CS 241: Systems Programming
Lecture 22. Multidimensional Arrays

Spring 2020

Prof. Stephen Checkoway

Two-dimensional arrays

```
int tab[4][5];
```

Rectangular 2D array,

- All memory allocated as a single, contiguous block

Indices are tab[row][column];

Data is stored in **row-major** format

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4
3,0	3,1	3,2	3,3	3,4

Row-major format

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20

1 2 3 4 5 2 4 6 8 10 3 6 9 12 15 4 8 12 16 20

entry (r,c) is stored in position r*cols + c in memory

Where does C store the size of an array?

Column-major format (not C)

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20

1 2 3 4 2 4 6 8 3 6 9 12 4 8 12 16 5 10 15 20

Given the 2D array, `table`, declared as follows,

```
size_t rows = 3;
size_t cols = 4;
double table[rows][cols];
```

how much memory does `table` occupy?

A. `3*4` bytes

B. `3*4*sizeof(double)` bytes

C. `3*sizeof(size_t)*4` bytes

D. `3*sizeof(size_t)*4*sizeof(size_t)` bytes

E. `3*sizeof(size_t)*4*sizeof(size_t)*sizeof(double)` bytes

Multidimensional arrays

```
float oneD[size];
float twoD[rows][cols];
float threeD[layers][rows][cols];
...
float general[size1][size2]...[sizeN];
```

Fixed-length arrays if all dimensions are integer constants

- `int const size = 10;` is not an integer constant!
- Can initialize using nested braces
- Can omit the size of the first dimension when using an initializer

Variable-length arrays if any dimensions are not integer constants

Multidimensional arrays

0	1	2	3

```
float oneD[size];
float twoD[rows][cols];
float threeD[layers][rows][cols];
...
float general[size1][size2]...[sizeN];
```

Fixed-length arrays if all dimensions are integer constants

- `int const size = 10;` is not an integer constant!
- Can initialize using nested braces
- Can omit the size of the first dimension when using an initializer

Variable-length arrays if any dimensions are not integer constants

Multidimensional arrays

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

0	1	2	3

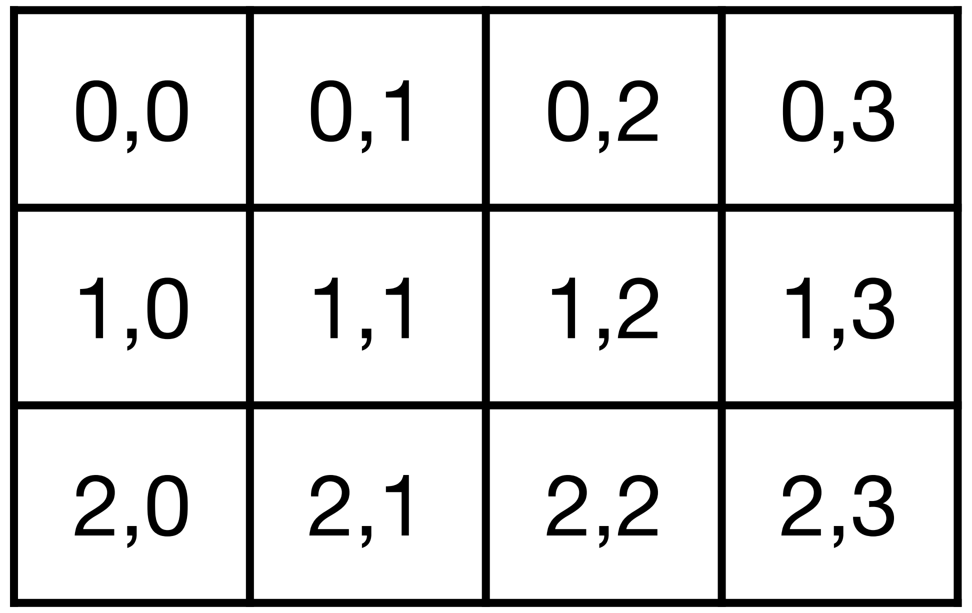
```
float oneD[size];
float twoD[rows][cols];
float threeD[layers][rows][cols];
...
float general[size1][size2]...[sizeN];
```

