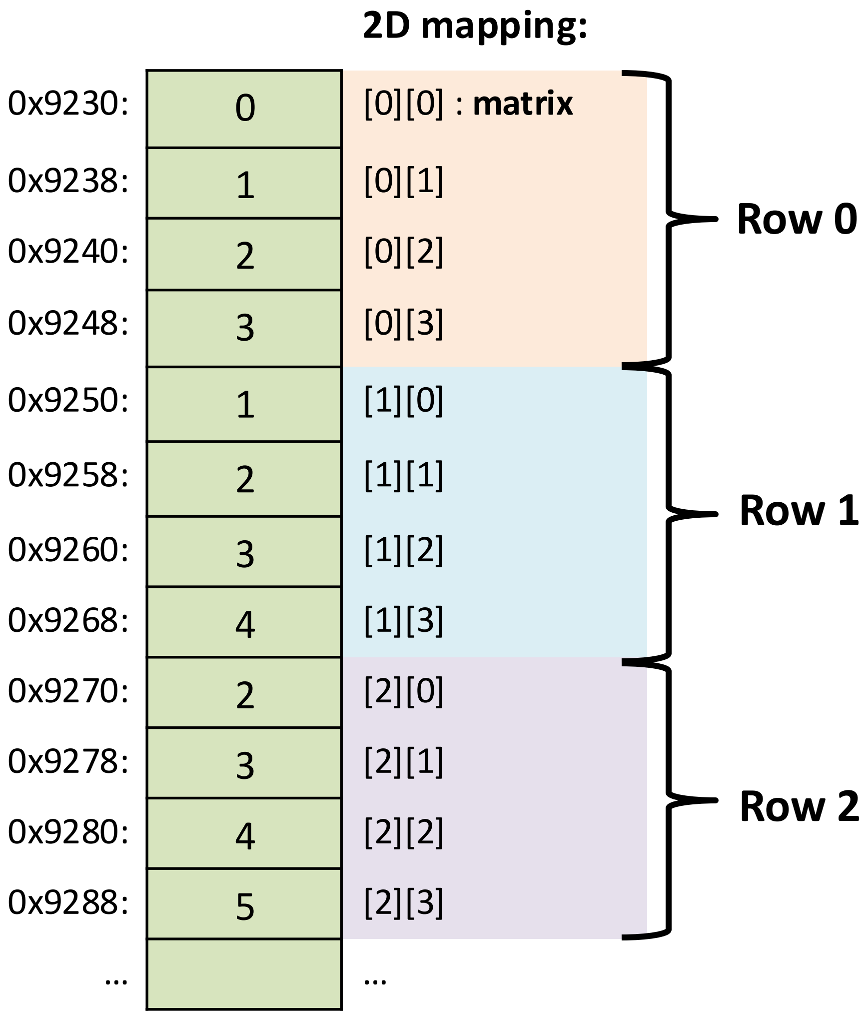
R rows

index	*0*	*1*	*2*	*3*
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5

matrix

***Row Major* Order in C:**

all Row 0 buckets, followed by
all Row 1 buckets, followed by
all Row 2 buckets, ...



		2D mapping:	
0x9230:	0	[0][0] : **matrix**	**Row 0**
0x9238:	1	[0][1]	
0x9240:	2	[0][2]	
0x9248:	3	[0][3]	
0x9250:	1	[1][0]	**Row 1**
0x9258:	2	[1][1]	
0x9260:	3	[1][2]	
0x9268:	4	[1][3]	
0x9270:	2	[2][0]	**Row 2**
0x9278:	3	[2][1]	
0x9280:	4	[2][2]	
0x9288:	5	[2][3]	
...		...	

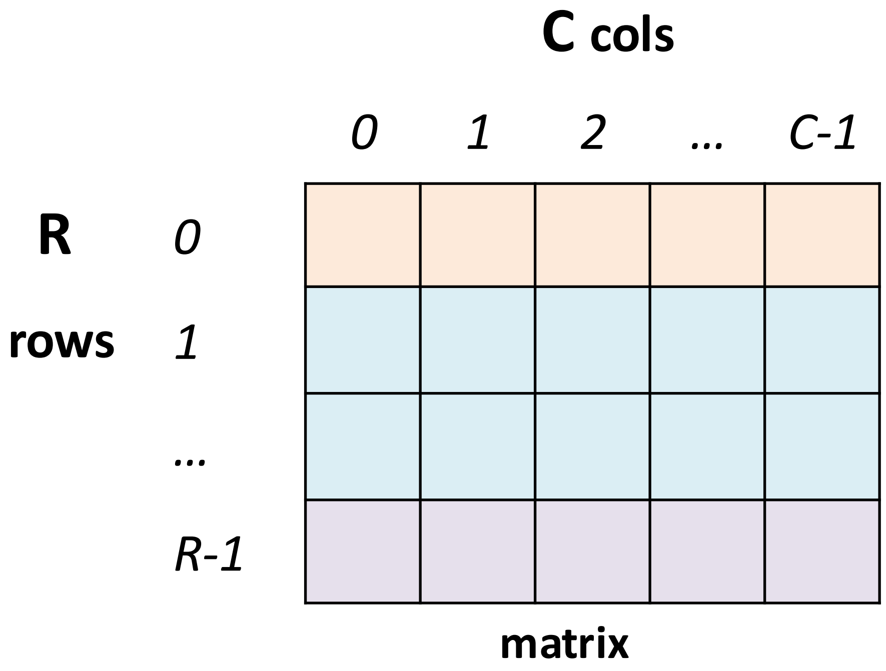
Using Static 2D Arrays as Parameters

- 2D array parameter must specify **column dimension**
  - **Why?** Compiler needs the column dimension to calculate offset from base address in memory of bucket [i][j]
- Row dimension passed as 2nd parameter to make function *more generic*
  - function can be passed any 2D array with same column dimension

```
void foo(int matrix[][C], int rows){

   int i, j;

   for(i=0; i < rows; i++) {
    for(j=0; j< C; j++) {
       matrix[i][j] = i*j;
    }
   }
}
```

```
#define R  3
#define C  4

int main() {

  int arr[R][C];
  int grid[100][C];

  foo(arr,  R);
  foo(grid, 100);
```

Calculating Offset for Static 2D Arrays

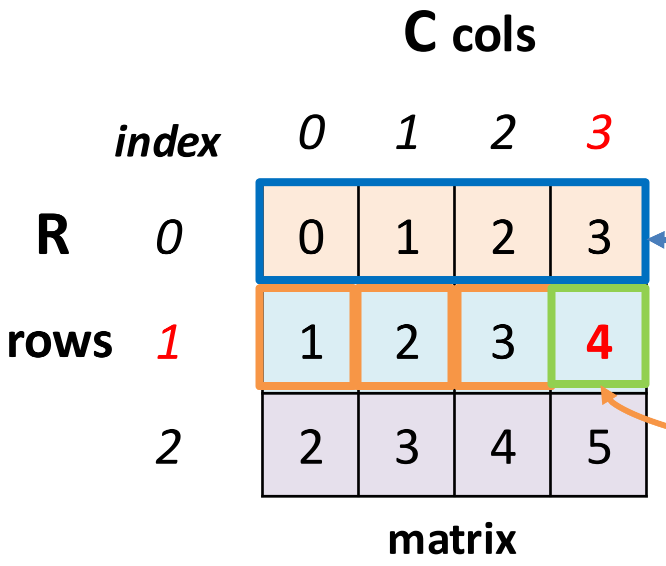


Offset of `matrix[row][col]` from base?

= row * MAX_COL + col

TIP: MAX_COL = how big each row is = max number of columns!

Calculating Offset for Static 2D Arrays



C cols

	index	*0*	*1*	*2*	*3*
R	*0*	0	1	2	3
rows	*1*	1	2	3	**4**
	2	2	3	4	5

matrix

Offset of `matrix[row][col]` from base?

= row * MAX_COL + col

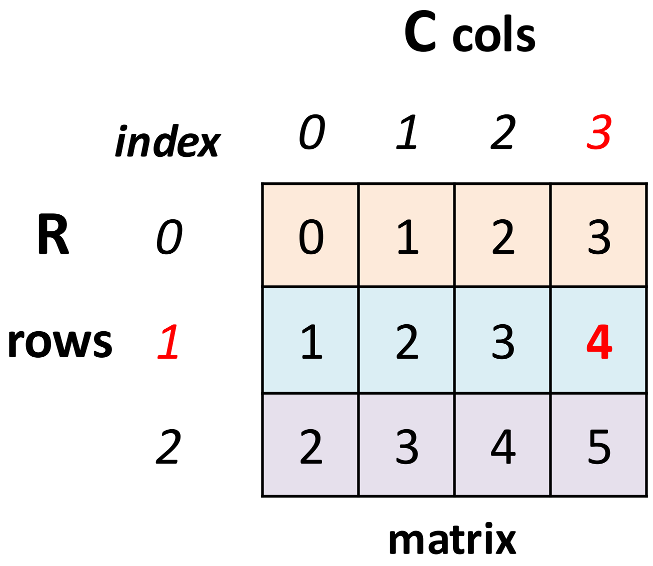
E.g., location of `matrix[1][3]`?

= base + (1 * MAX_COL + 3) buckets *// skip 1 full row and 3 buckets*

= base + (1 * 4 + 3) buckets

= base + 7 buckets *// skip 7 buckets*

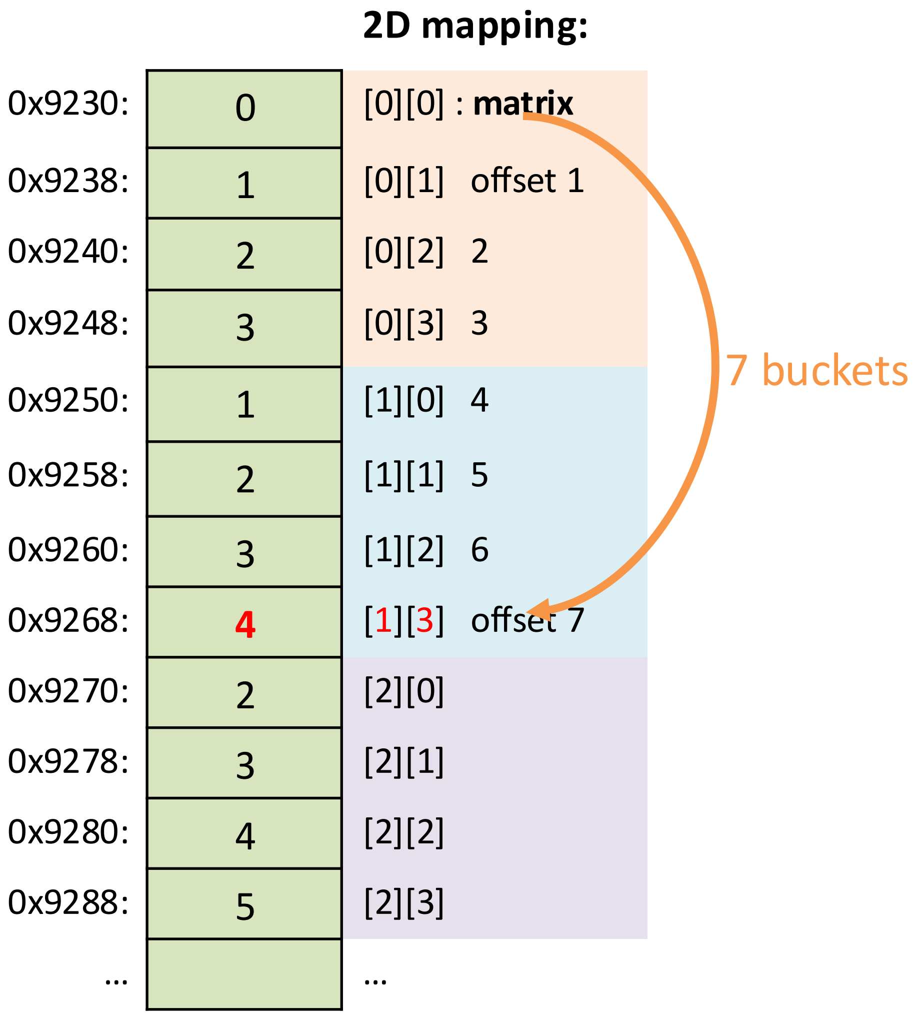
Calculating Offset for Static 2D Arrays



C cols

	index	*0*	*1*	*2*	*3*
R	*0*	0	1	2	3
rows	*1*	1	2	3	**4**
	2	2	3	4	5

matrix



		2D mapping:
0x9230:	0	[0][0] : **matrix**
0x9238:	1	[0][1] offset 1
0x9240:	2	[0][2] 2
0x9248:	3	[0][3] 3
0x9250:	1	[1][0] 4
0x9258:	2	[1][1] 5
0x9260:	3	[1][2] 6
0x9268:	**4**	[1][3] offset 7
0x9270:	2	[2][0]
0x9278:	3	[2][1]
0x9280:	4	[2][2]
0x9288:	5	[2][3]
…		…

Offset of `matrix[`**`row`**`][`**`col`**`]` from base?

= row * MAX_COL + col

E.g., location of `matrix[`**`1`**`][`**`3`**`]`?

= base + (1 * MAX_COL + 3) buckets

= base + (1 * 4 + 3) buckets

= base + 7 buckets

Calculating Address for Static 2D Arrays

C cols

R rows *index*	*0*	*1*	*2*	*3*
0	0	1	2	3
1	1	2	3	**4**
2	2	3	4	5

SIZE

Address of `matrix[`**`row`**`][`**`col`**`]` from base?

