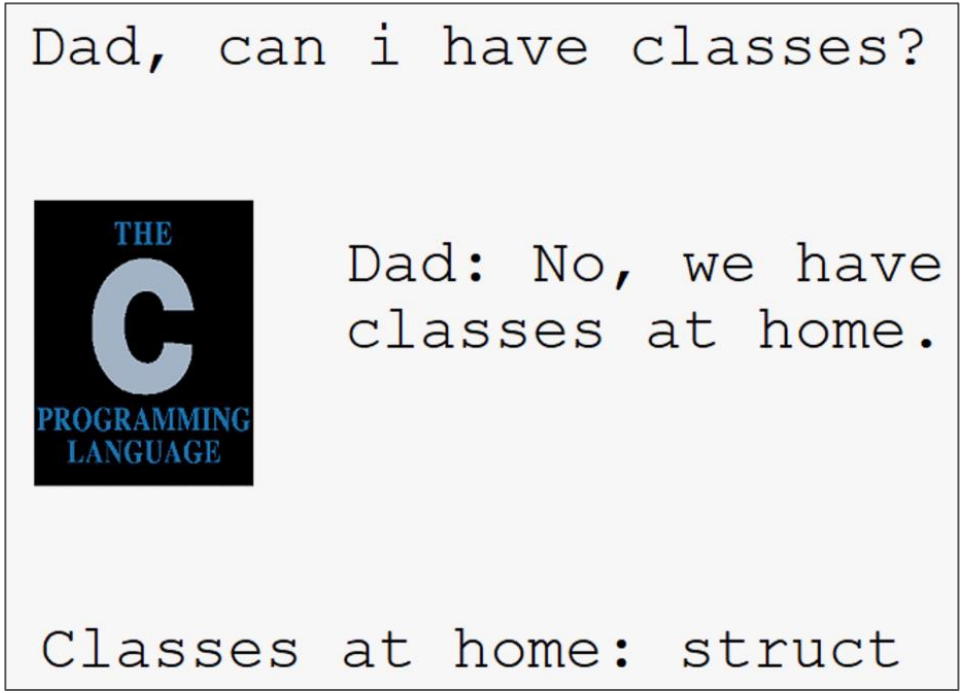
Arrays, Structs & Alignment

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang

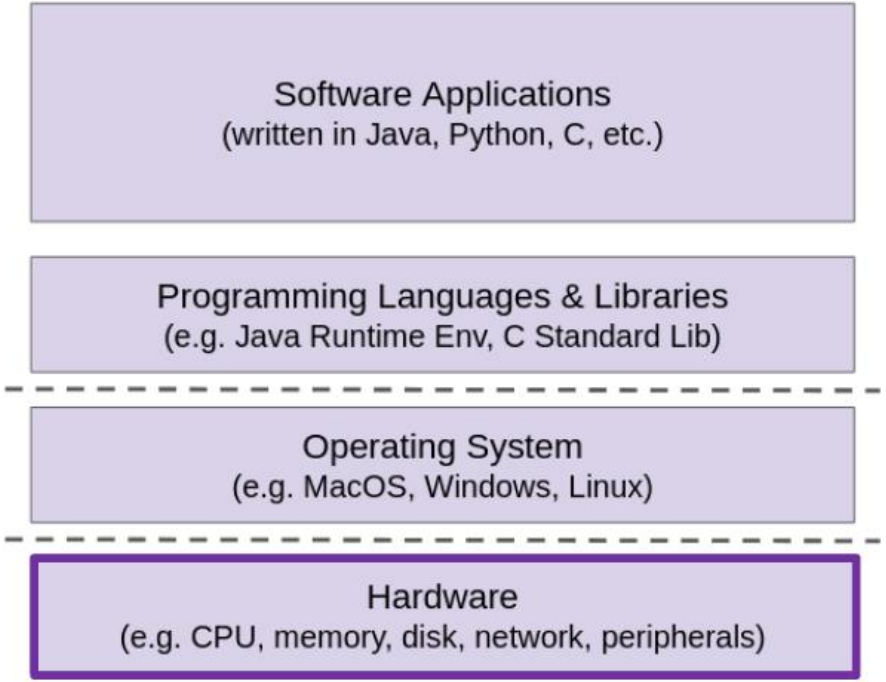


Administrivia

- Today
  - HW10 due (11:59pm)
  - Mid-Quarter Survey out on Canvas
    - Part of your participation grade
- Wednesday, 7/17
  - RD13 due (1pm)
  - HW11 due (11:59pm)
  - **Mid-Quarter Survey due (11:59pm)**
- Friday, 7/19
  - RD14 due (1pm)
  - HW12 due (11:59pm)
  - **Lab2 due (11:59pm)**
- **Quiz 2 starts next Monday!**
  - Due the following week

Layers of Computing Revisited

- Back to Hardware for today
  - How are compound data types (arrays, structs) stored in memory?
  - How do we get individual elements/fields out of them?
- Why now?
  - Knowing a little about assembly will help you understand this
  - Will be helpful for future lectures



Review Questions

Both questions questions use the following code:

```
struct ll_node {
    long data;
    struct ll_node* next;
} n1, n2;
```

8B 8B
data | next = 16B total

ptr type

"." is for an instance, "->" is for pointers

1. How much space (in bytes) does one instance of `struct ll_node` take?
2. Which of the following statements are syntactically valid?

A) `n1.next = &n2;` ✓

B) `n2->data = 351;` n2 is an instance, should be .