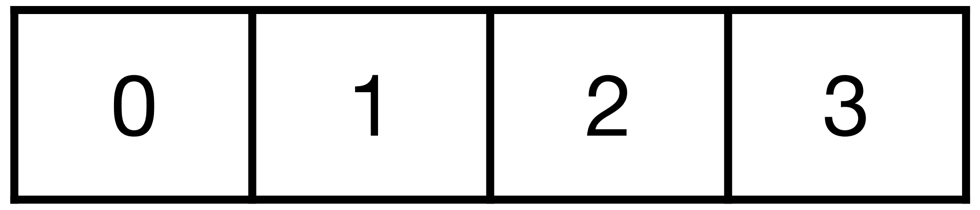
Fixed-length arrays if all dimensions are integer constants

- `int const size = 10;` is not an integer constant!
- Can initialize using nested braces
- Can omit the size of the first dimension when using an initializer

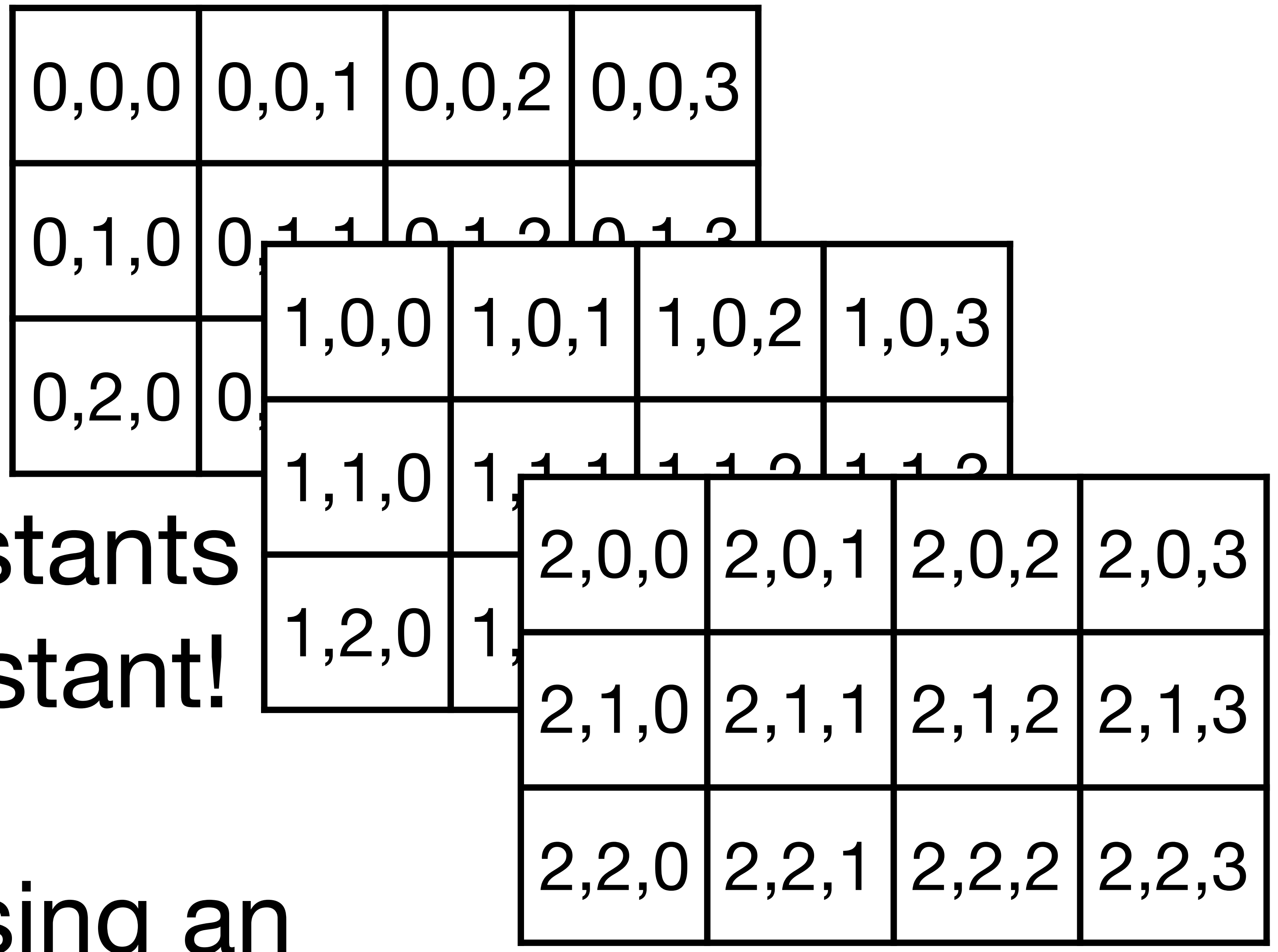
Variable-length arrays if any dimensions are not integer constants