= base address + row * MAX_COL*SIZE + col*SIZE

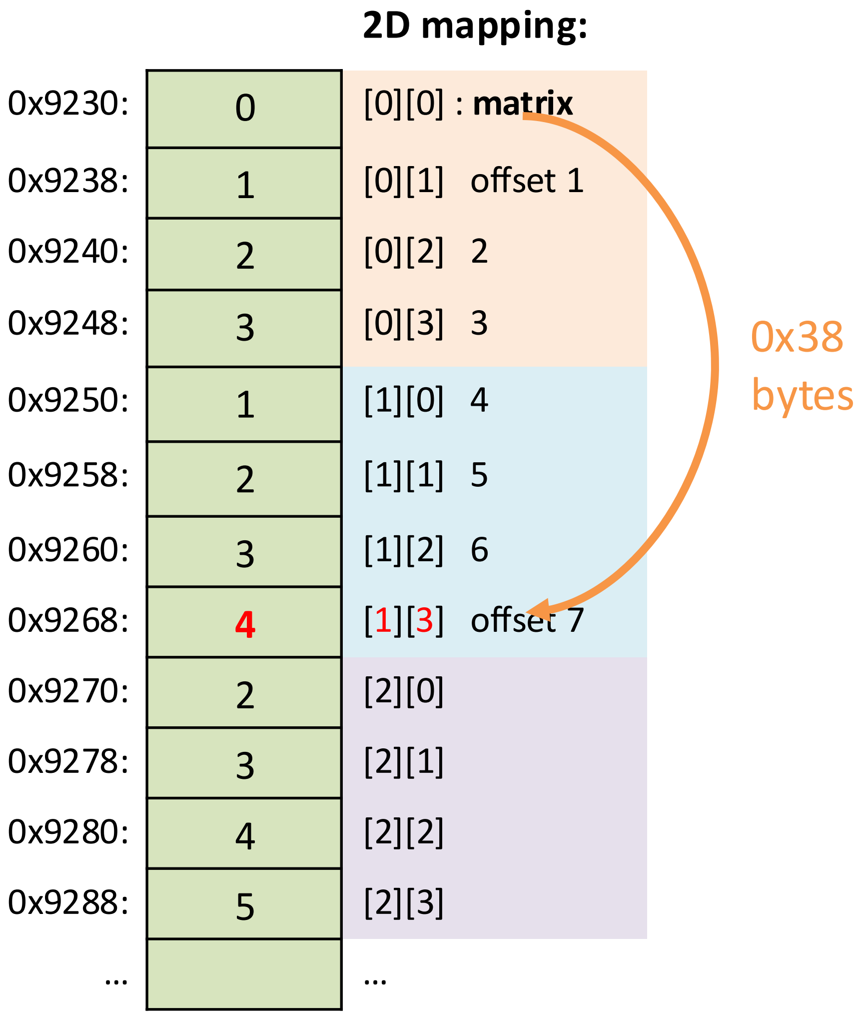
E.g., address of `matrix[`**`1`**`][`**`3`**`]`? Assume SIZE of bucket is 8 bytes

= base addr. + (1 * MAX_COL *SIZE + 3*SIZE) bytes

= base addr. + (1 * 4 * 8 + 3 * 8) bytes

= base addr. + (32 + 24) bytes

= base addr. + 0x38 → 0x9320 + 0x38 = 0x9268



2D mapping:

0x9230:	0	[0][0] : **matrix**
0x9238:	1	[0][1] offset 1
0x9240:	2	[0][2] 2
0x9248:	3	[0][3] 3
0x9250:	1	[1][0] 4
0x9258:	2	[1][1] 5
0x9260:	3	[1][2] 6
0x9268:	**4**	[1][3] offset 7
0x9270:	2	[2][0]
0x9278:	3	[2][1]
0x9280:	4	[2][2]
0x9288:	5	[2][3]
…		…

If we declared `long int matrix[5][3];`, and the base of matrix is `0x3420`, what is the address of `matrix[3][2]`? Assume `sizeof(long int) = 8` bytes.

A. `0x3488`
B. `0x3470`
C. `0x3478`
D. `0x344C`
E. None of these

*address = base address + row * MAX_COL *SIZE + col*SIZE*

If we declared `long int matrix[5][3];`, and the base of matrix is `0x3420`, what is the address of `matrix[3][2]`? Assume `sizeof(long int) = 8` bytes.

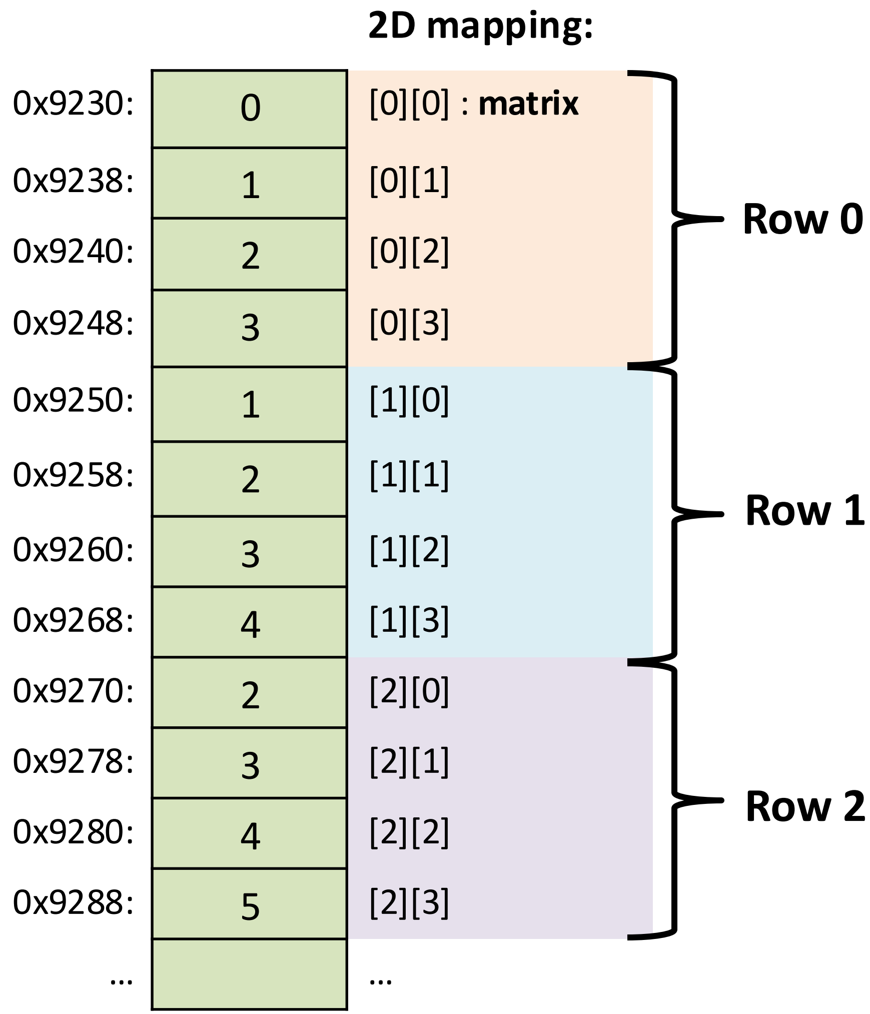
A. 0x3488
B. 0x3470
C. 0x3478
D. 0x344C
E. None of these

*address = base address + row * MAX_COL *SIZE + col*SIZE*

Dynamically Allocating 2D Arrays: Contiguous Memory

- Given the *row-major order* layout, a "two-dimensional array" is still just a **contiguous block** of memory:

  The malloc function just needs to return… a pointer to a contiguous block of memory! That is, you only need **one call** to malloc.



		2D mapping:	
0x9230:	0	[0][0] : **matrix**	**Row 0**
0x9238:	1	[0][1]	
0x9240:	2	[0][2]	
0x9248:	3	[0][3]	
0x9250:	1	[1][0]	**Row 1**
0x9258:	2	[1][1]	
0x9260:	3	[1][2]	
0x9268:	4	[1][3]	
0x9270:	2	[2][0]	**Row 2**
0x9278:	3	[2][1]	
0x9280:	4	[2][2]	
0x9288:	5	[2][3]	
…		…	

Dynamically Allocating 2D Arrays: Contiguous Memory

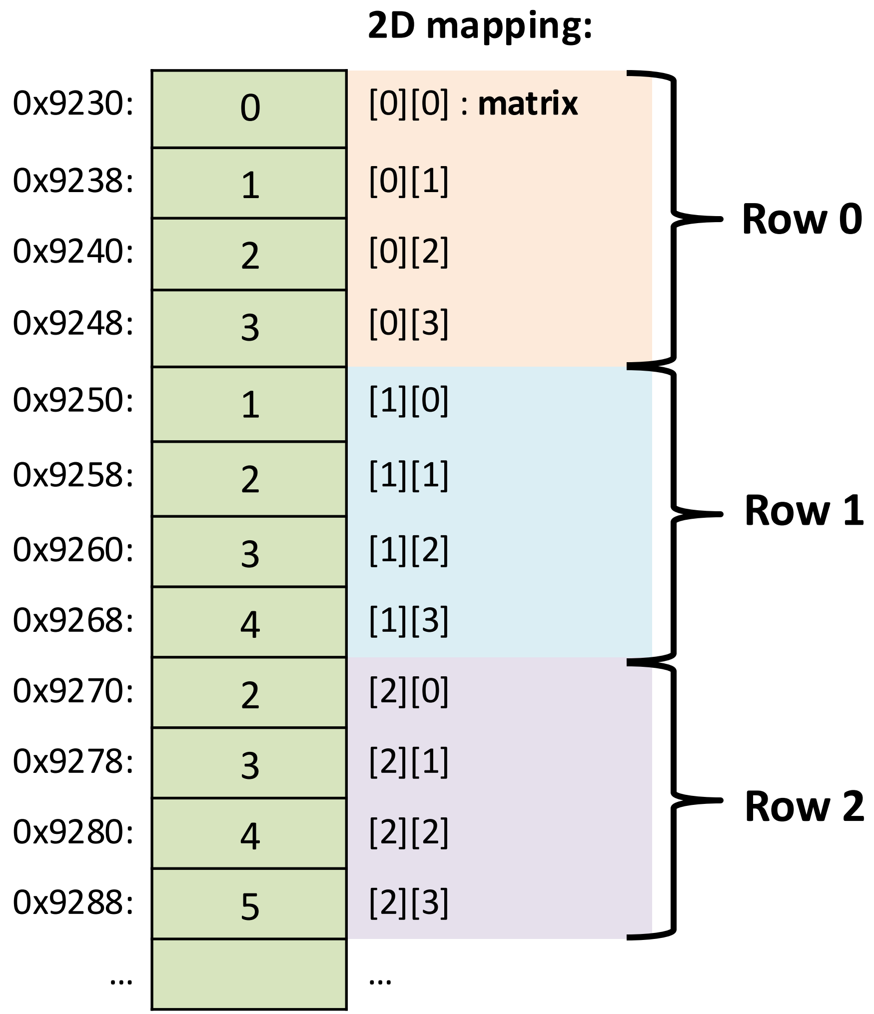
For this example, with three rows and four columns:

	C cols			
R rows	*0*	*1*	*2*	*3*
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5

```
long int * matrix = malloc(3 * 4 * sizeof (long int));
```

Caveat: the C compiler doesn't know that you're planning to use this block of memory with more than one index (i.e., row and column).

Can't access: `matrix[i][j]`!



		2D mapping:	
0x9230:	0	[0][0] : **matrix**	**Row 0**
0x9238:	1	[0][1]	
0x9240:	2	[0][2]	
0x9248:	3	[0][3]	
0x9250:	1	[1][0]	**Row 1**
0x9258:	2	[1][1]	
0x9260:	3	[1][2]	
0x9268:	4	[1][3]	
0x9270:	2	[2][0]	**Row 2**
0x9278:	3	[2][1]	
0x9280:	4	[2][2]	
0x9288:	5	[2][3]	
…		…	

Dynamically Allocating 2D Arrays: Contiguous Memory

For this example, with three rows and four columns:

C cols

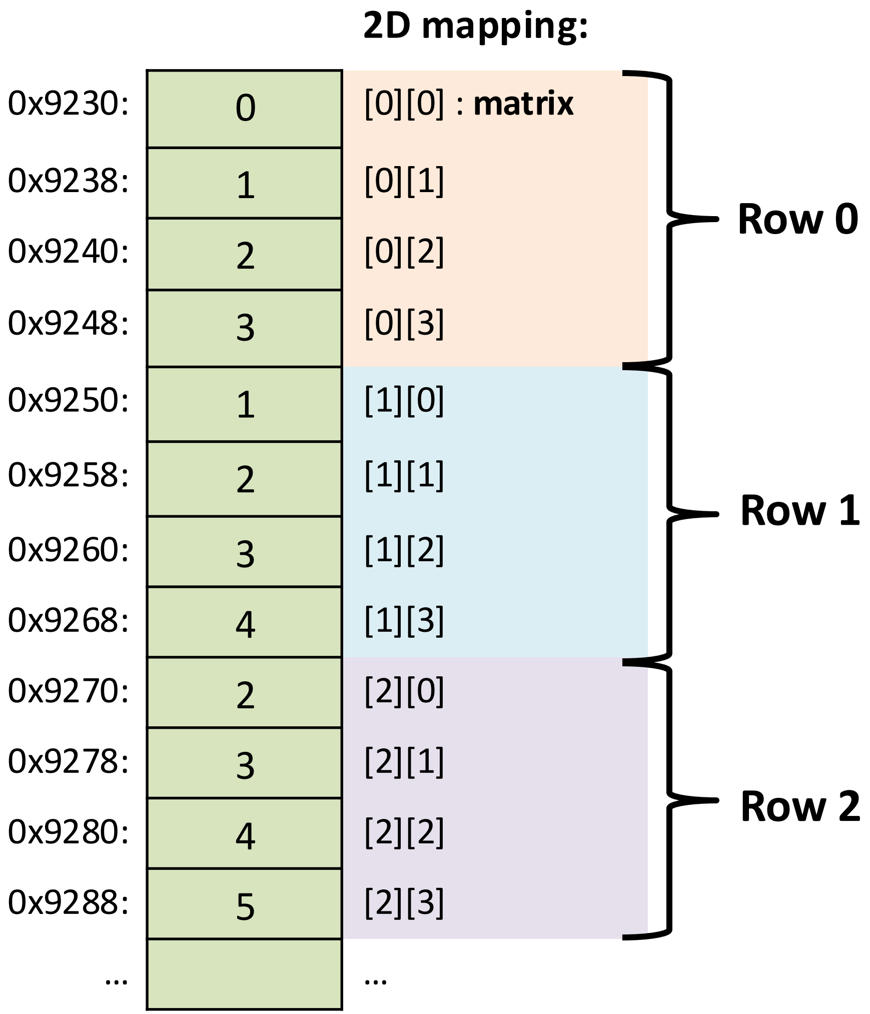
R rows	*0*	*1*	*2*	*3*
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5

```
long int * matrix = malloc(3 * 4 * sizeof (long int));
```

