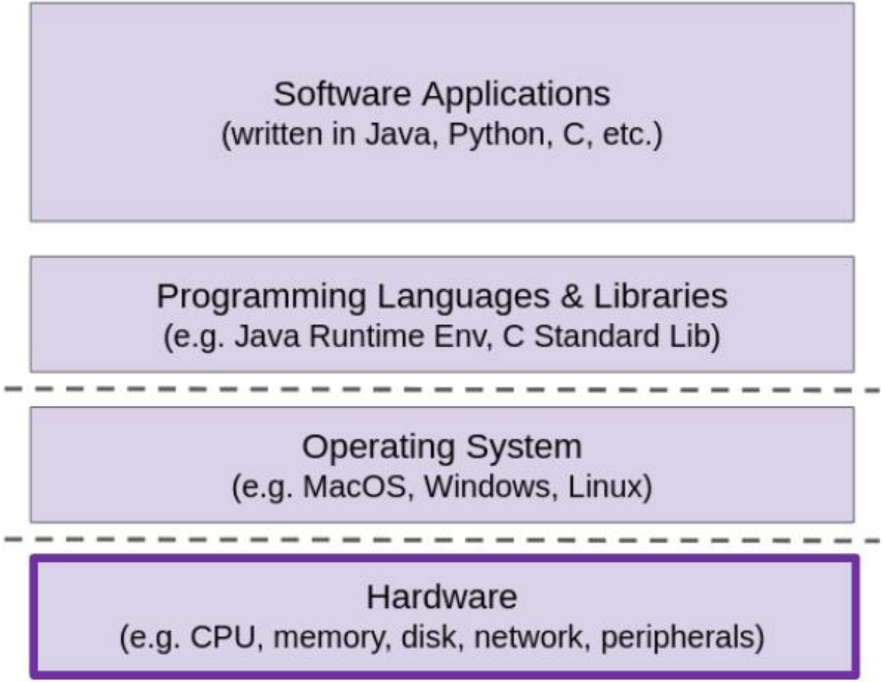
Arrays, Structs & Alignment

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



Software Applications
(written in Java, Python, C, etc.)

Programming Languages & Libraries
(e.g. Java Runtime Env, C Standard Lib)

Operating System
(e.g. MacOS, Windows, Linux)

Hardware
(e.g. CPU, memory, disk, network, peripherals)

Lecture Topics

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

Arrays start at 0

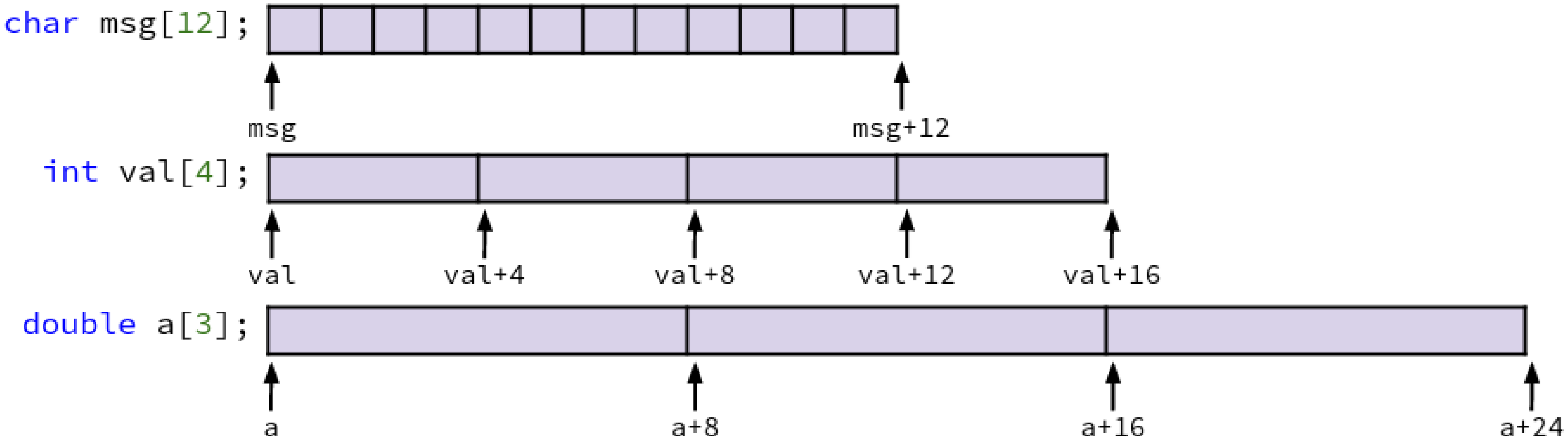
Arrays start at 1

Arrays can start wherever ¯_(ツ)_/¯

Arrays start at 4, stop at 6, restart at 1, stop again at 3, restart at 7 then continue on