Multidimensional arrays






```
float oneD[size];
float twoD[rows][cols];
float threeD[layers][rows][cols];
...
float general[size1][size2]...[sizeN];
```



Fixed-length arrays if all dimensions are integer constants

- `int const size = 10;` is not an integer constant!
- Can initialize using nested braces
- Can omit the size of the first dimension when using an initializer

Variable-length arrays if any dimensions are not integer constants

Passing arrays to functions

Passing arrays to functions

Remember: No array values, arrays **decay** to pointers so parameters must be pointers

Passing arrays to functions

Remember: No array values, arrays **decay** to pointers so parameters must be pointers

Three ways to declare the same function

```
void foo(size_t len, int *arr);
void foo(size_t len, int arr[]);
void foo(size_t len, int arr[len]);
```

Passing arrays to functions

Remember: No array values, arrays **decay** to pointers so parameters must be pointers

Three ways to declare the same function

```
void foo(size_t len, int *arr);
void foo(size_t len, int arr[]);
void foo(size_t len, int arr[len]);
```

For a multi-dimensional array, it's similar

```
void bar(size_t rows, size_t cols, int (*arr)[]);
void bar(size_t rows, size_t cols, int (*arr)[cols]);
void bar(size_t rows, size_t cols, int arr[][cols]);
void bar(size_t rows, size_t cols, int arr[rows][cols]);
```

Passing arrays to functions

Remember: No array values, arrays **decay** to pointers so parameters must be pointers

Three ways to declare the same function

```
void foo(size_t len, int *arr);
void foo(size_t len, int arr[]);
void foo(size_t len, int arr[len]);
```

Pointer to an array

For a multi-dimensional array, it's similar

```
void bar(size_t rows, size_t cols, int (*arr)[]);
void bar(size_t rows, size_t cols, int (*arr)[cols]);
void bar(size_t rows, size_t cols, int arr[][cols]);
void bar(size_t rows, size_t cols, int arr[rows][cols]);
```

Passing arrays to functions

Remember: No array values, arrays **decay** to pointers so parameters must be pointers

Three ways to declare the same function

```
void foo(size_t len, int *arr);
void foo(size_t len, int arr[]);
void foo(size_t len, int arr[len]);
```

Pointer to an array

For a multi-dimensional array, it's similar

```
void bar(size_t rows, size_t cols, int (*arr)[]);
void bar(size_t rows, size_t cols, int (*arr)[cols]);
void bar(size_t rows, size_t cols, int arr[][cols]);
void bar(size_t rows, size_t cols, int arr[rows][cols]);
```

Dynamically allocating multi-D arrays

Dynamically allocating multi-D arrays

Allocate a 1D array and perform index calculations manually

Dynamically allocate an `int[rows][cols]`

```
int *arr = malloc(sizeof(*arr) * rows * cols);
```

We can't use `arr[r][c]` because the type is wrong, instead use `arr[r*cols + c]`

Dynamically allocating multi-D arrays

Allocate a 1D array and perform index calculations manually

Dynamically allocate an `int[rows][cols]`

```
int *arr = malloc(sizeof(*arr) * rows * cols);
```

We can't use `arr[r][c]` because the type is wrong, instead use `arr[r*cols + c]`

Pro: By far the most common method of dealing with multi-D arrays
Con: Indexing is error prone
Con: Can't pass `arr` and some other `int arr2[rows][cols]` to the same function because they have different types

We have a 1D array of floats representing a 2D array where we're keeping track of the indices manually.

```c
size_t rows, cols; // Assume these have values
float *arr = malloc(sizeof(float)*rows*cols);
```

What is the expression for the 10th column of the 8th row? (I.e., we want "arr[8][10]" but we can't actually use that because arr is 1D.)