To access `matrix[i][j]`, compute the offset manually:

```
index = i * COL_MAX + j;
```

```
matrix[index] = …
```



		2D mapping:	
0x9230:	0	[0][0] : **matrix**	**Row 0**
0x9238:	1	[0][1]	
0x9240:	2	[0][2]	
0x9248:	3	[0][3]	
0x9250:	1	[1][0]	**Row 1**
0x9258:	2	[1][1]	
0x9260:	3	[1][2]	
0x9268:	4	[1][3]	
0x9270:	2	[2][0]	**Row 2**
0x9278:	3	[2][1]	
0x9280:	4	[2][2]	
0x9288:	5	[2][3]	
…		…	

Using Dynamically Allocated 2D Arrays as Parameters

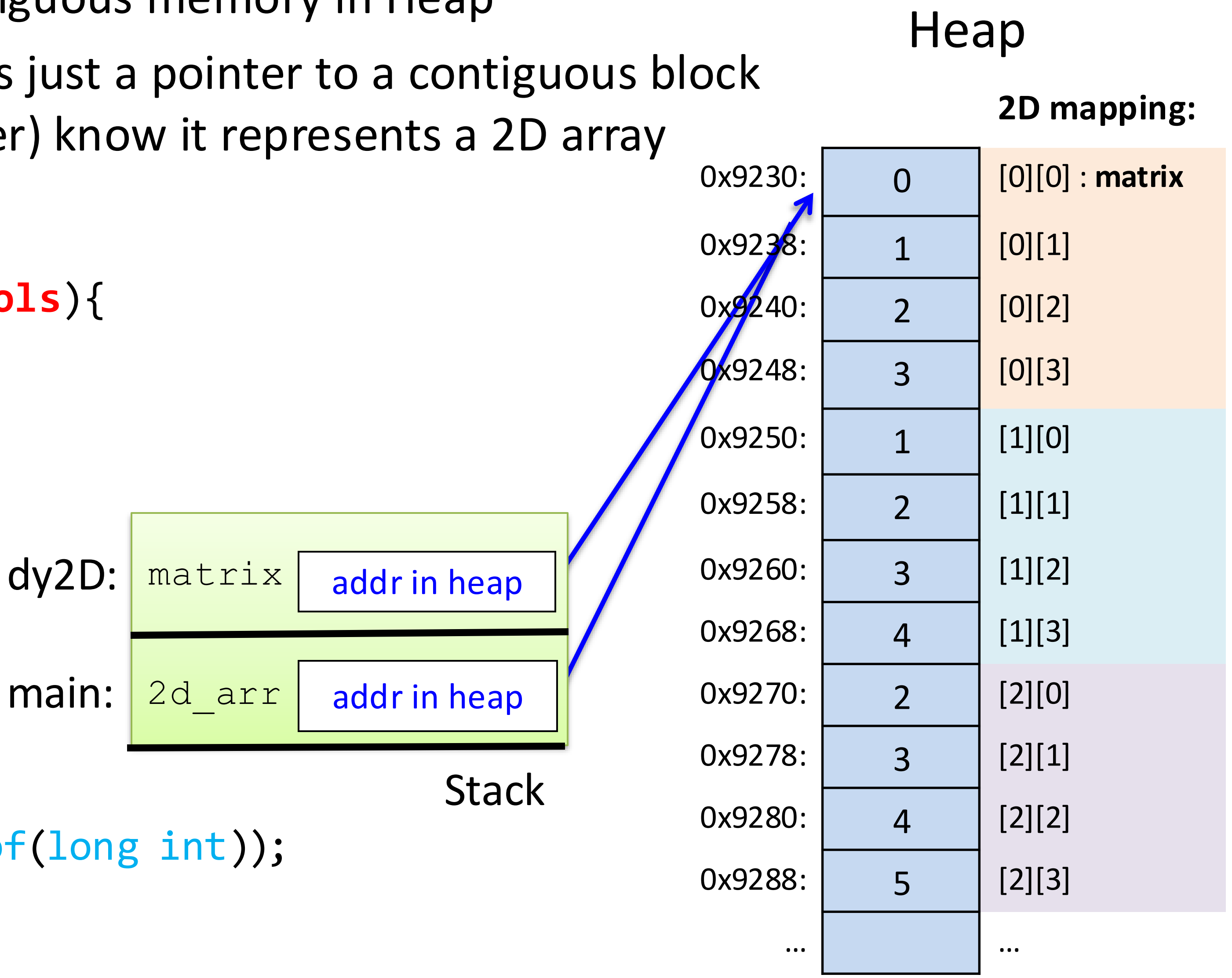
- Parameter gets base address of contiguous memory in Heap
- Just like 1D arrays (almost). **Why?** It’s just a pointer to a contiguous block of memory, only we (the programmer) know it represents a 2D array
- Pass *row* and *column* dimensions

```
void dy2D(int *matrix, int rows, int cols){
   int i, j;
   for(i=0; i < rows; i++) {
        for(j=0; j< cols; j++) {
            matrix[i*cols + j] = i*j;
    }
   }
}
int main() {
  long int *2d_arr = malloc(3 * 4 * sizeof(long int));
  dy2D(2d_arr, 3, 4);
}
```

Using Dynamically Allocated 2D Arrays as Parameters

- Parameter gets base address of contiguous memory in Heap
- Just like 1D arrays (almost). **Why?** It's just a pointer to a contiguous block of memory, only we (the programmer) know it represents a 2D array
- Pass *row* and *column* dimensions

```
void dy2D(int *matrix, int rows, int cols){
   int i, j;
   for(i=0; i < rows; i++) {
        for(j=0; j< cols; j++) {
             matrix[i*cols + j] = i*j;
    }
   }
}
int main() {
  long int *2d_arr = malloc(3 * 4 * sizeof(long int));
  dy2D(2d_arr, 3, 4);
}
```



Stack

dy2D:	matrix	addr in heap
main:	2d_arr	addr in heap

Heap

		2D mapping:
0x9230:	0	[0][0] : **matrix**
0x9238:	1	[0][1]
0x9240:	2	[0][2]
0x9248:	3	[0][3]
0x9250:	1	[1][0]
0x9258:	2	[1][1]
0x9260:	3	[1][2]
0x9268:	4	[1][3]
0x9270:	2	[2][0]
0x9278:	3	[2][1]
0x9280:	4	[2][2]
0x9288:	5	[2][3]
…		…

But… can’t we have pointers to pointers?

- If we want a dynamic **array** of **ints**:
  - declare `int *array = malloc(N * sizeof(int))`
  - Treat this internally as a 2D array `(i*COL + j)`

- If we want an **array** of **int pointers**:
  - declare `int **array = malloc(…)`
  - For *each pointer*, dynamically allocate an array

But… can't we have pointers to pointers?

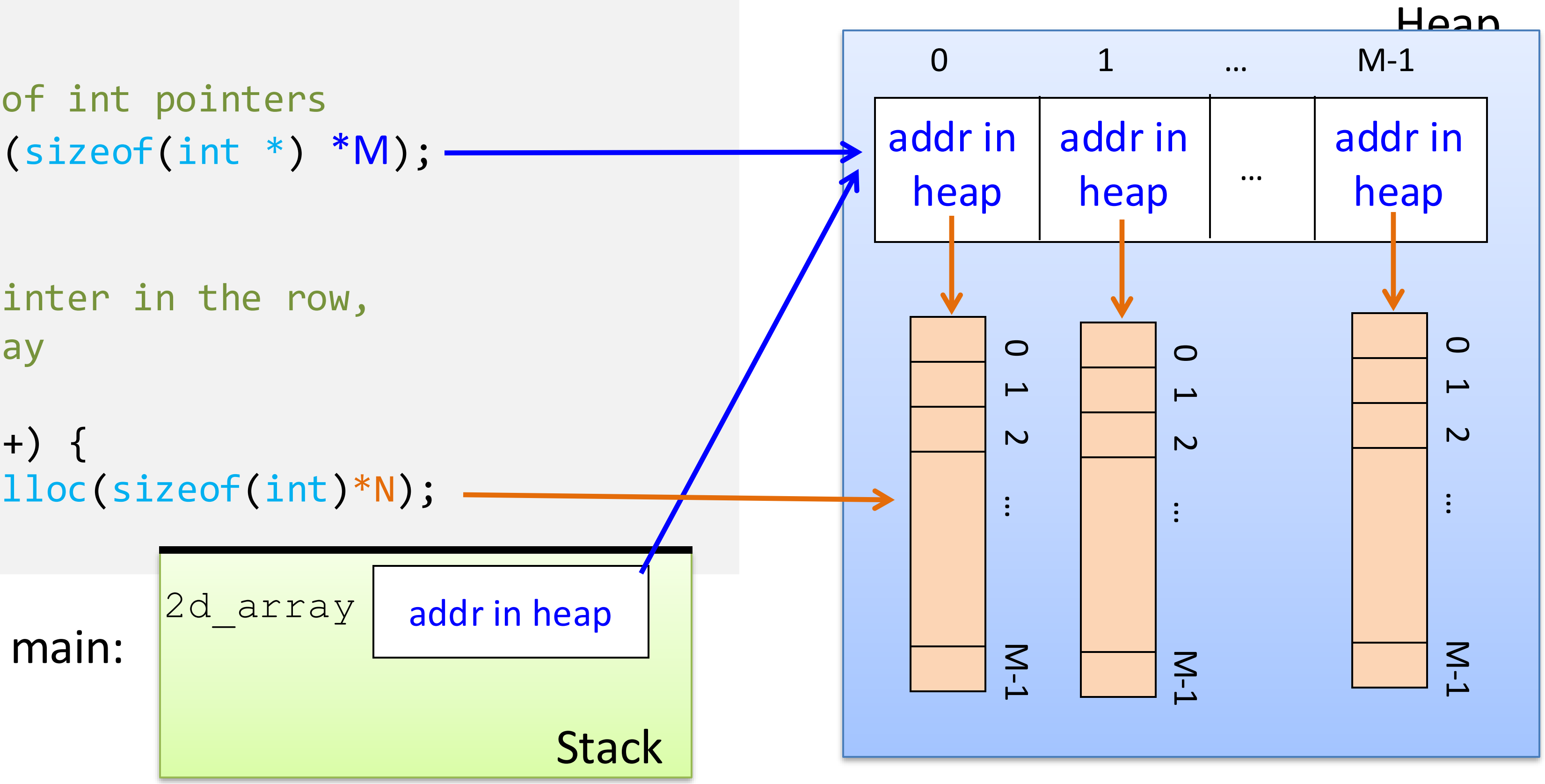
- If we want a dynamic **array** of **ints**:
  - declare `int *array = malloc(N * sizeof(int))`
  - Treat this internally as a 2D array (`i*COL + j`)

- If we want an **array** of **int pointers**:
  - declare `int **array = malloc(…)`
  - For *each pointer*, dynamically allocate an array
  - The type of array[0], array[1], etc. is: `int *`
  - For each one of those, we can malloc an array of ints:
    - `array[0] = malloc(M * sizeof(int))`

Dynamically Allocated 2D Array: Array of Pointers

- One malloc for an array of rows: an array of `int*`
- N mallocs for each row's column values: arrays of `int`
  - variable type is `int**`
  - stores address of rows array: an array of `int*`

```
int ** 2d_array;

// allocate a row of int pointers
2d_array = malloc (sizeof(int *) *M);


// for each int pointer in the row,
// allocate an array

for(i=0; i < M; i++) {
  2d_array[i] = malloc(sizeof(int)*N);
}
```

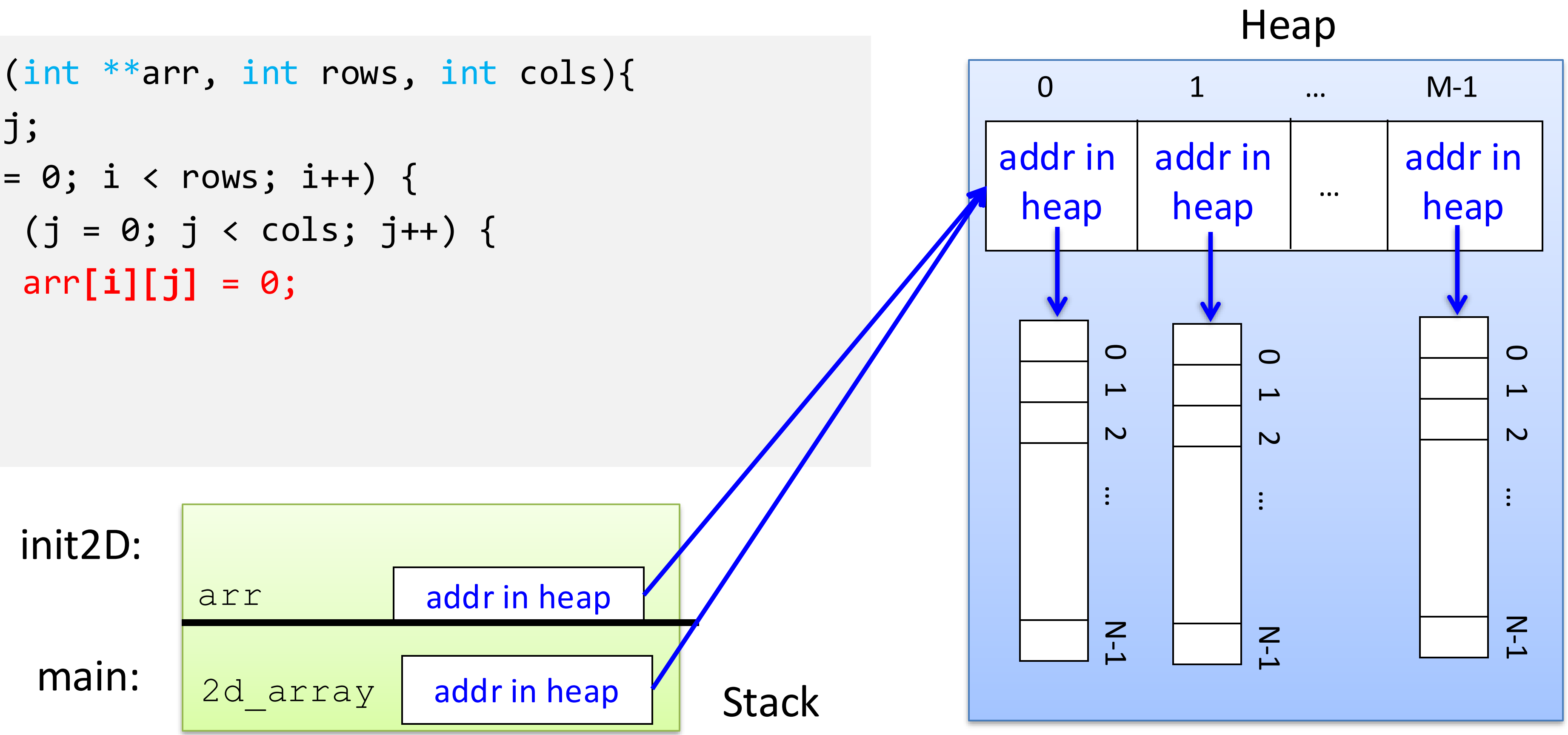


Using 2D Array (Array of Pointers) As Parameters

parameter gets base address of rows array of `int*`

- its type is `int**` : a pointer to `int*`: (with buckets of `int`)
- pass row and column dimension values
- Can use [i][j] to index into a specific location in 2D array.

```
void init2D(int **arr, int rows, int cols){
    int i, j;
    for (i = 0; i < rows; i++) {
        for (j = 0; j < cols; j++) {
            arr[i][j] = 0;
        }
    }
}
```