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*`)

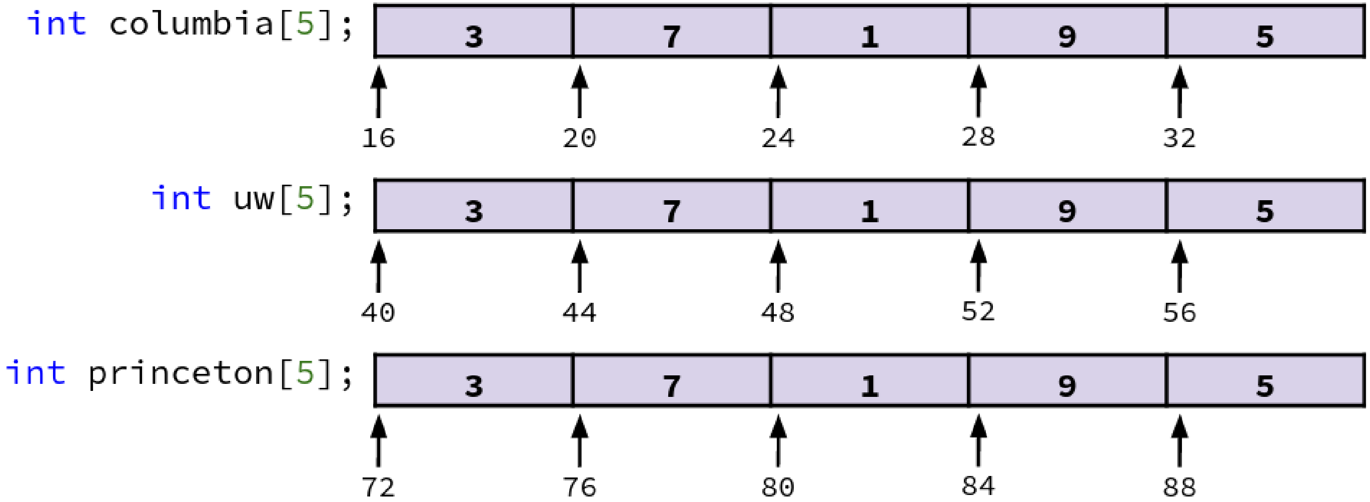


Arrays in Memory Example

```
// 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!



Nested Array Example

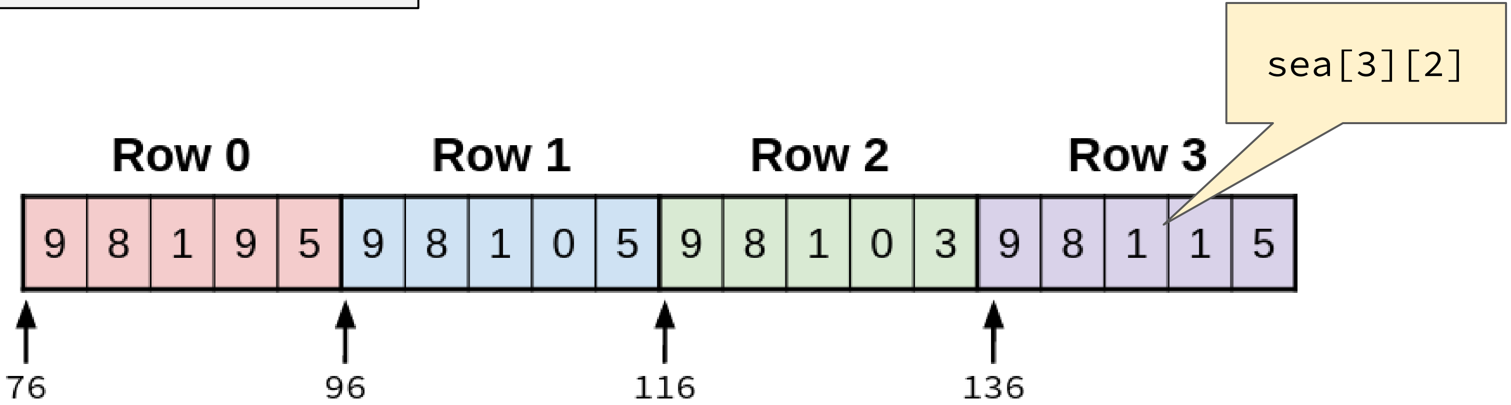
```
sea =
    [[ 9, 8, 1, 9, 5 ],
     [ 9, 8, 1, 0, 5 ],
     [ 9, 8, 1, 0, 3 ],
     [ 9, 8, 1, 1, 5 ]];
```

- **Multidimensional** (i.e. “**nested**”) array
- What’s the layout in memory?

Nested Array Example (pt 2)