A. arr[10*8]
B. arr[8*rows + 10]
C. arr[8*cols + 10]
D. arr[10*rows + 8]
E. arr[10*cols + 8]

We have a 1D array of floats representing a 2D array where we're keeping track of the indices manually.

```c
size_t rows, cols; // Assume these have values
float *arr = malloc(sizeof(float)*rows*cols);
```

What is the expression for the 10th column of the 8th row? (I.e., we want "arr[8][10]" but we can't actually use that because arr is 1D.)

A. arr[10*8]
B. arr[8*rows + 10]
C. arr[8*cols + 10]
D. arr[10*rows + 8]
E. arr[10*cols + 8]

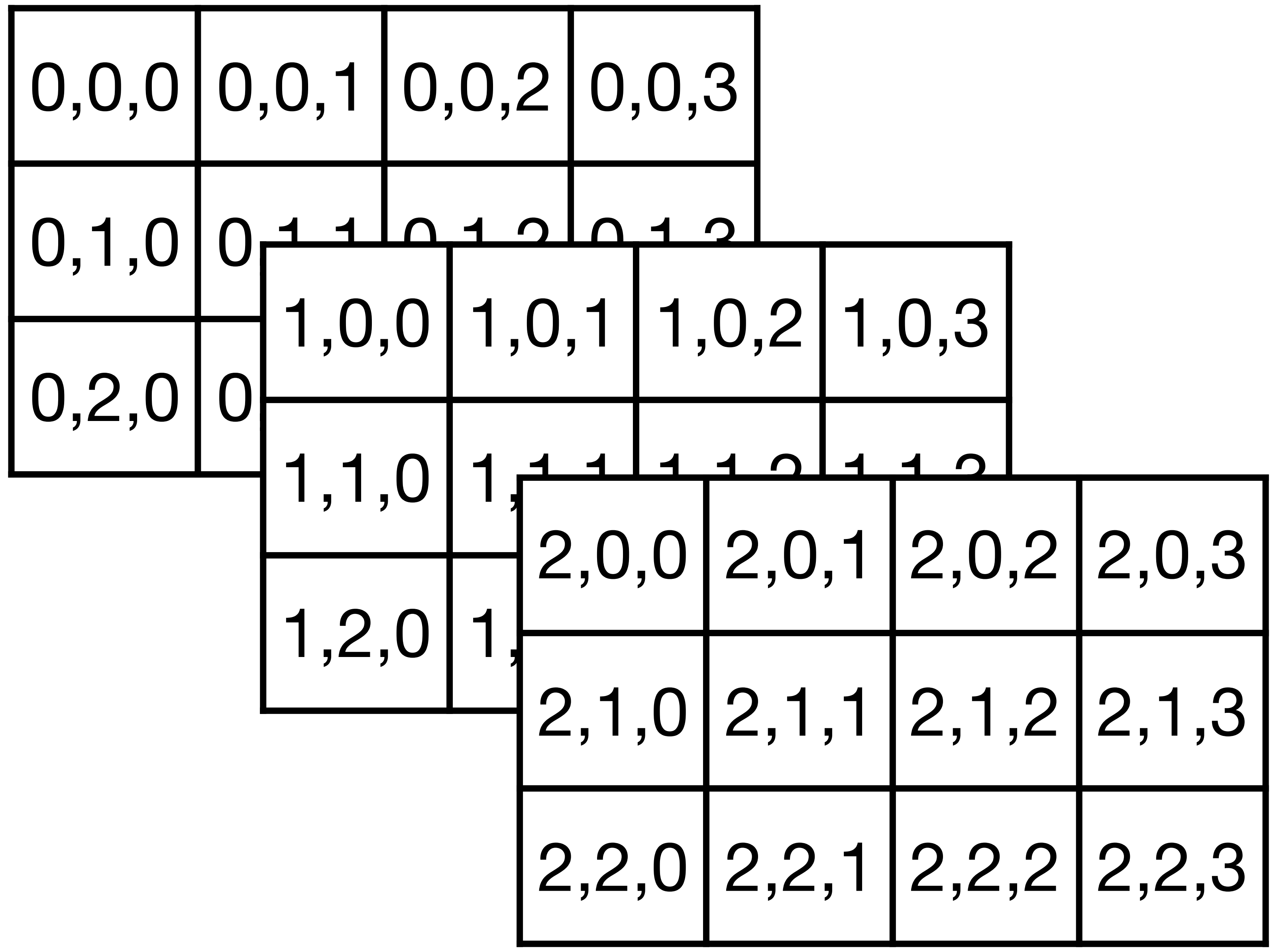
0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