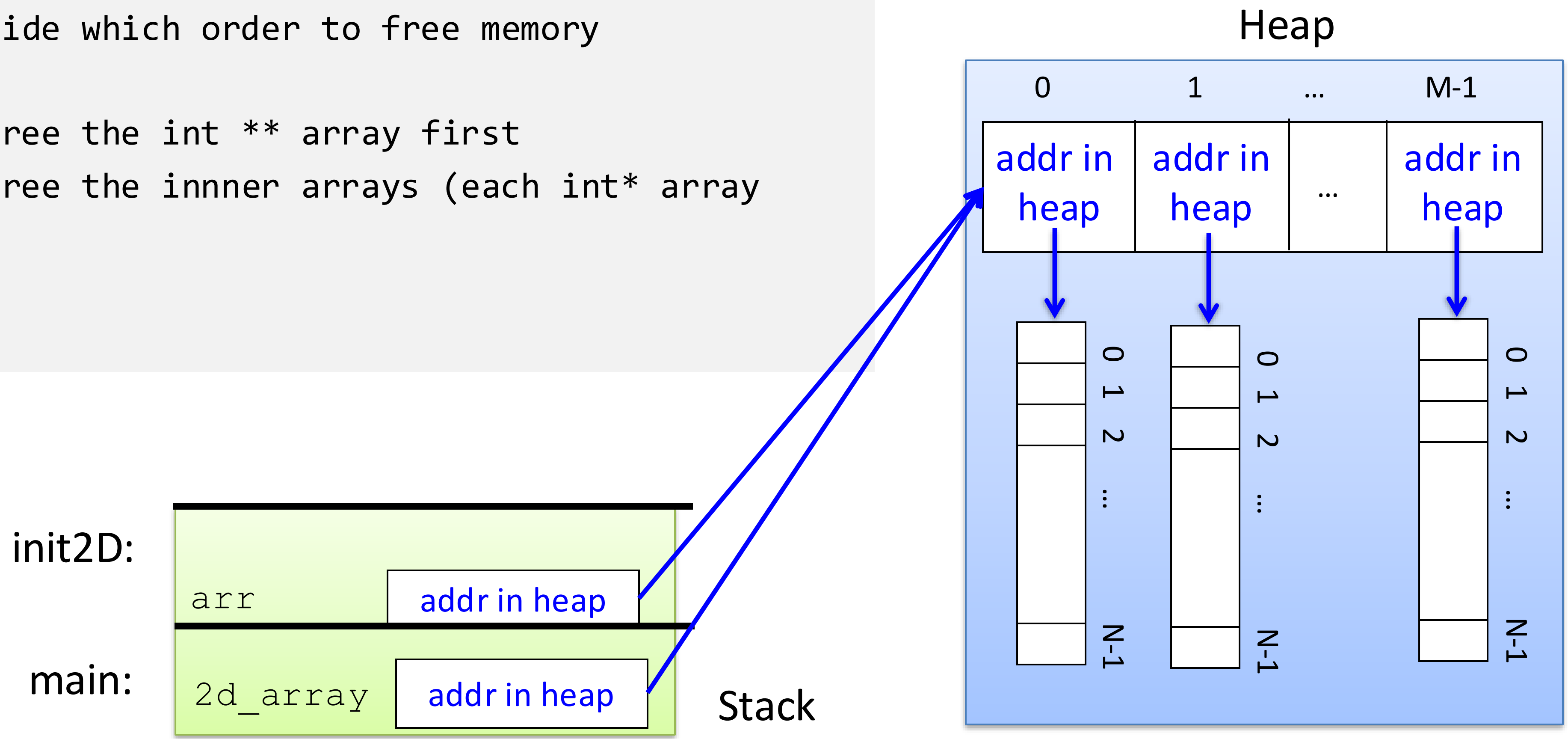


Using 2D Array (Array of Pointers): How about free-ing this memory?

parameter gets base address of rows array of `int*`

- its type is `int** ->` a pointer to an array of `int*->`
- each `int* ->` a pointer to an array of `ints`

```
void free(int **arr){
//TODO: decide which order to free memory

Option A: free the int ** array first
Option B: free the innner arrays (each int* array
first)

}
```



Two Ways for 2D Arrays

- We'll use BOTH methods in future labs:
  - **Lab 7**:
    - column-major, large chunk of memory that we treat as a 2D array,
    - use arr[index] where index = i * ROWSIZE + j to deference values
  - **Lab 8/9**:
    - array of integer pointers,
    - can use arr[N][M] to dereference values

Structs

- Multiple values (fields) stored together
  - Defines a new type in C's type system

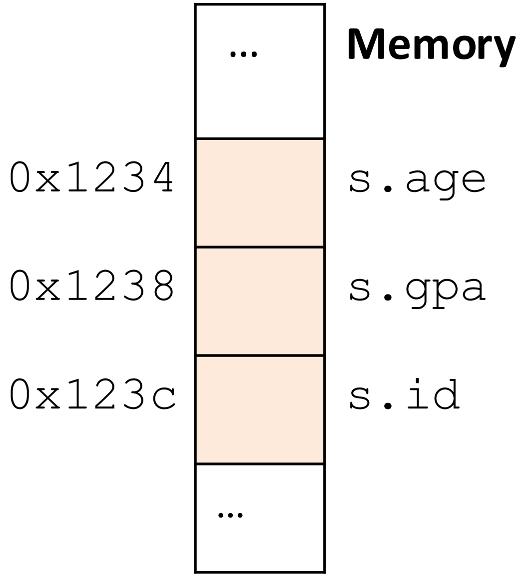
- Laid out contiguously by field (with a caveat we'll see later)
  - In order of field declaration.

Structs

Laid out contiguously by field (with a caveat we'll see later)

- In order of field declaration.

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Memory

…

0x1234 s.age

0x1238 s.gpa

0x123c s.id

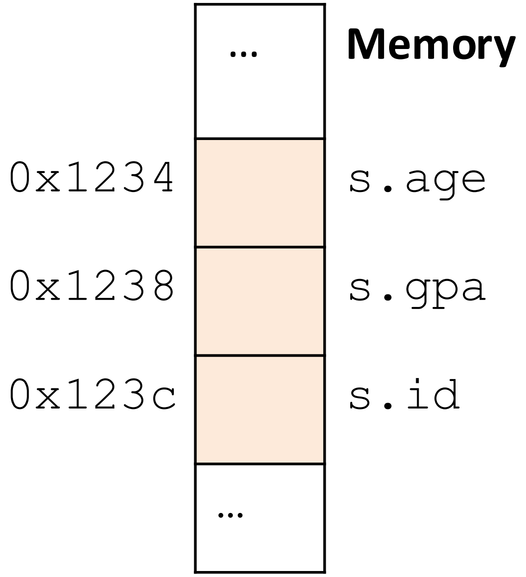
…

Structs

Struct fields accessible as a base + displacement

- Compiler knows (constant) displacement of each field

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```

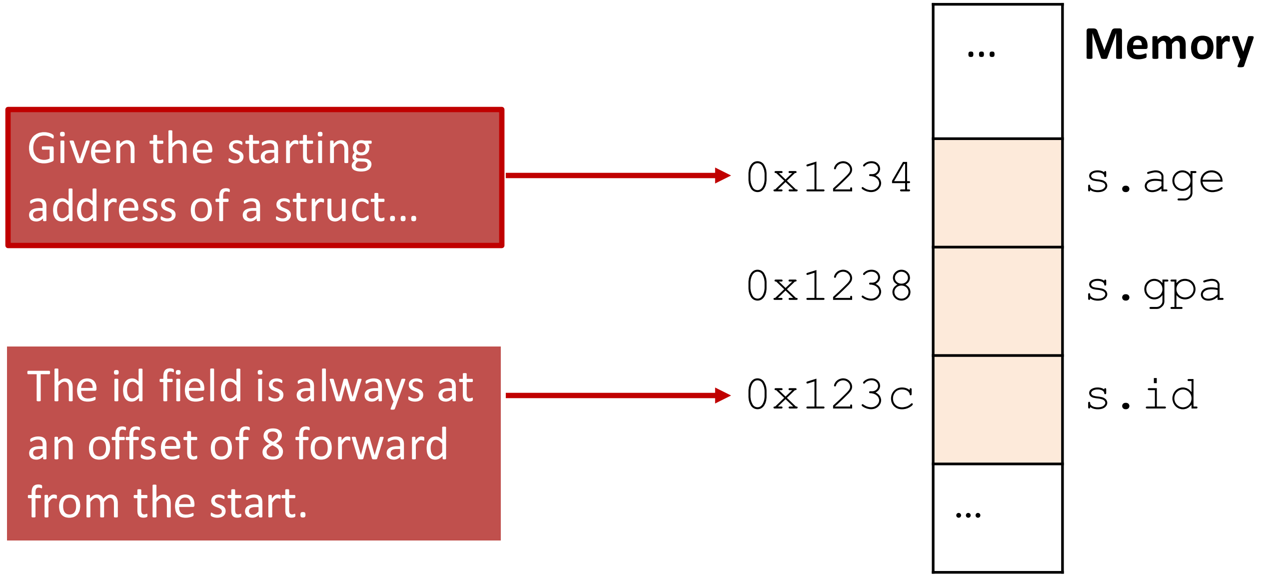


Structs

Struct fields accessible as a base + displacement

- Compiler knows (constant) displacement of each field

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Given the starting address of a struct…

The id field is always at an offset of 8 forward from the start.

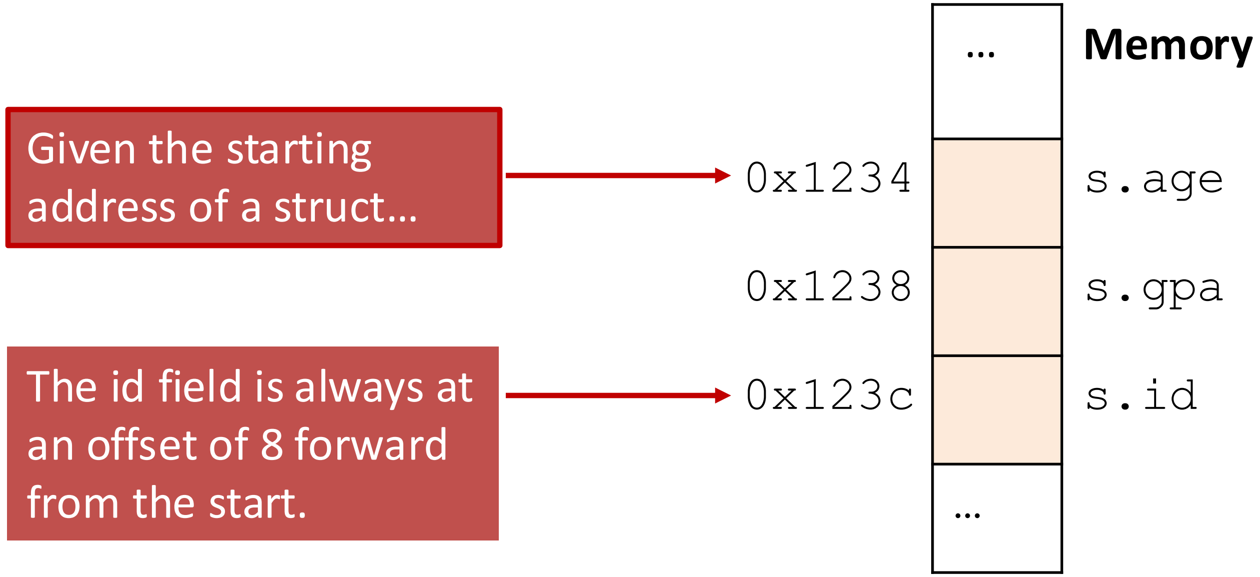
Structs

Struct fields accessible as a base + displacement
In assembly: `mov reg_value, 8(reg_base)`

Where:

- `reg_value` is a register holding the value to store (say, 12)
- `reg_base` is a register holding the base address of the struct