C) `n1.next->data = 333;` ✓

D) `(&n2)->next->next.data = 451;` ✗

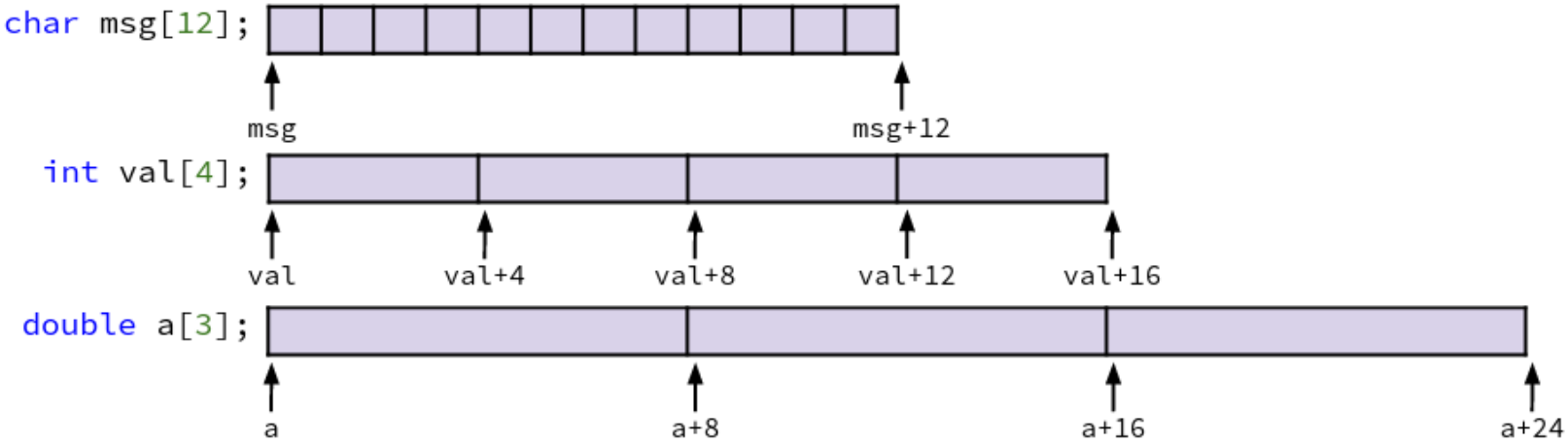
next is a pointer, should be →

Lecture Topics

- **Arrays**
  - **Array review**
  - **Arrays in C**
  - Multidimensional (nested) arrays
  - Multilevel arrays
- Structs
  - Structs in C
  - Struct memory layout
  - Alignment

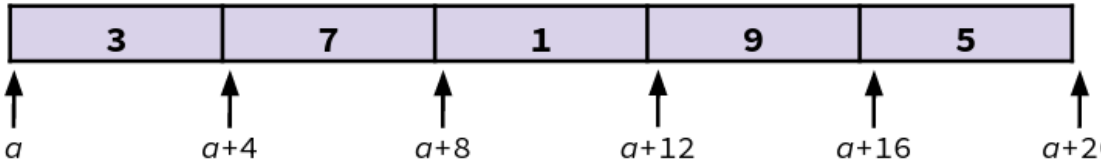
Recap: Arrays

- `T A[N]` → array `A` of type `T` and length `N`
  - Contiguously allocated region of `N*sizeof(T)` bytes
  - Identifier `A` evaluates to the address of the array (type `T*`)



Recap: Array Accesses

```
int x[5];
```



a = starting address of x

Expression	Type	Value
x	int*	a
x[4]	int	5
x + 1	int*	a + 4
&x[2]	int*	a + 8
*(x + 1)	int	7
x[5]	int	??? whatever is at address at 20

address of x[2]

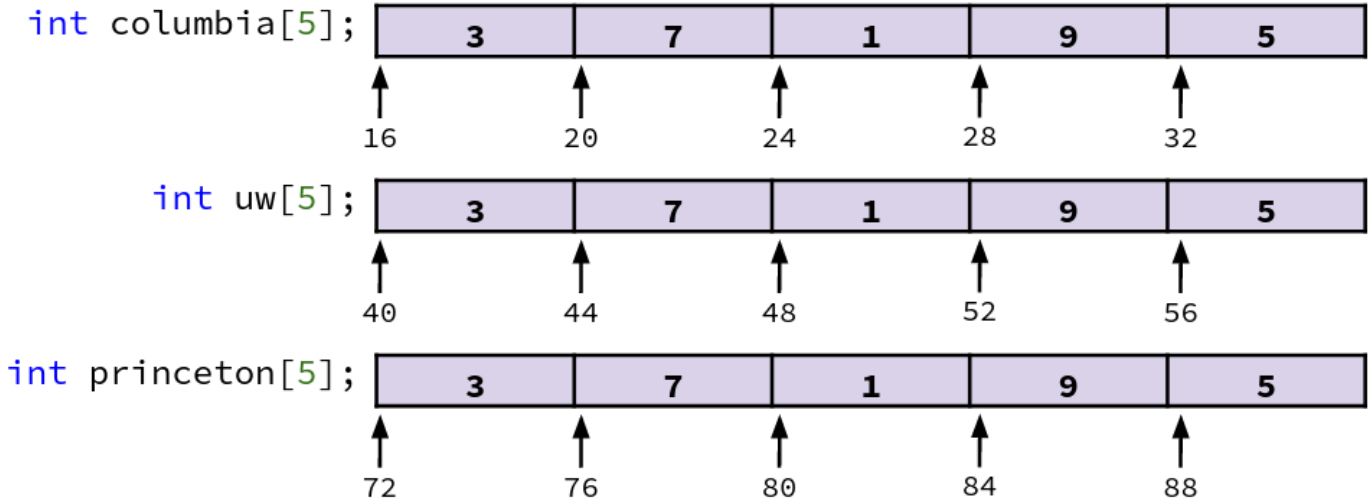
same as x[1]

Arrays in Memory Example

```
// arrays of ZIP code digits
int columbia[5]   = { 1, 0, 0, 2, 7 };
int uw[5]         = { 9, 8, 1, 9, 5 };
int princeton[5]  = { 0, 8, 5, 4, 0 };
```

Initialization list

- Each array is contiguous, but multiple arrays are *not guaranteed* to be contiguous with each other!



C Details: Arrays and Pointers

- Array variables are *almost* identical to pointers
  - `char* string` and `char[] string` are nearly identical declarations
  - Subtle differences: initialization, `sizeof()`, etc.
- An array name is an *expression*, **not** a variable!
  - Evaluates to the address of the first (0^{th}) element
  - Translates to a label in assembly
    - Doesn't store the address in a variable (unlike a pointer)
    - Read-only, can't re-assign array name

C Details: Arrays and Functions

- Allocated on the stack, only allocated while scope is valid
  - Ex: What's wrong with this code?

```
char* foo() {
        char string[32];
        . . .
        return string;
}
```

- An array is passed into a function as a pointer
  - Can no longer use `sizeof()`!
  - Ex:

```
int foo(int ar[], unsigned int size)
```

Really just an `int*`

Need to pass in the size!