We have a 1D array of floats representing a 3D array where we're keeping track of the indices manually.

```c
size_t layers, rows, cols; // Assume these have values
float *arr = malloc(sizeof(float)*layers*rows*cols);
```

What is the expression for the 3rd column of the 4th row of the 5th layer?
(I.e., we want "arr[5][4][3]" but we can't actually use that because arr is 1D.)

A. arr[5*layers + 4*rows + 3*cols]
B. arr[5*rows*cols + 4*cols + 3]
C. arr[5*layers*rows + 4*rows + 3]



Dynamically allocating multi-D arrays

Allocate the multi-dimensional array and let the compiler deal with indexes

Dynamically allocate an `int[rows][cols]`

```
int (*arr)[cols] = malloc(sizeof(int[rows][cols]));
```

Now we can just use `arr[r][c]` to access an element!

Pro: Convenient array indexing
Pro: Can use the same function for local and dynamic multi-D arrays
Con: Hideous syntax!
Con: Returning them from functions requires *very* unusual syntax and really only works with fixed-length arrays or 2D variable-length arrays

Returning dynamic arrays

For the 1-D case (as well as faking multi-D with 1-D), just return a pointer

```
int *bar(void);
```

For the 2-D case, we have some more horrible syntax

```
int (*new_array(size_t rows, size_t cols))[] {
  int (*arr)[cols] = malloc(sizeof(int[rows][cols]));
  for (size_t r = 0; r < rows; ++r) {
    for (size_t c = 0; c < cols; ++c)
      arr[r][c] = r + c;
  }
  return arr;
}
```

More than 2-D is worse

Aside about alignment

Types have a size and an **alignment**

Alignment constrains where in memory a variable can reside

- Alignment is a property of the underlying architecture
- An alignment of n means that the address of the variable must be a multiple of n

There's an **`alignof`** operator that works like **`sizeof`** except it returns the alignment of a variable or type

- This is almost never needed in real code

If an `int` has an alignment of 4 (which is common), which of the following address is **not** valid for a variable of type `int`?