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
s.id = 12;
```

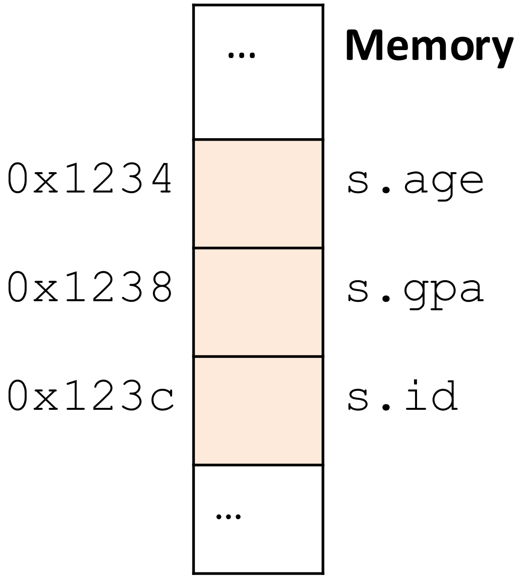


Given the starting address of a struct…

The id field is always at an offset of 8 forward from the start.

Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.



	…	**Memory**
`0x1234`		`s.age`
`0x1238`		`s.gpa`
`0x123c`		`s.id`
	…	

Data Alignment:

- Where (which address) can a field be located?

- char (1 byte): can be allocated at any address:

  0x1230, 0x1231, 0x1232, 0x1233, 0x1234, …

- short (2 bytes): must be aligned on 2-byte addresses:

  0x123**0**, 0x123**2**, 0x123**4**, 0x123**6**, 0x123**8**, …

- int (4 bytes): must be aligned on 4-byte addresses:

  0x123**0**, 0x123**4**, 0x123**8**, 0x123**c**, 0x124**0**, …

Why do we want to align data on multiples of the data size?

A. It makes the hardware faster.

B. It makes the hardware simpler.

C. It makes more efficient use of memory space.

D. It makes implementing the OS easier.

E. Some other reason.

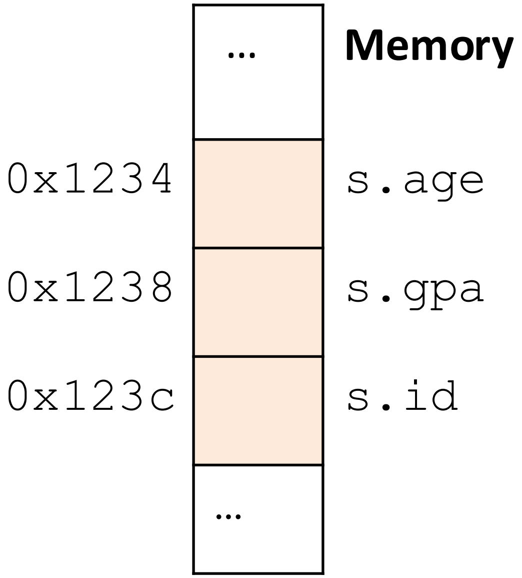
Data Alignment: Why?

- Simplify hardware
  - e.g., only read ints from multiples of 4
  - Don’t need to build wiring to access 4-byte chunks at any arbitrary location in hardware
- Inefficient to load/store single value across alignment boundary (1 vs. 2 loads)
- Simplify OS:
  - Prevents data from spanning virtual pages
  - Atomicity issues with load/store across boundary

Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Structs

```
struct student{
  char name[11];
  short age;
  int id;
};
```

How much space do we need to store one of these structures? Why?

```
struct student{
  char name[11];
  short age;
  int id;
};
```

A. 17 bytes
B. 18 bytes
C. 20 bytes
D. 22 bytes
E. 24 bytes

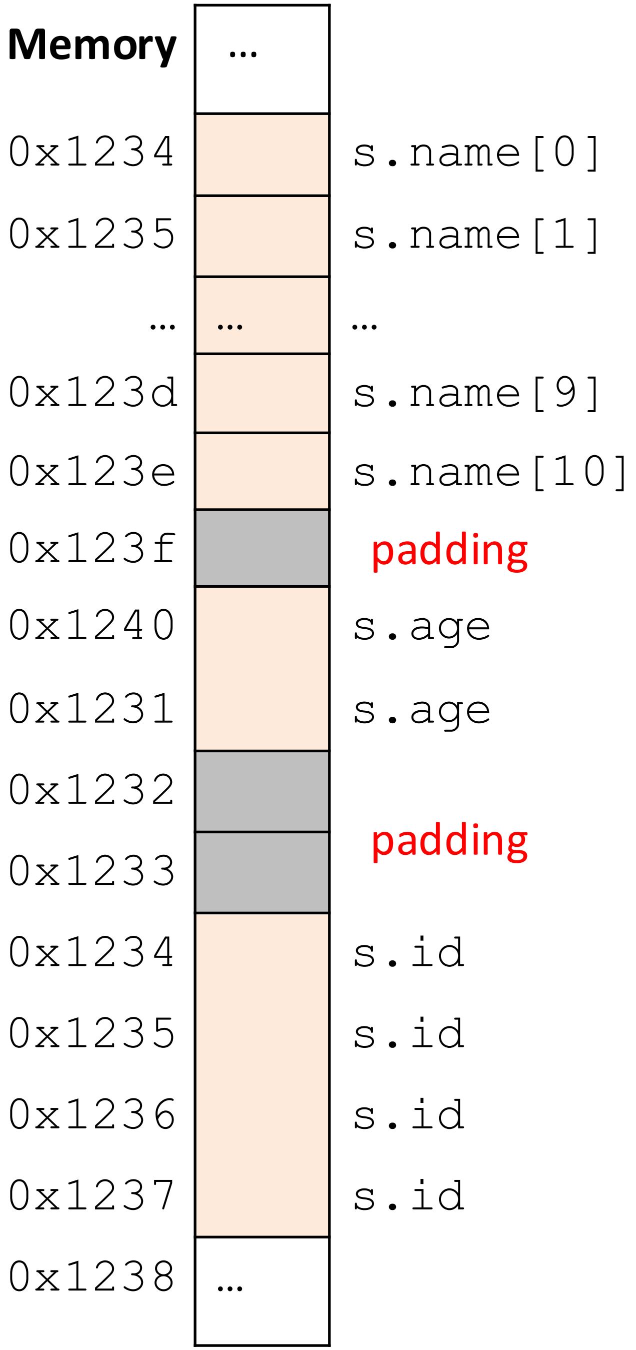
Structs

```
struct student{
  char name[11];
  short age;
  int id;
};
```

size of data: 17 bytes

size of struct: 20 bytes!

Use sizeof() when allocating structs with malloc()!



Memory	…	
0x1234		s.name[0]
0x1235		s.name[1]
…	…	…
0x123d		s.name[9]
0x123e		s.name[10]
0x123f		padding
0x1240		s.age
0x1231		s.age
0x1232		padding
0x1233		
0x1234		s.id
0x1235		s.id
0x1236		s.id
0x1237		s.id
0x1238	…	

Alternative Layout

```
struct student{
   char name[11];
   short age;
   int id;
};
```

Same fields, declared in a different order.

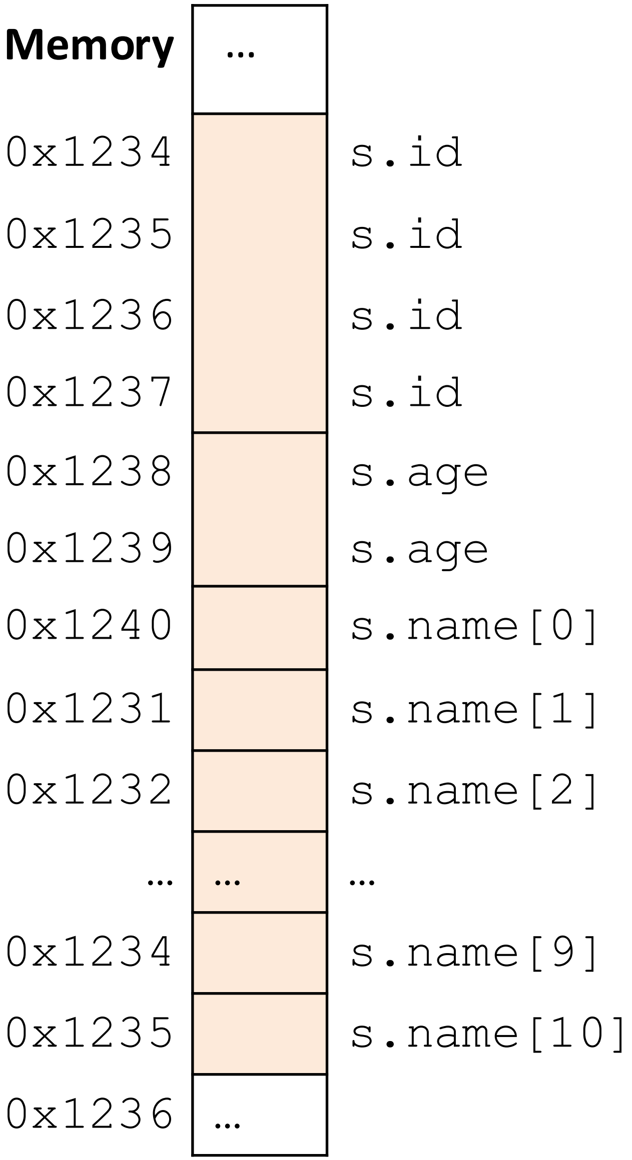
Alternative Layout

```
struct student{
  char name[11];
  short age;
  int id;
};
```

`size of data:  17 bytes`

`size of struct: 17 bytes`

In general, this isn't a big deal on a day-to-day basis. Don't go out and rearrange all your struct declarations.



Memory	…	
0x1234		s.id
0x1235		s.id
0x1236		s.id
0x1237		s.id
0x1238		s.age
0x1239		s.age
0x1240		s.name[0]
0x1231		s.name[1]
0x1232		s.name[2]
…	…	…
0x1234		s.name[9]
0x1235		s.name[10]
0x1236	…	

Aside: Network Headers

- In networks, we attach metadata to packets
  - Things like destination address, port #, etc.
- Common for these to be a specific size/format
  - e.g., the first 20 bytes must be laid out like …
- Naïvely declaring a struct might introduce padding, violate format.

Cool, so we can get rid of this struct padding by being smart about declarations?

A. Yes (why?)

B. No (why not?)

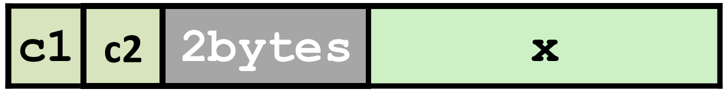
Cool, so we can get rid of this padding by being smart about declarations?

- Answer: Maybe.
- Rearranging helps, but often padding after the struct can’t be eliminated.

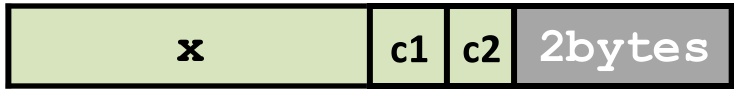
```
struct T1 {
    char c1;
    char c2;
    int  x;
};
```

```
struct T2 {
    int x;
    char c1;
    char c2;
};
```

T1 :



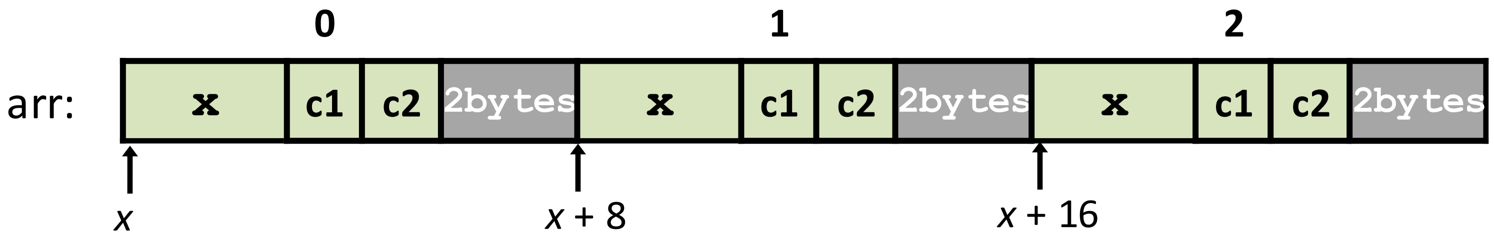
T2 :



“External” Padding

Array of Structs: Field values in each bucket must be properly aligned:

```
struct T2 arr[3];
```



Buckets must be on a 8-byte aligned address

Struct field syntax…

```
struct student {
  int id;
  short age;
  char name[11];
};
struct student s;

s.id = 406432;
s.age = 20;
strcpy(s.name, “Alice”);
```

Struct is declared on the stack.
(NOT a pointer)

Struct field syntax…

What about this?

```
struct student {
  int id;
  short age;
  char name[11];
};
struct student *s = malloc(sizeof(struct student));
```

How do we get to the id and age?

Struct field syntax…

What about this?

```
struct student {
   int id;
   short age;
   char name[11];
};
struct student *s = malloc(sizeof(struct student));
```

How do we get to the id and age?

Option 1: Works but ugly

```
(*s).id = 406432;
(*s).age = 20;
strcpy((*s).name, “Alice”);
```

Option 2: Use struct pointer dereference!

```
s->id = 406432;
s->age = 20;
strcpy(s->name, “Alice”);
```

✓

Memory alignment applies elsewhere too!