Lecture Topics

- **Arrays**
  - Array review
  - Arrays in C
  - **Multidimensional (nested) arrays**
  - Multilevel arrays
- Structs
  - Structs in C
  - Struct memory layout
  - Alignment

Nested Array Example

can think of as an "array of arrays"

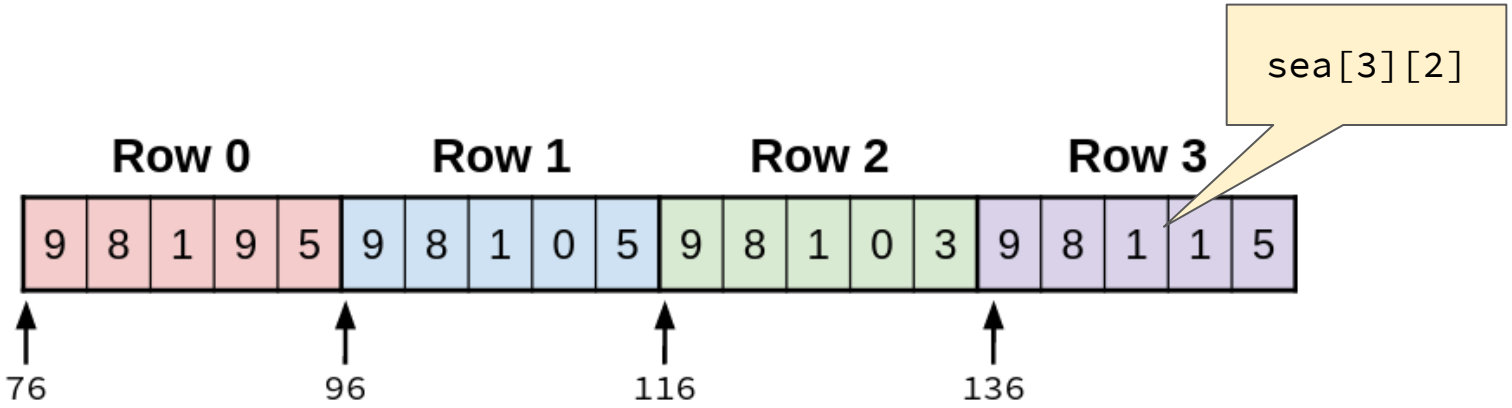
```
int sea[4][5] =
    {{ 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 }};
```

- **Multidimensional** (i.e. "**nested**") array
  - sea evaluates to an `int**`
- What's the layout in memory?

Nested Array Example (pt 2)

```
int sea[4][5] =
    {{ 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 }};
```

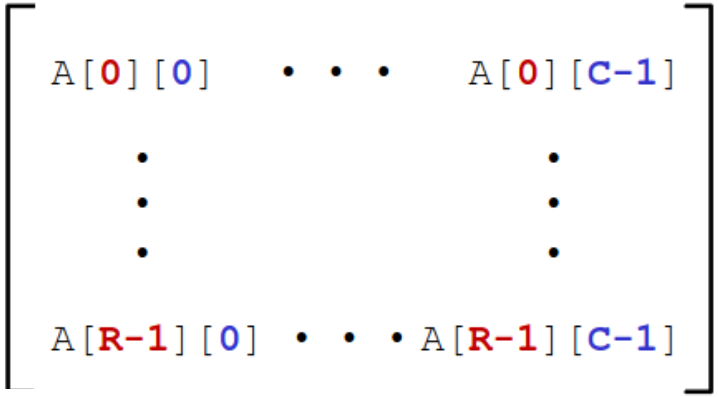
- **Row-major order**: each row stored contiguously
  - Guaranteed (in C)



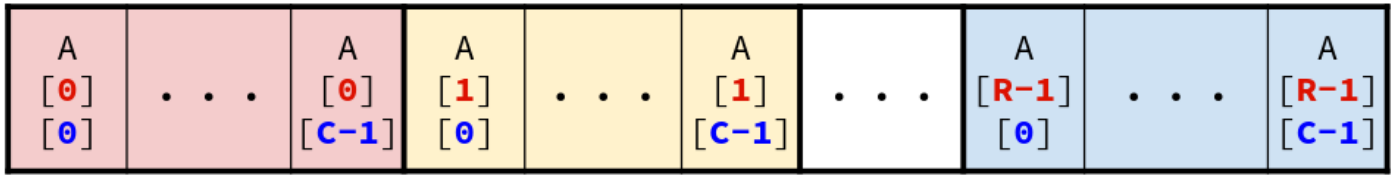
Multi-Dimensional (Nested) Arrays

- Declaration: `T A[R][C];`
  - 2D array of type `T`
  - **`R`** rows, **`C`** columns
  - Each element requires `sizeof(T)` bytes
- How big is this array?
  - **`R`**`*`**`C`**`*sizeof(T)` bytes
- Arrangement: **row-major** ordering

Conceptual view:



`int A[R][C]`

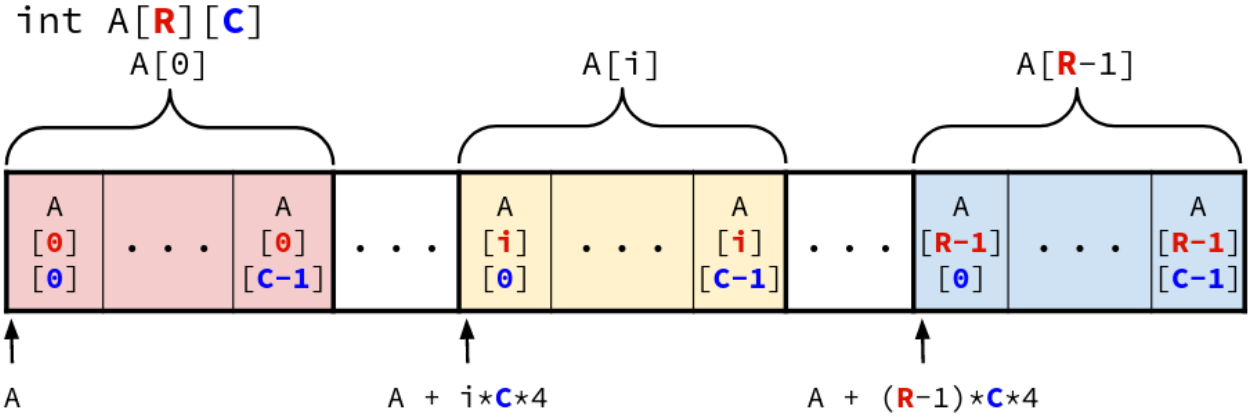


Nested Array Row Access

- Given `T A[R][C]`
  - `A[i]` is the array of elements in row `i`
  - Pointer arithmetic:
    - `A` is the address of the start of the array
    - Starting address of row `i` = `A + i*C*sizeof(T)`