A. 0x1234

B. 0x248A

C. 0x333C

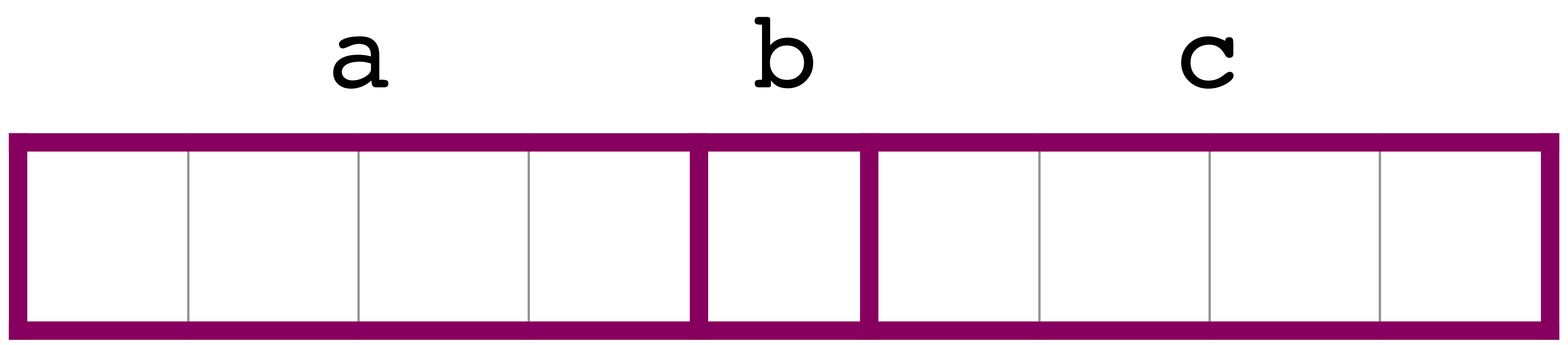
D. 0x4440

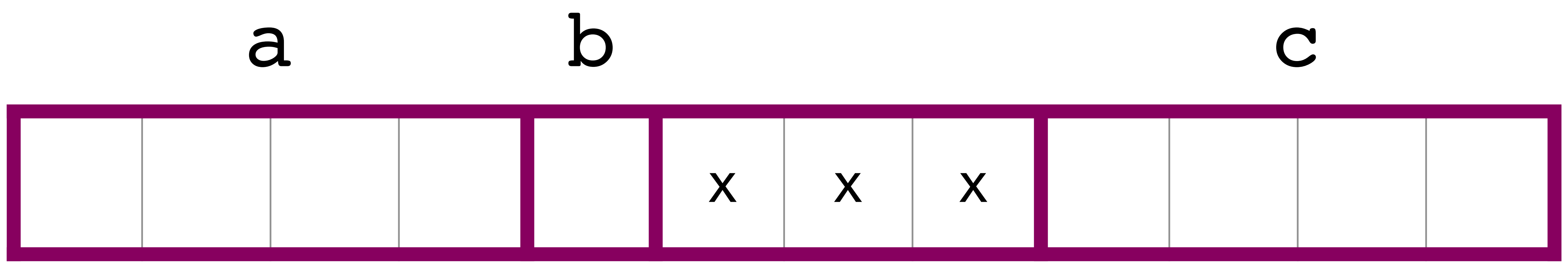
E. 0xA2D8

Padding in structs

```
struct foo {
  int a;
  char b;
  int c;
};
```

This can't be laid out in memory like this because of the alignment of a and c



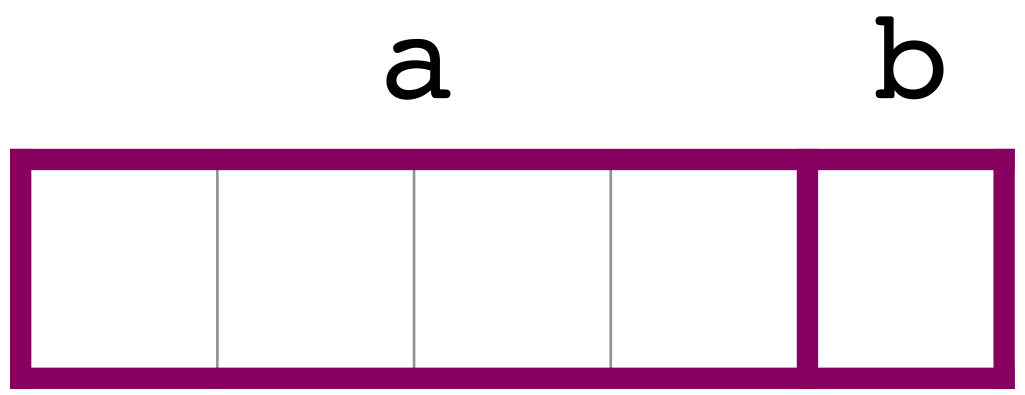


It needs **padding** bytes

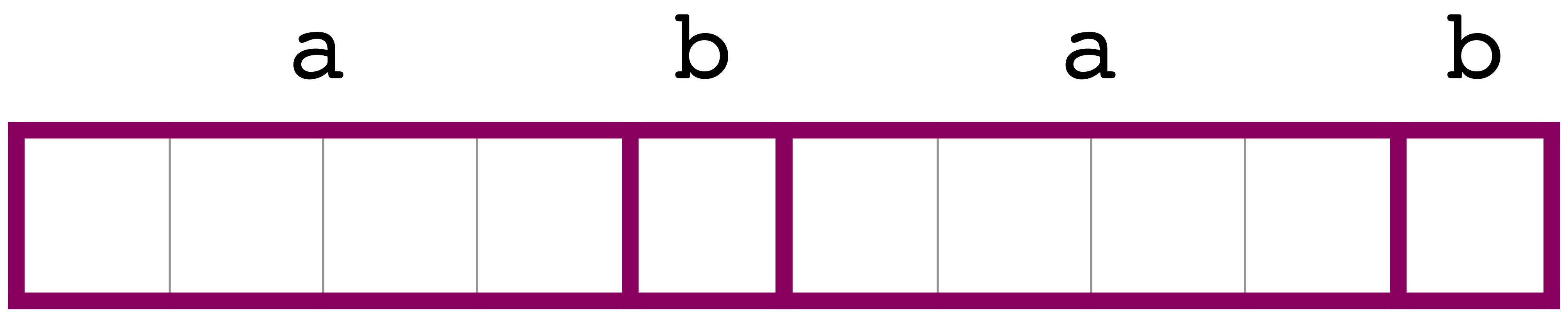
The alignment of a struct is the maximum of the alignments of its members

Array of structs

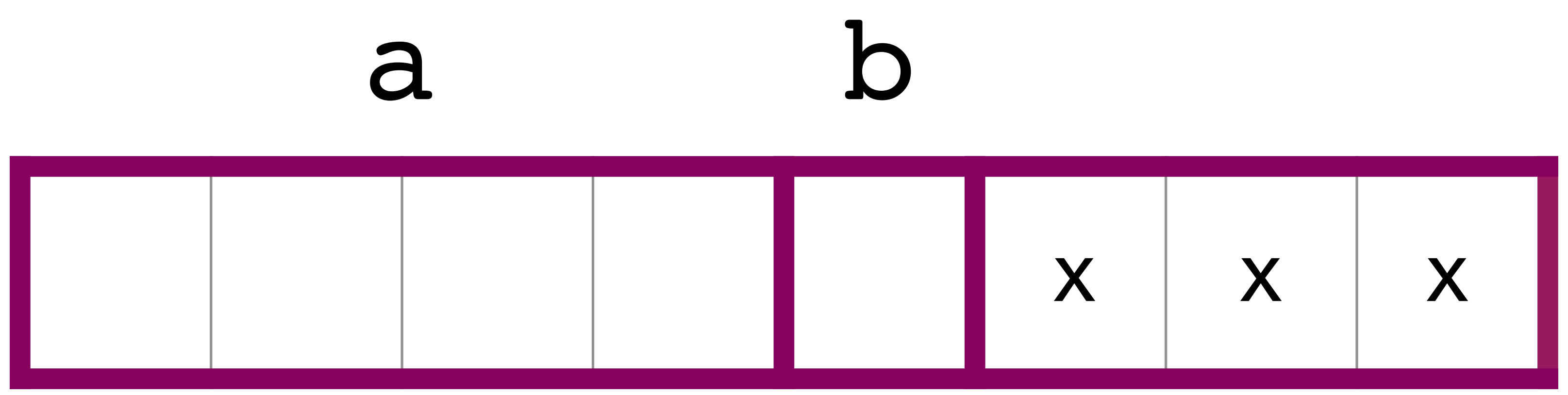
```
struct bar {
  int a;
  char b;
};
```



This can't be laid out in memory like this because of the alignment of a in subsequent elements of the array



It needs **padding** bytes at the end



What can we say about the sizes of these two structures (assuming `alignof(int) > alignof(char)`)?

```
struct s1 {
  char ch1;
  int x;
  char ch2;
};
```

```
struct s2 {
  char ch1;
  char ch2;
  int x;
};
```

A. `struct s1` is larger than `struct s2`

B. `struct s2` is larger than `struct s1`

C. Both are the same size

D. Sizes are implementation defined so there's no way to know

E. It's impossible for `alignof(int)` to be greater than `alignof(char)`

In-class exercise

https://checkoway.net/teaching/cs241/2020-spring/exercises/Lecture-22.html

Grab a laptop and a partner and try to get as much of that done as you can!