```
int x;         vs.    double y;
char ch[5];           int x;
short s;              short s;
double y;             char ch[5];
```

In nearly all cases, *you shouldn't stress about this.* The compiler will figure out where to put things.

Exceptions: networking, OS

Structs and Arrays

- Use Structs & Arrays to build complex data types
- Very important to think about type!

  from the outside in: (e.g.) a[3].age
  - type of a is a pointer to an array of student
  - can use [i] notation to access a bucket of this array
  - type of a[3] is a student struct
  - can use **.** to access a field in struct
  - type of a[3].age is an int
- Remember how different types are passed
  - semantics of passing an array vs. a struct
  - it is all pass by value, but what value is differs by type

Up next…

- New topic: Storage and the Memory Hierarchy

Transition

- First half of course: hardware focus
  - How the hardware is constructed
  - How the hardware works
  - How to interact with hardware / ISA

- Up next: performance and software systems
  - Memory performance
  - Operating systems
  - Standard libraries (strings, threads, etc.)

Efficiency

- How to <u>Efficiently</u> Run Programs
- Good algorithm is critical…
- Many systems concerns to account for too!
  - The memory hierarchy and its effect on program performance
  - OS abstractions for running programs efficiently
  - Support for parallel programming

Efficiency

- How to Efficiently Run Programs
- Good algorithm is critical…
- Many systems concerns to account for too!
  - The memory hierarchy and its effect on program performance
  - OS abstractions for running programs efficiently
  - Support for parallel programming

Suppose you're designing a new computer architecture. Which type of memory would you use? Why?

A. low-capacity (~1 MB), fast, expensive

B. medium-capacity (a few GB), medium-speed, moderate cost

C. high-capacity (100's of GB), slow, cheap

D. something else (it must exist)

trade-off between capacity and speed

Classifying Memory

- Broadly, two types of memory:
  1. Primary storage: CPU instructions can access any location at any time (assuming OS permission)