C: sizeof(T) = size of a row

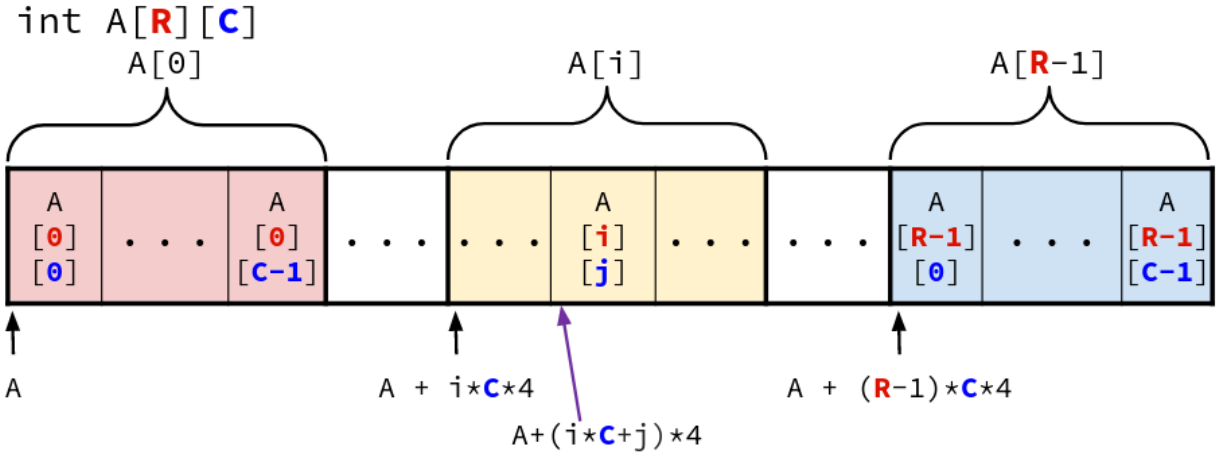
rows are contiguous, so offset of row i = i * size of row



Nested Array Element Access

- Given T A[R][C]
  - A[i][j] is element j of row i
- [i][j] = Mem[A + (i*C + j)*sizeof(T)]
  - Address of row i + offset of element j

i · C · sizeof(T) = row offset
j · sizeof(T) = offset of element w/i row
so offset of [i][j] =
i · C · sizeof(T) + j · sizeof(T)
= (i · C + j) · sizeof(T)



Lecture Topics

- **Arrays**
  - Array review
  - Arrays in C
  - Multidimensional (nested) arrays
  - **Multilevel arrays**
- Structs
  - Structs in C
  - Struct memory layout
  - Alignment

Multilevel Array

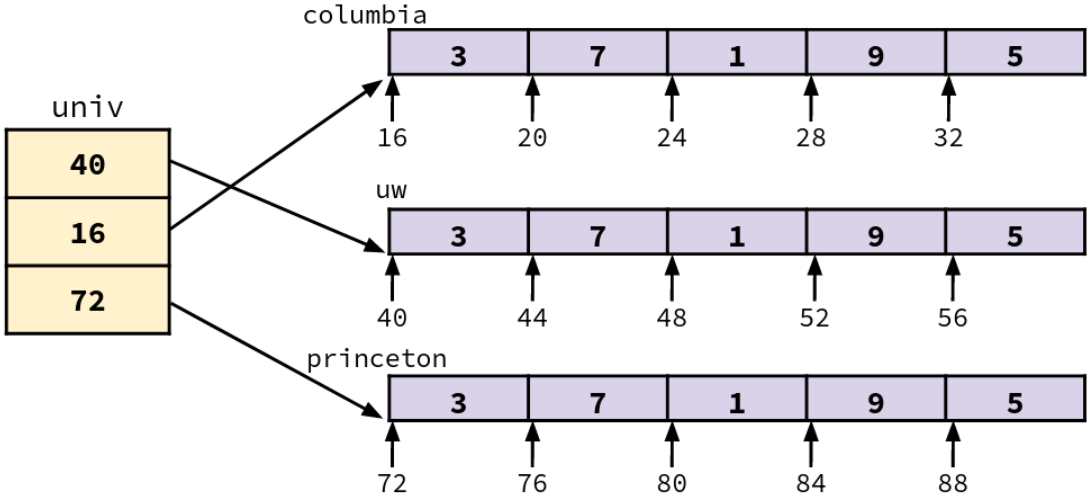
This is how Java stores "2D arrays"

```
// 1-D arrays of ints
int columbia[5]   = { 1, 0, 0, 2, 7 };
int uw[5]         = { 9, 8, 1, 9, 5 };
int princeton[5] = { 0, 8, 5, 4, 0 };
```