  2. Secondary storage: CPU can’t access this directly

Random Access Memory (RAM)

- Any location can be accessed directly by CPU
  - Volatile Storage: lose power → lose contents
- Static RAM (SRAM)
  - Latch-Based Memory (e.g. RS latch), 1 bit per latch
  - Faster and more expensive than DRAM
    - “On chip”: Registers, Caches
- Dynamic RAM (DRAM)
  - Capacitor-Based Memory, 1 bit per capacitor
    - “Main memory”: Not part of CPU

Memory Technologies

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, \$2000 – \$5000 per GB

- Dynamic RAM (DRAM)
  - 50ns – 100ns, \$20 – \$75 per GB
    (Main memory, “RAM”)

We’ve talked a lot about registers (SRAM) and we’ll cover caches (SRAM) soon. Let’s look at main memory (DRAM) now.

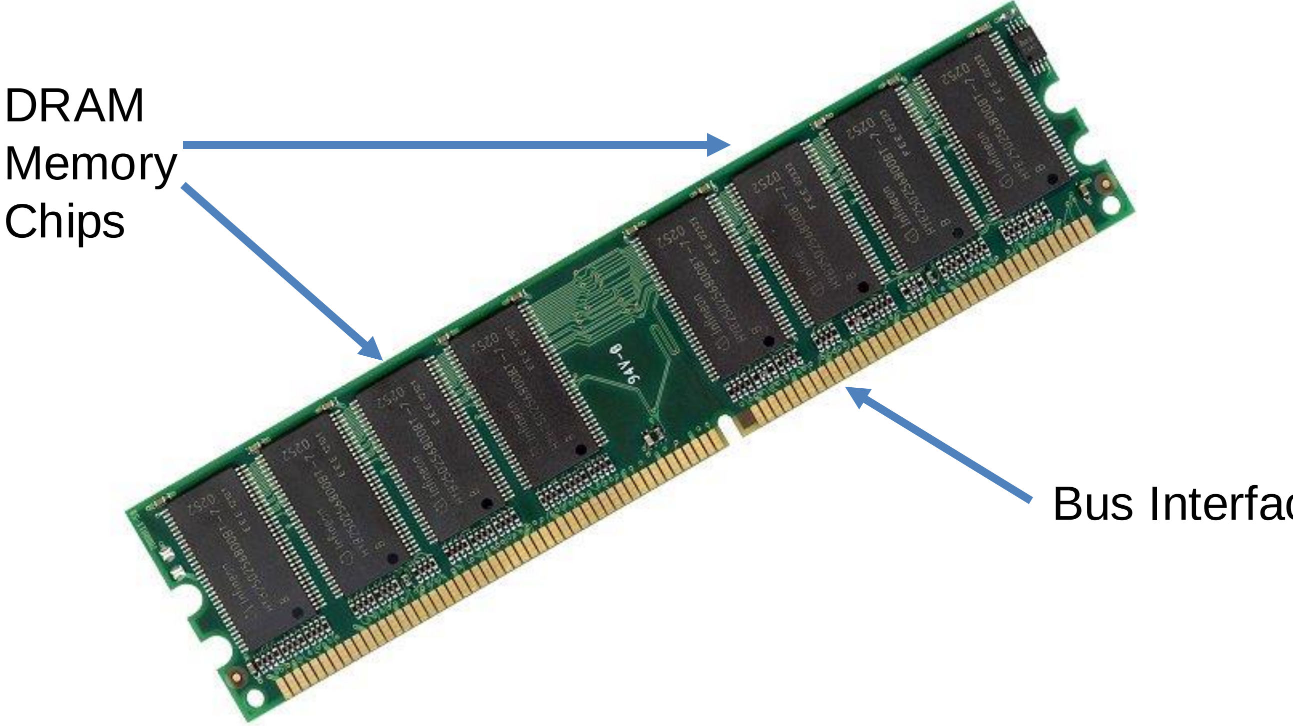
Dynamic Random Access Memory (DRAM)

Capacitor based:

- cheaper and slower than SRAM
- capacitors are leaky (lose charge over time)
- Dynamic: value needs to be refreshed (every 10-100ms)

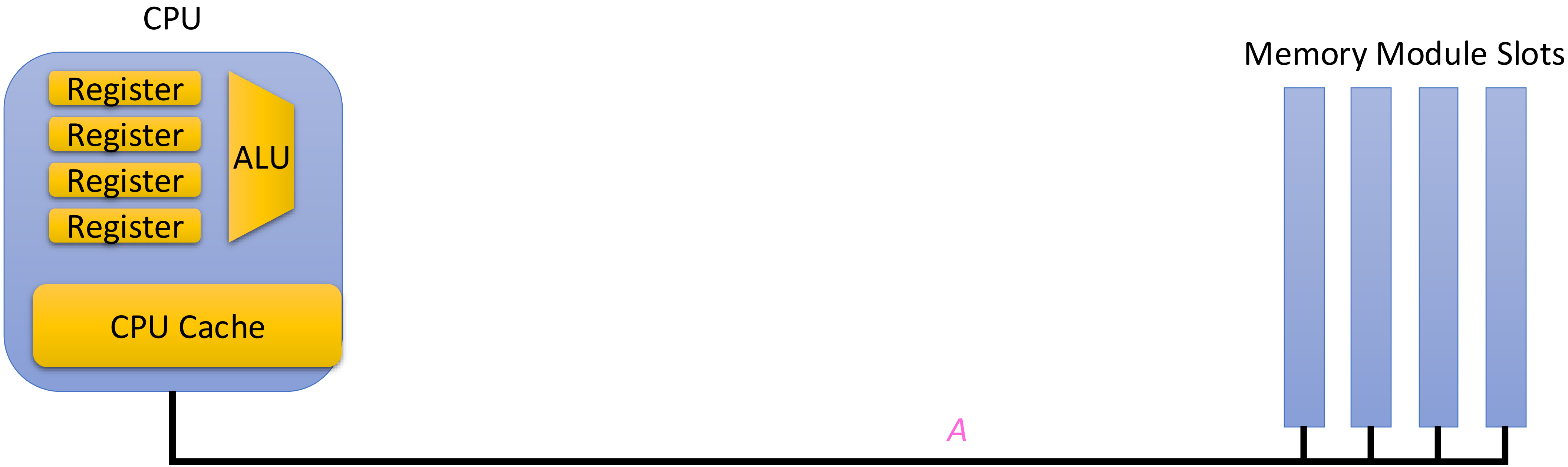
Example: DIMM

(Dual In-line Memory Module):



Connecting CPU and Memory

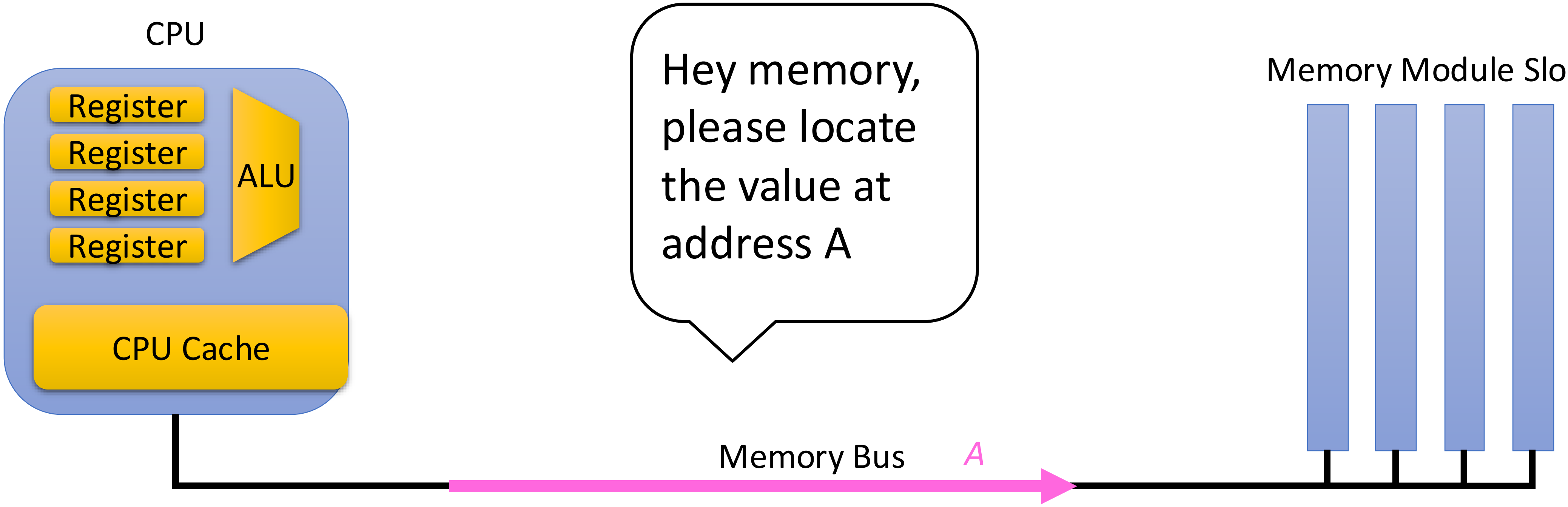
- Components are connected by a bus:
  - A bus is a collection of parallel wires that carry address, data, and control signals.
  - Buses are typically shared by multiple devices.



How A Memory Read Works

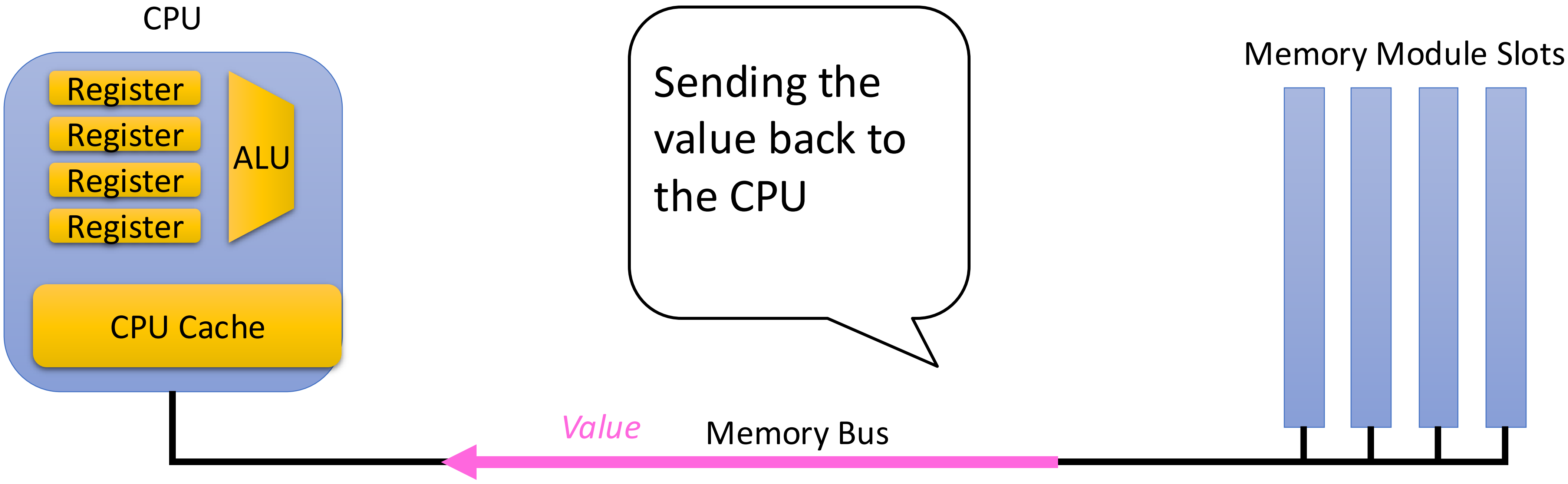
(1) CPU places address A on the memory bus.

Load operation: `mov (Address A), %rax`



How A Memory Read Works

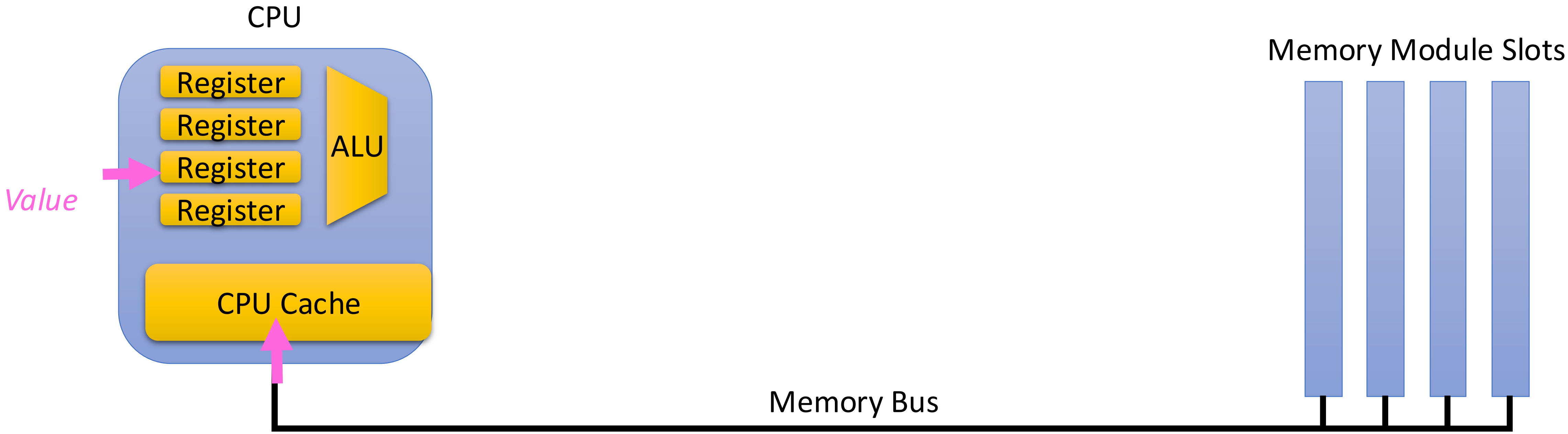
(2) Main Memory reads address A from memory, fetches value at that address and puts it on the bus



How A Memory Read Works

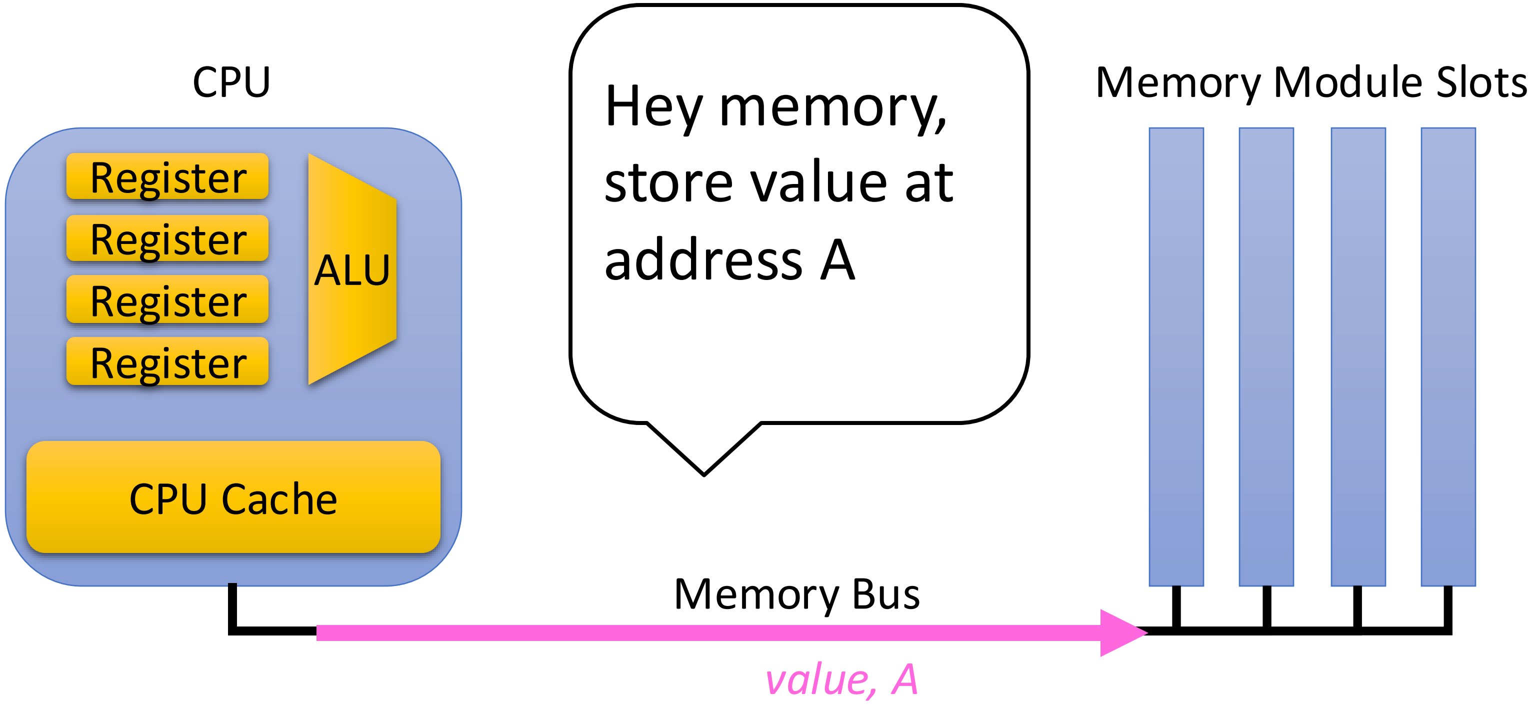
(3) CPU reads value from the bus, and copies it into register rax.

a copy also goes into the on-chip cache memory



How a Memory Write Works

1. CPU writes A to bus, memory reads it
2. CPU writes value to bus, memory reads it
3. Memory stores value at address A



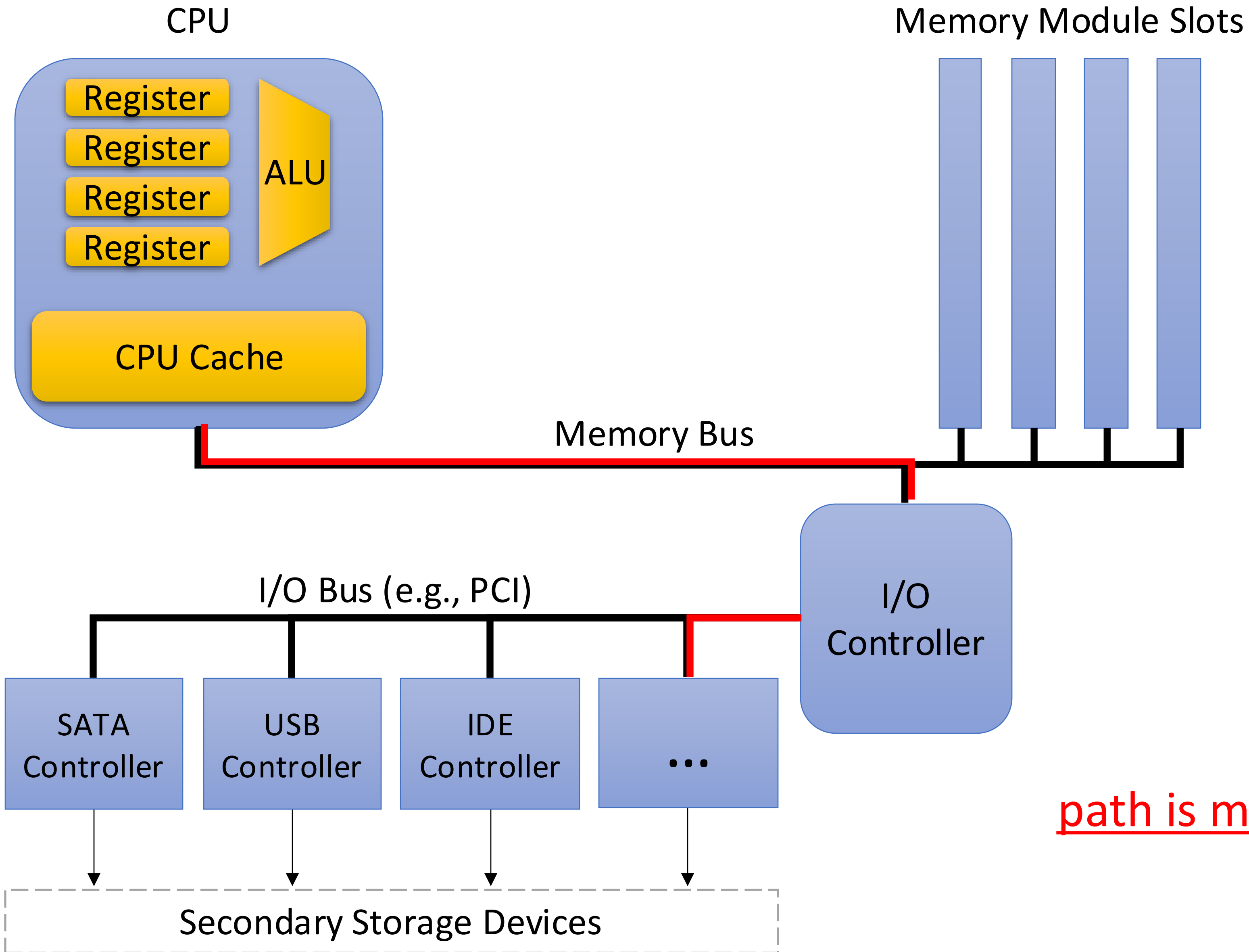
Secondary Storage

- Disk, Tape Drives, Flash Solid State Drives, …

- Non-volatile: retains data without a charge

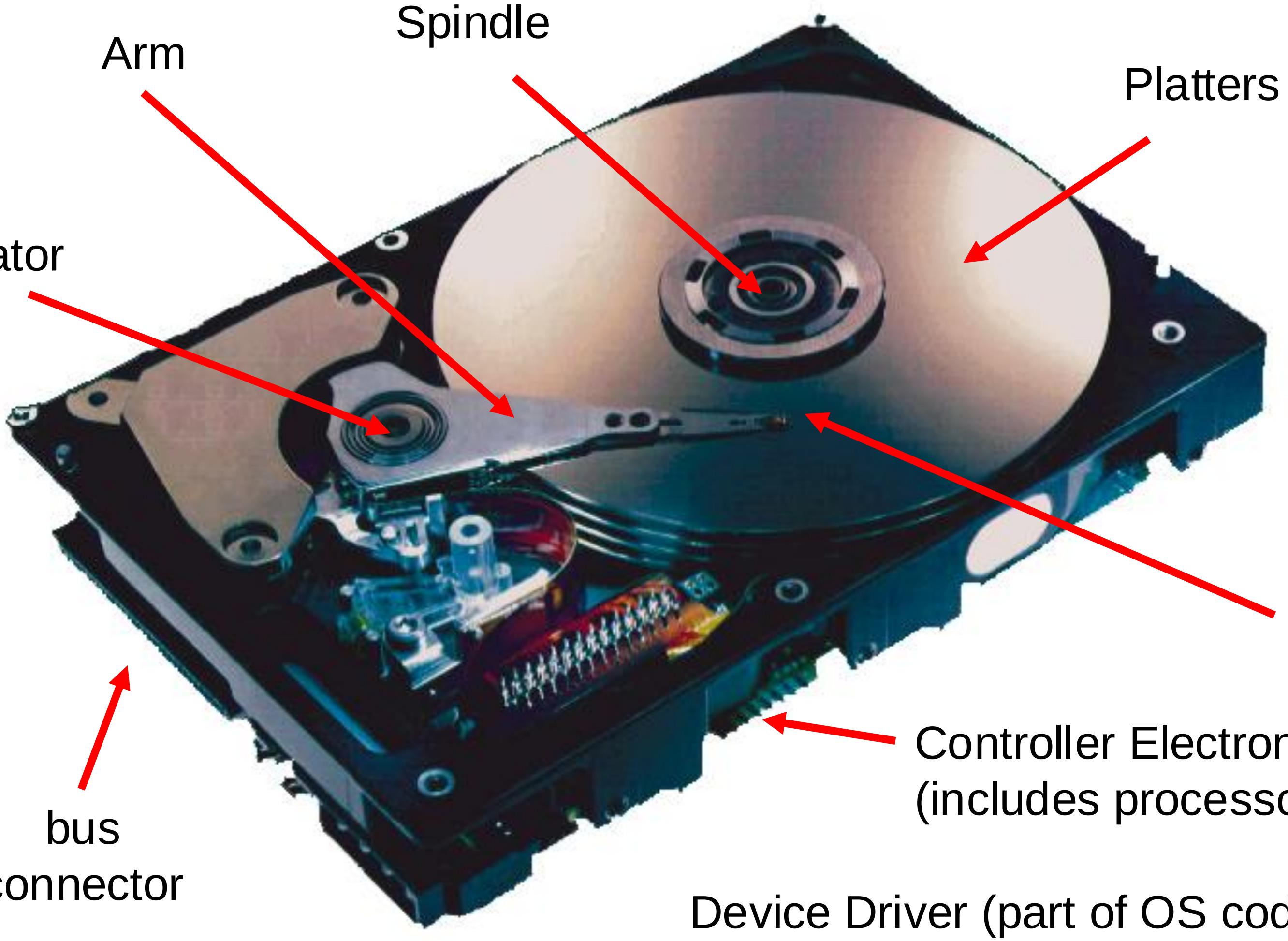
- Instructions CANNOT directly access data on secondary storage
  - No way to specify a disk location in an instruction
  - Operating System moves data to/from memory

Secondary Storage



path is much longer

What’s Inside A Disk Drive?



Spindle

Arm

Platters

Data Encoded as
points of
magnetism on
Platter surfaces

Actuator

R/W head

Controller Electronics
(includes processor & memory)

bus
connector

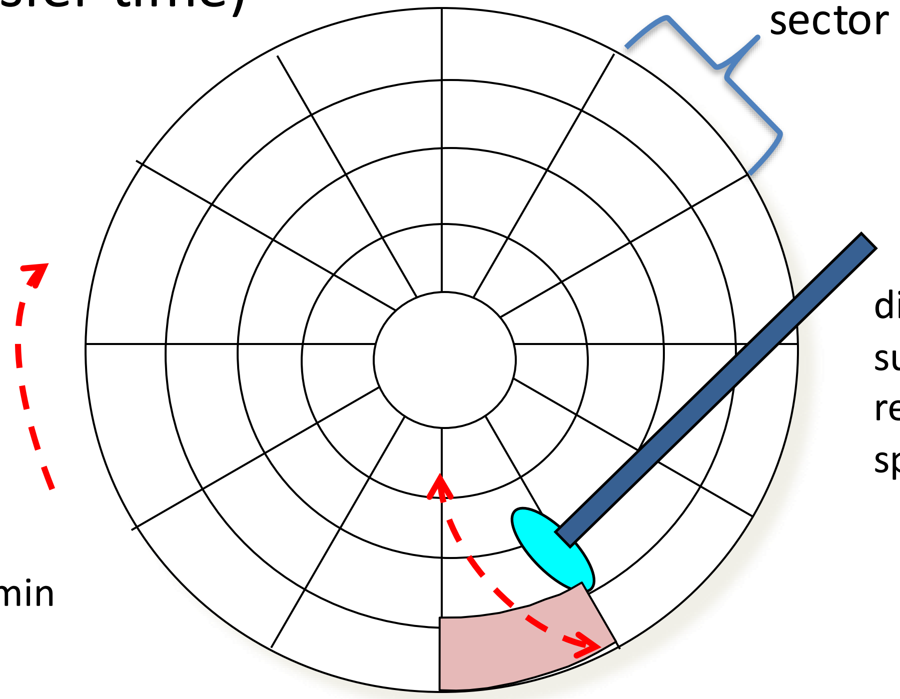
Device Driver (part of OS code)
interacts with Controller to R/W to disk

Image from Seagate Technology

Reading and Writing to Disk

Data blocks located in some Sector of some Track on some Surface

1. Disk Arm moves to correct track (seek time)
2. Wait for sector spins under R/W head (rotational latency)
3. As sector spins under head, data are Read or Written (transfer time)



disk arm sweeps across surface to position read/write head over a specific track.

disk surface spins at a fixed rotational rate ~7200 rotations/min

Memory Technology

- Static RAM (SRAM)
  - 0.5ns – 2.5ns, $2000 – $5000 per GB

Like walking:

Down the hall

- Dynamic RAM (DRAM)
  - 50ns – 100ns, $20 – $75 per GB

Across campus

Solid-state disks (flash): 100 us – 1 ms, $2 - $10 per GB

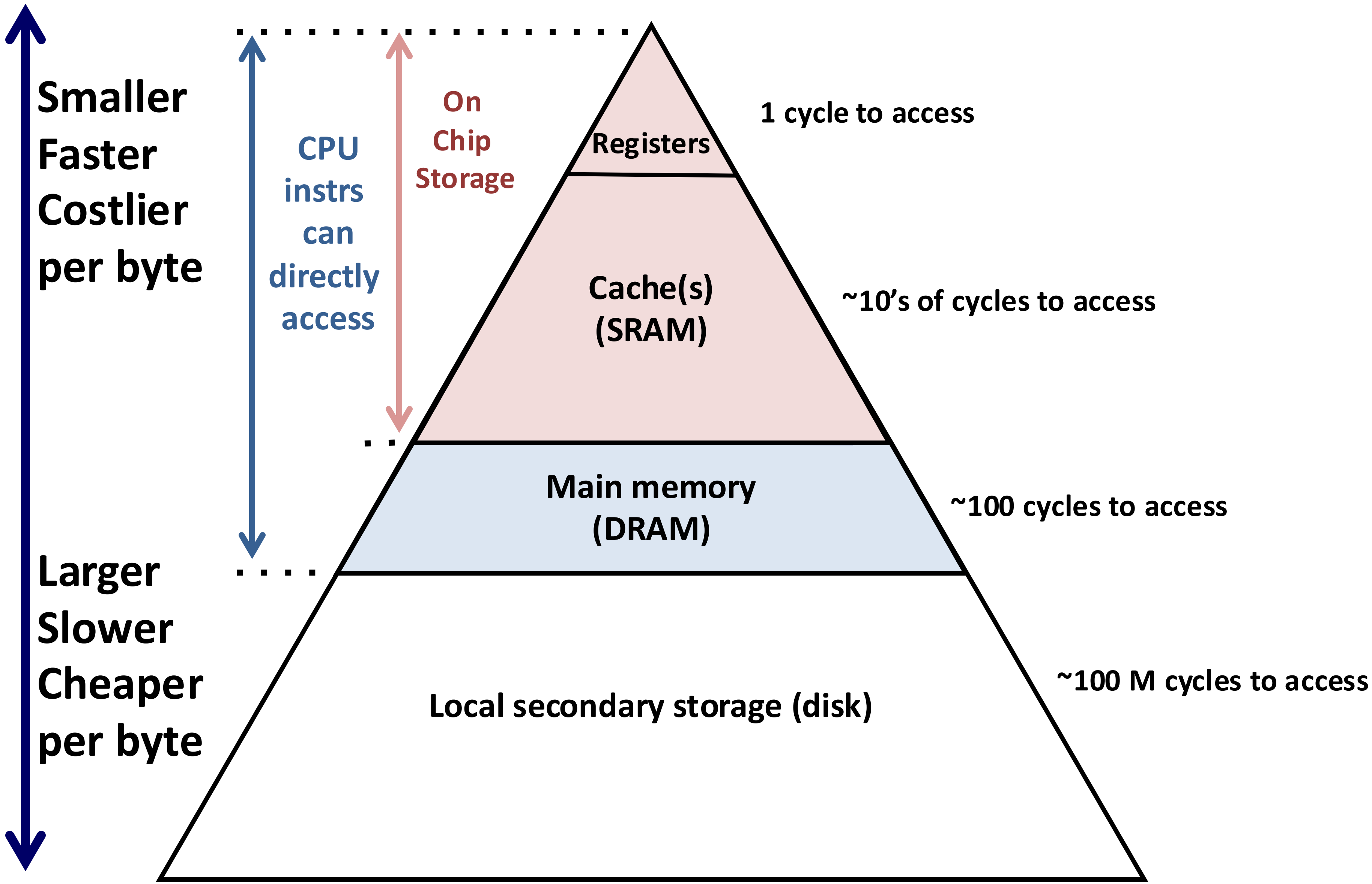
(to Cleveland / Indianapolis)

- Magnetic disk
  - 5ms – 15ms, $0.20 – $2 per GB

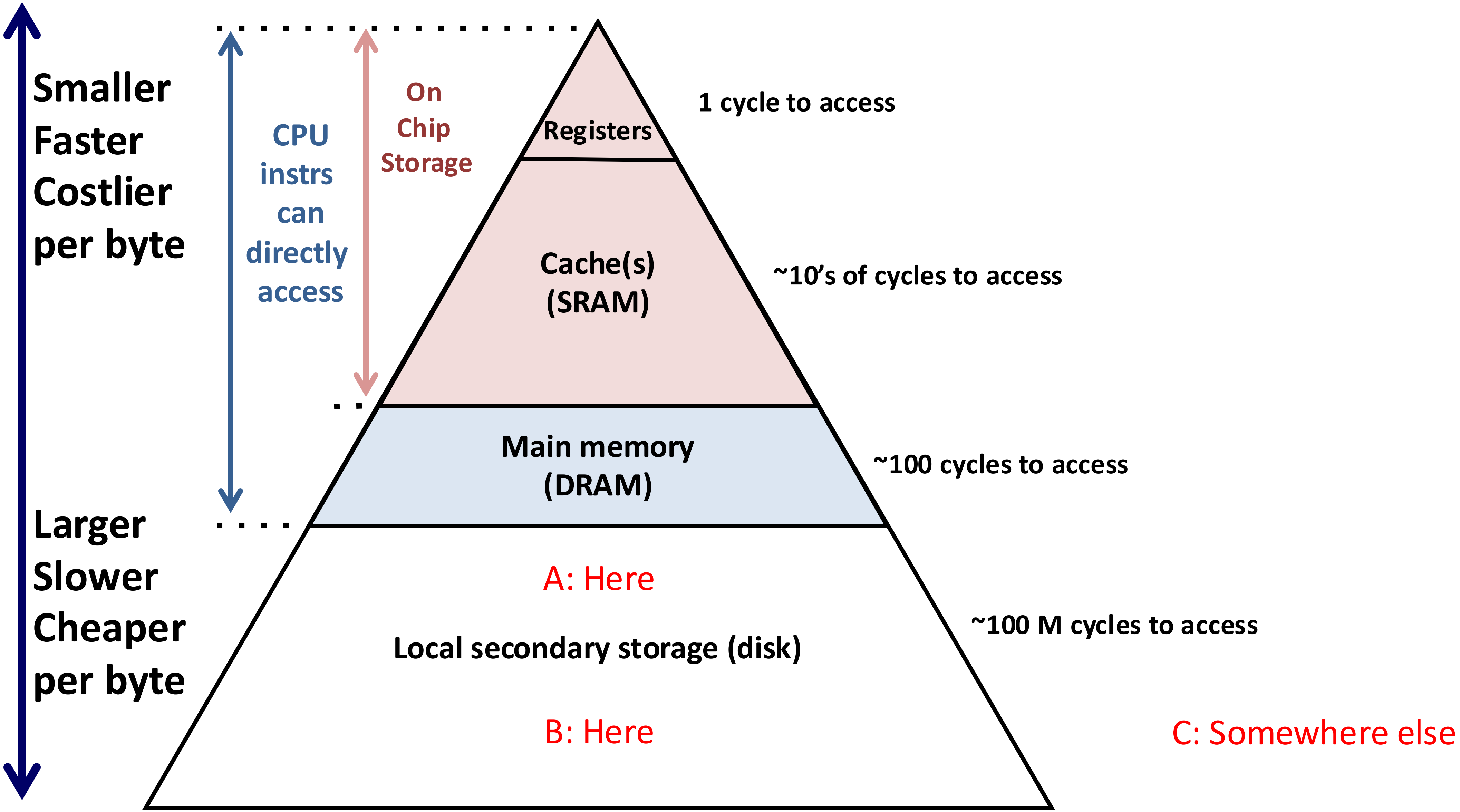
To Seattle

1 ms == 1,000,000 ns

The Memory Hierarchy



Where does accessing the network belong?



Smaller
Faster
Costlier
per byte

Larger
Slower
Cheaper
per byte

CPU instrs can directly access

On Chip Storage

Registers

1 cycle to access

Cache(s)
(SRAM)

~10's of cycles to access

Main memory
(DRAM)

~100 cycles to access

A: Here

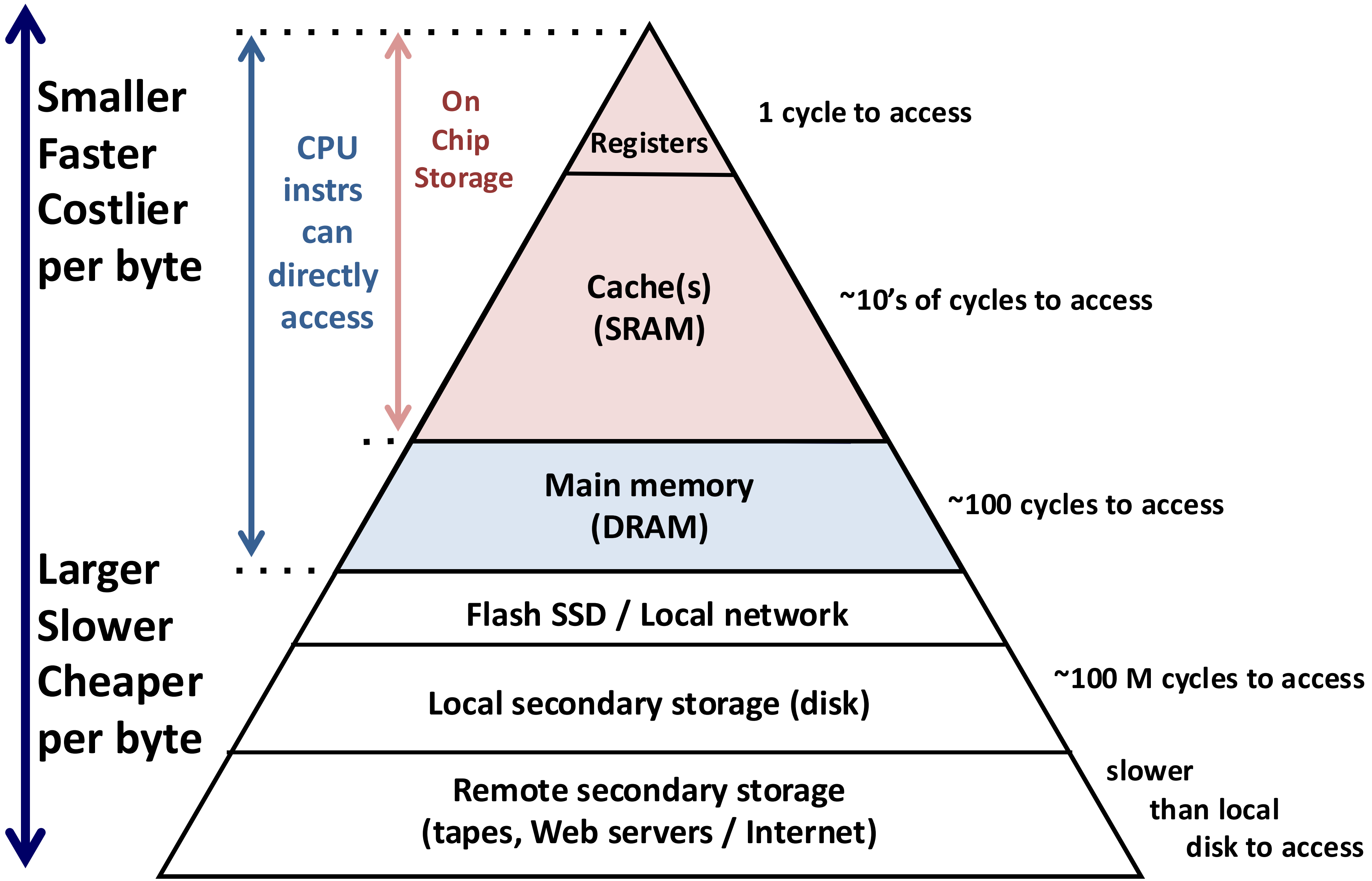
Local secondary storage (disk)

~100 M cycles to access

B: Here

C: Somewhere else

The Memory Hierarchy



Abstraction Goal

- Reality: There is no one type of memory to rule them all!
- Abstraction: hide the complex/undesirable details of reality.
- Illusion: We have the speed of SRAM, with the capacity of disk, at reasonable cost.