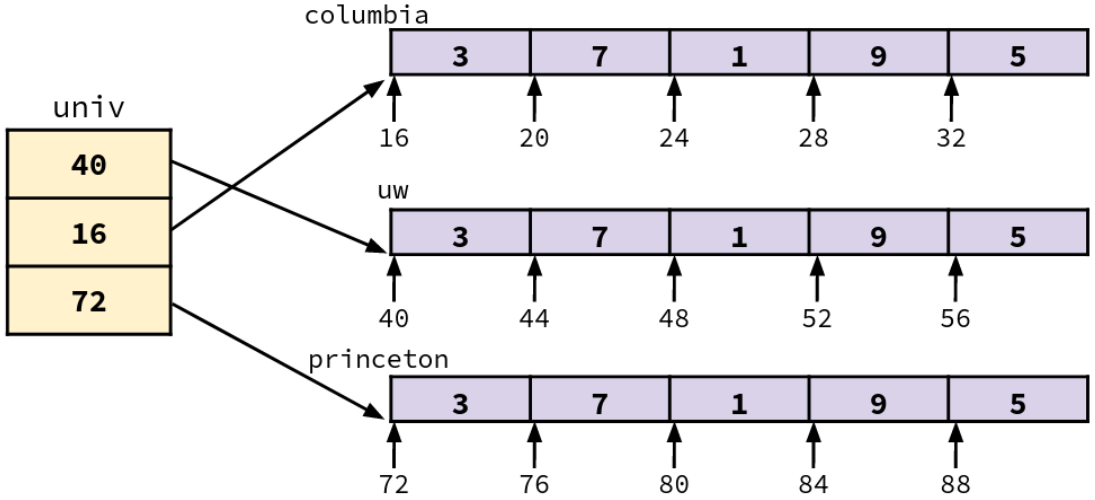
```
// Multi-level array
int* univ[3] = {uw,
columbia, princeton};
```

- Variable `univ` is an array of pointers
- Each pointer points to an array of `ints`
  - Could be different lengths!

↑ "jagged arrays"



Multilevel Array Element Access

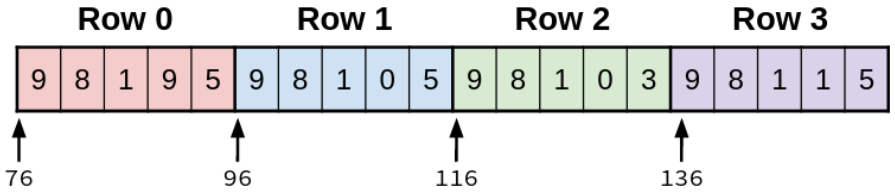


- Ex: `univ[1][3]`
  - **Requires two memory reads.** 1) to get pointer to row array. 2) to get element.
  - `Mem[Mem[univ + 1*8] + 3*4]`

TLDR: Array Element Accesses

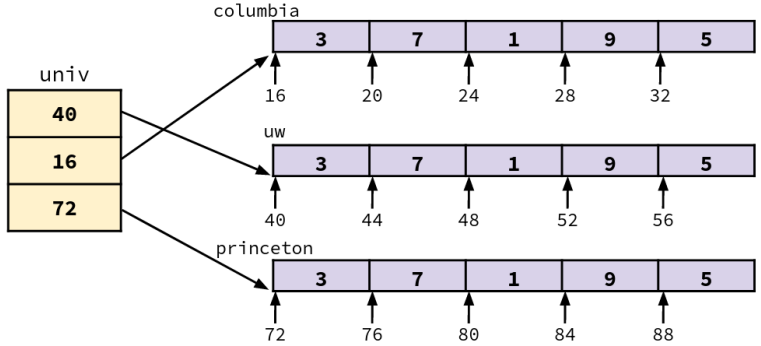
- Syntax looks the same, but memory layout is different

Multidimensional




```
A[i][j] = Mem[A+(i*C+j)*sizeof(T)]
```

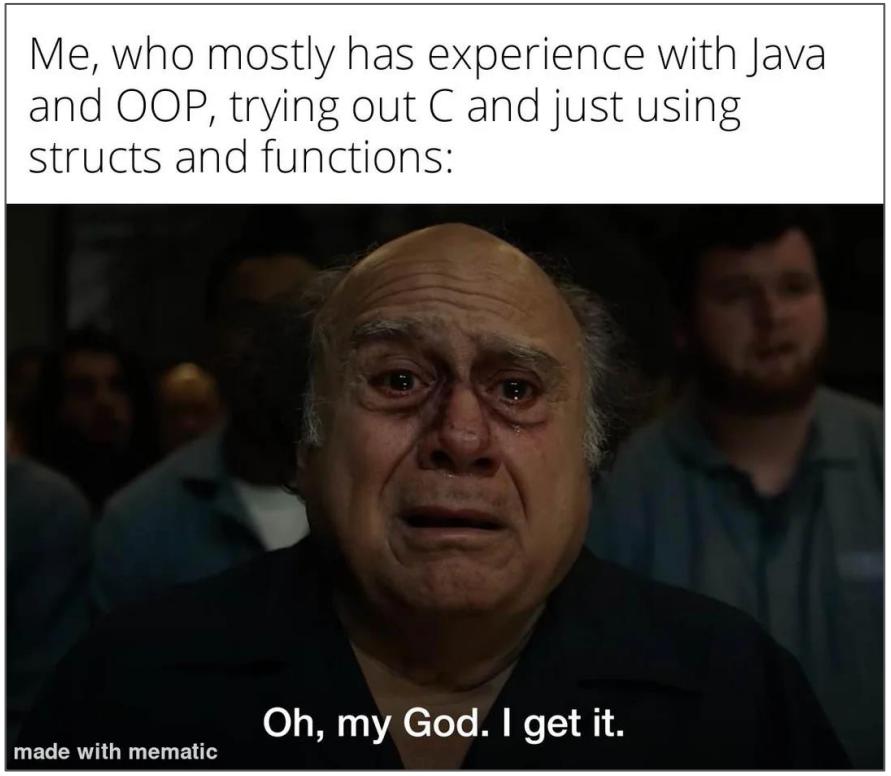
Multilevel




```
A[i][j] =
 Mem[Mem[A+i*ptr_size]+j*sizeof(T)]
```

Lecture Topics

- Arrays
  - Array review
  - Arrays in C
  - Multidimensional (nested) arrays
  - Multilevel arrays
- **Structs**
  - **Structs in C**
  - Struct memory layout
  - Alignment



Structs in C

- User-defined compound data type - structured group of variables
  - *Kinda* like a Java object, but no methods or inheritance, just fields

Example:

```
struct album {
    char* title;
    char* artist;
    int year_released;
};
```

```
struct album album1;
album1.title =  “Heavy Rocks”;
album1.artist = “Boris”;
album1.year_released = 2002;

struct album album2;
album2.title = “Heathen”;
album2.artist = “Thou”;
album2.year_released = 2014;
```

Struct Definition

```
struct album {
    char* title;
    char* artist;
    int year_released;
};
```

Remember the semicolon!

- Does **not** declare a variable, just defines the type
- Variable type is "`struct <name>`"
  - have to write "struct" every time :(
- Variable declarations are like any other data type:

```
struct album album1;      // instance
struct album* a_ptr;      // pointer
struct album playlist[5]; // array
```

- Can also combine struct definitions and instance declarations:

```
struct album {
    . . .
} a, *ptr = &a;
```

same as

```
struct album {
    . . .
};
struct album a;
struct album * ptr = &a;
```

Typedef in C

- A way to create an *alias* for another data type:
  - `typedef <data type> <alias>;`
  - Alias can be used interchangeably with the original data type
- Example:

```
typedef unsigned long ul;
        unsigned long x = 12131989;
ul x = 12131989;
```

- Can combine definition and typedef - don't have to type "struct" every time

```
struct a {
    /* fields */
};
typedef struct a album;
```

==

```
typedef struct {
    /* fields */
} album;
```

Can omit struct name

Scope of Struct Definition

- Why is the placement of a struct definition important?
  - C compiler is dumb, reads through file top⟶bottom
  - Declaring a variable creates space for it somewhere
    - Without type definition, program doesn't know how much space allocate!
- Structs follow normal C scope rules
- In practice, almost always define at the top of a file

Side note: similar rules apply to functions in C. Must define *before* they're used!

Accessing Struct Fields

```
struct album {
    char* title;
    char* artist;
    int year_released;
};
```

- Given a struct instance, use the '**.**' operator

```
struct album a1;
a1.title = "Deathconsciousness";
```

- Given a pointer to a struct:

```
struct album* p1 = &a1;
```

  - Two equivalent options:
    - Dereference, then use '**.**'

```
(*a1).artist = "Have a Nice Life";
```

    - Use '**->**'

```
p1->year_released  = 2008;
```

Aside: Java

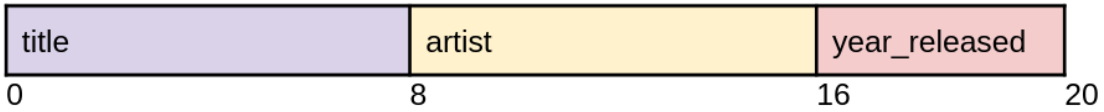
- An instance of a Java class is like a pointer ("*reference*") to a struct containing the fields
  - So Java's `x.f` is like C's `x->f`
  - Ignoring methods and inheritance, that's a future lecture!
- Arrays are similar. Allocated elsewhere, variable is a pointer to the array

Lecture Topics

- Arrays
  - Array review
  - Arrays in C
  - Multidimensional (nested) arrays
  - Multilevel arrays
- **Structs**
  - Structs in C
  - **Struct memory layout**
  - **Alignment**

Struct Representation

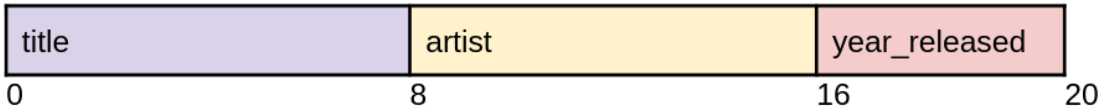
```
struct album {
    char* title;
    char* artist;
    int year_released;
};
```



- **Contiguously** allocated in memory
- Fields ordered according to declaration order
  - Even if another ordering would be more efficient!
  - Why? Easier to debug, programmer can predict what offsets fields are at
- Compiler determines size + position of fields
  - Machine-level program has no understanding of structs

Accessing Struct Fields (pt 2)

```
struct album {
    char* title;
    char* artist;
    int year_released;
} a, p = &a;
```



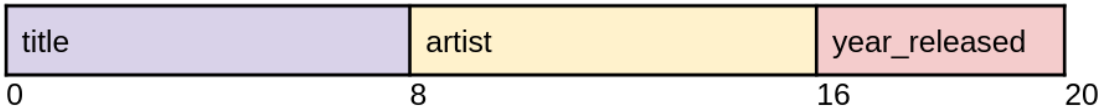
- Compiler knows the offset of each field

```
int get_year(struct album* p) {
    return p->year_released;
}
```

```
get_year:
    # pointer r in %rdi
    movl 16(%rdi), %eax
    ret
```

Pointers to Struct Fields

```
struct album {
    char* title;
    char* artist;
    int year_released;
} a, p = &a;
```



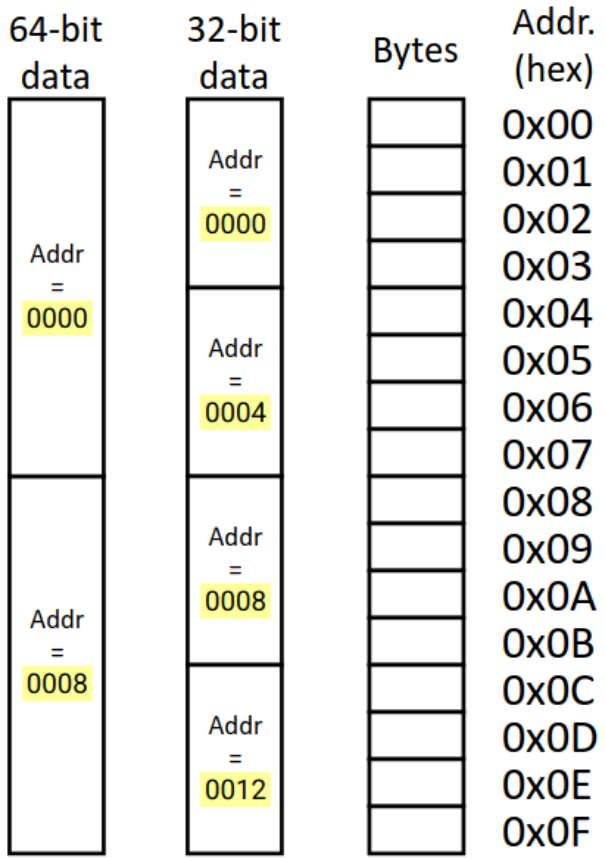
- We can get the addresses of fields within a struct!

```
int* get_year_ptr(struct album* p) {
    return &(p->year_released);
}
```

```
get_year_ptr:
    # pointer r in %rdi
    leaq 16(%rdi), %rax
    ret
```

Recap: Alignment

- Primitive-type data is stored at a starting address that is a **multiple of its size**
  - Required on some systems, advised in x86
- Why?... **Performance**
  - Allows hardware optimizations
    - Some (non-x86) hardware will not work without it
  - Speeds up memory accesses (future lecture!)



Alignments in x86-64

- Primitive of *K* bytes must have an address that is divisible by *K*
- **Useful Fact**: if a number is divisible by 2^n, then its binary representation will end in *n* 0's

Useful for Lab 5!

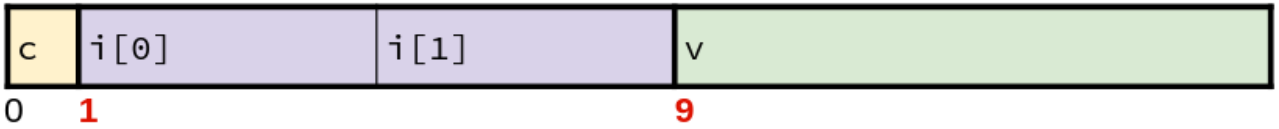
Data Type	Size (bytes)	Address
`char`	1	No restrictions
`short`	2	Lowest bit is 0: `0b…0`
`int, float`	4	Lowest 2 bits are 0: `0b…00`
`long, double`	8	Lowest 3 bits are 0: `0b…000`
`long double`	16	Lowest 4 bits are 0: `0b…0000`

Alignment in Structs

```
struct S1 {
    char c;
    int i[2];
    double v;
}
```

- Each field must be aligned to the size of its data type
  - **Internal fragmentation**: unused space *between fields* to that each field is aligned

Unaligned



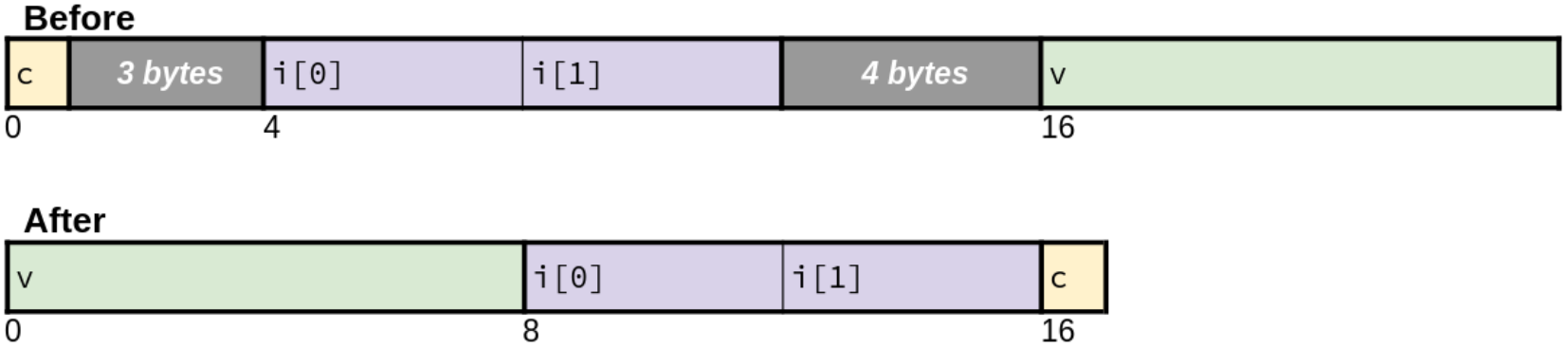
Aligned

How can we minimize **internal fragmentation**?

Alignment in Structs (pt 2)

```
struct S1 {
    double v;
    int i[2];
    char c;
}
```

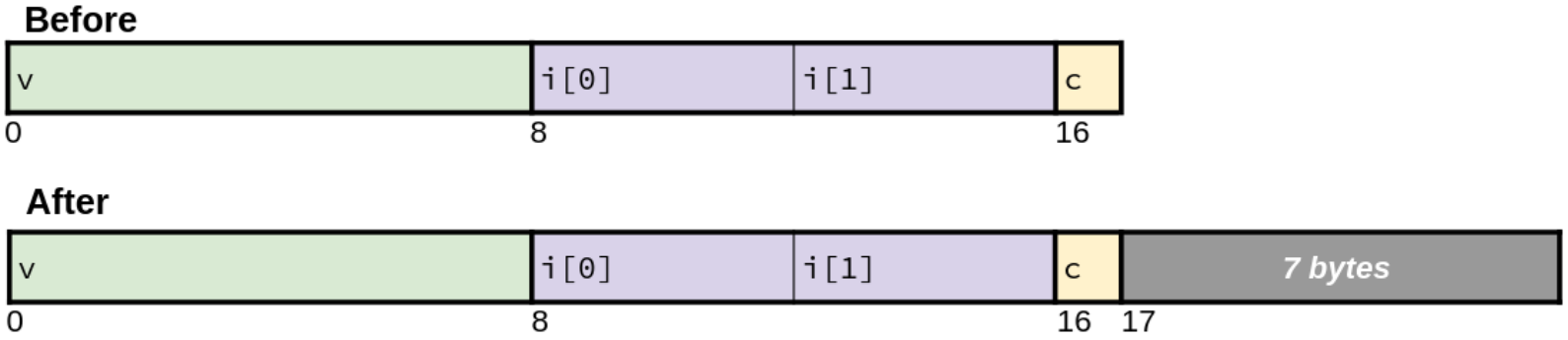
- **Internal fragmentation**: unused space *between fields* to that each field is aligned
  - Changing ordering of fields can reduce the amount of padding needed



Alignment in Structs (pt 3)

```
struct S1 {
    double v;
    int i[2];
    char c;
}
```

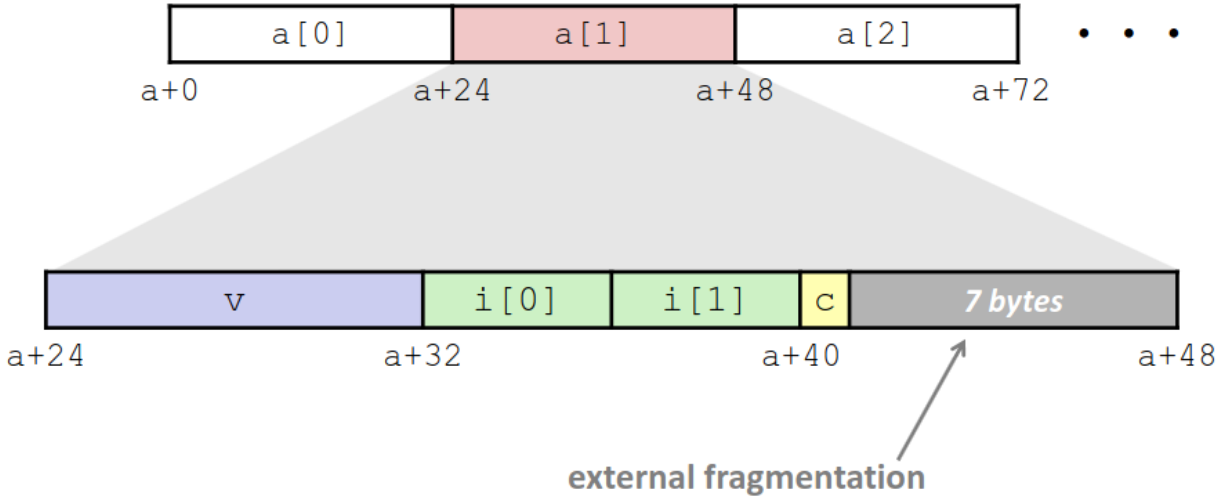
- Within struct: fields must be aligned to the size of their types
- Overall struct
  - Must be aligned to K_{max}
    - K_{max} = largest alignment requirement of any field
  - Add **external fragmentation** at the end of the struct to maintain alignment



Why External Fragmentation?

```
struct S1 {
    double v;
    int i[2];
    char c;
}
```

- Arrays of structs
  - If overall structure is aligned to K_{max}, each field will be aligned for every element in the array

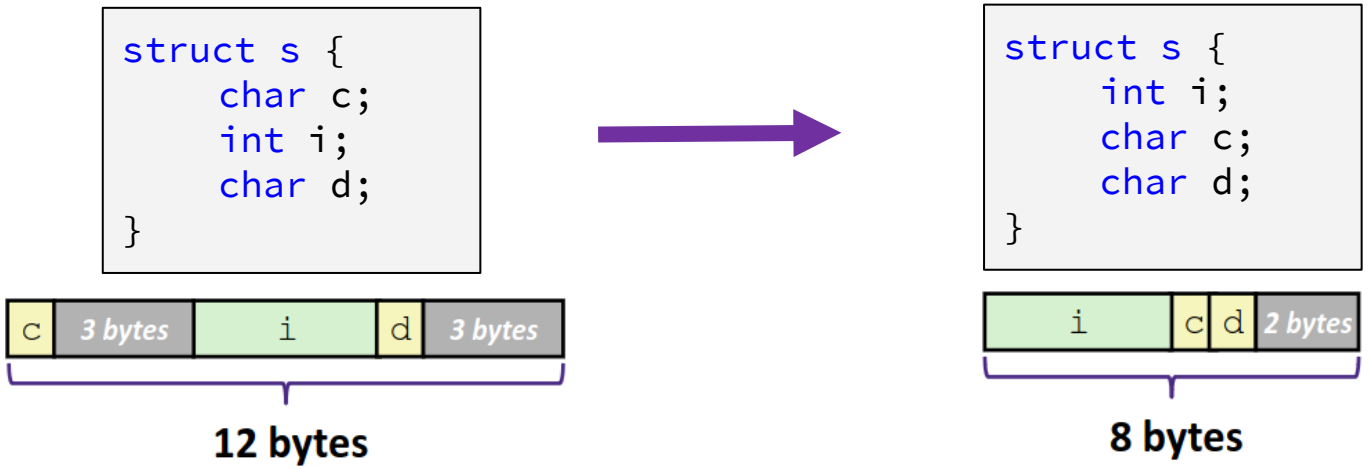


Alignment of Structs (pt 4)

- Compiler will do the following:
  - Maintain the order of fields specified in the C code
  - Add **internal fragmentation** to preserve alignment of fields
  - Add **external fragmentation** to preserve alignment of the overall struct

Programming With Structs Tips

- Because of padding, size of a struct **!=** the sum of the sizes of each field
  - Use `sizeof()` to get the true size
- To save space, **declare larger fields first**



Practice Question

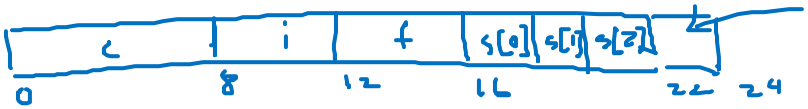
1. Minimize the size of the struct by re-ordering the fields.

```
struct old {
    int i;
    short s[3];
    char* c;
    float f;
};
```

```
struct new{
    char*  c;
    int    i;
    float  f;
    short  s[3];
};
```

8B
4B
4B
6B long, but aligned to 2

1. `sizeof(struct old)` == 32 B. What `sizeof(struct new)`?
   - A) 22 bytes
   - B) 24 bytes
   - C) 28 bytes
   - D) 32 bytes



external fragmentation

Summary: Arrays

- Contiguously allocated
- Array name evaluates to **starting address**
  - Not a variable! Becomes a label in assembly
- **Multidimensional arrays** stored in **row-major order**: `T A[R][C]`
  - `A[i]` = array of row i = `A + i*C*sizeof(T)`
  - `A[i][j]` = element j of row i = `Mem[A + (i*C + j)*sizeof(T)]`
- **Multilevel arrays** are arrays of *pointers* to other arrays: `T* A[R] = {...}`
  - `A[i]` = `Mem[A + i*sizeof(pointer)]`
  - `A[i][j]` = `Mem[Mem[A+i*sizeof(pointer)] + j*sizeof(T)]`

Summary: Structs

- User defined types containing **fields**
  - Fields allocated in order declared by programmer
- Accessing Fields
  - For an instance of the struct, use `.`
  - For a pointer, use `->`
    - Equivalent to `*` (dereference) and then `.`
- Compiler adds padding to maintain **alignment**
  - Each field must be aligned to its size: add **internal fragmentation**
  - Struct must be aligned to K_{max}: add **external fragmentation**
  - Minimize fragmentation by ordering fields from largest to smallest in C code