```
sea =
    [[ 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)



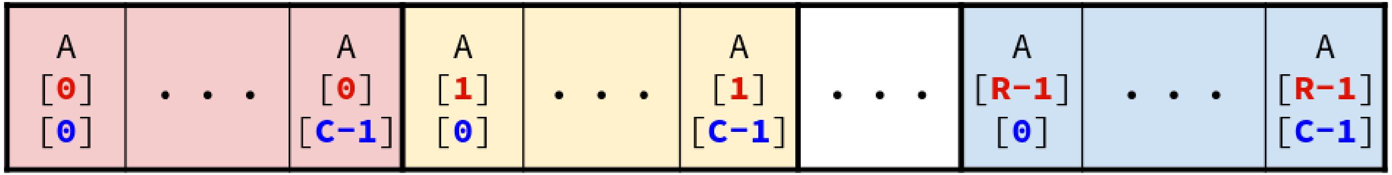
Multi-Dimensional (Nested) Arrays

- 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:

$$\begin{bmatrix} A[0][0] & \cdots & A[0][C-1] \\ \vdots & & \vdots \\ A[R-1][0] & \cdots & A[R-1][C-1] \end{bmatrix}$$

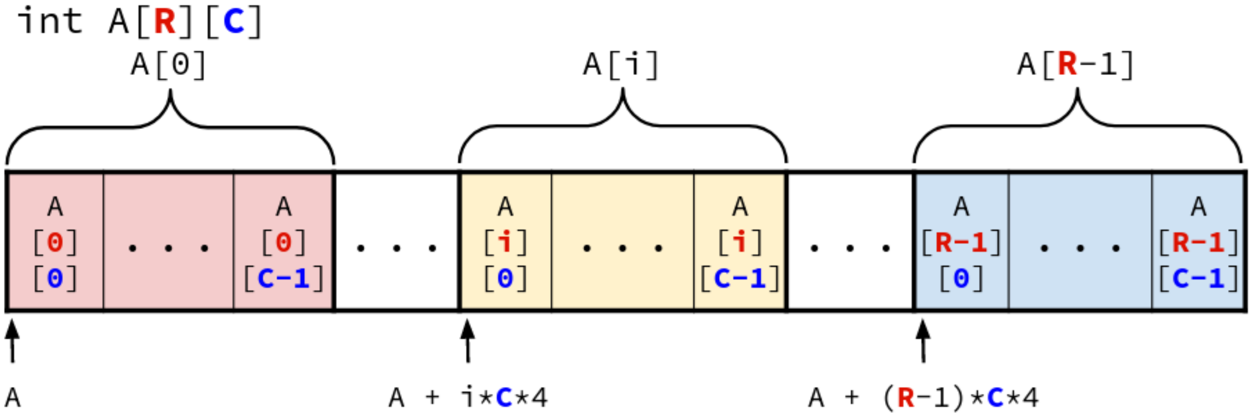
`int A[R][C]`



A [0] [0]	• • •	A [0] [C-1]	A [1] [0]	• • •	A [1] [C-1]	• • •	A [R-1] [0]	• • •	A [R-1] [C-1]

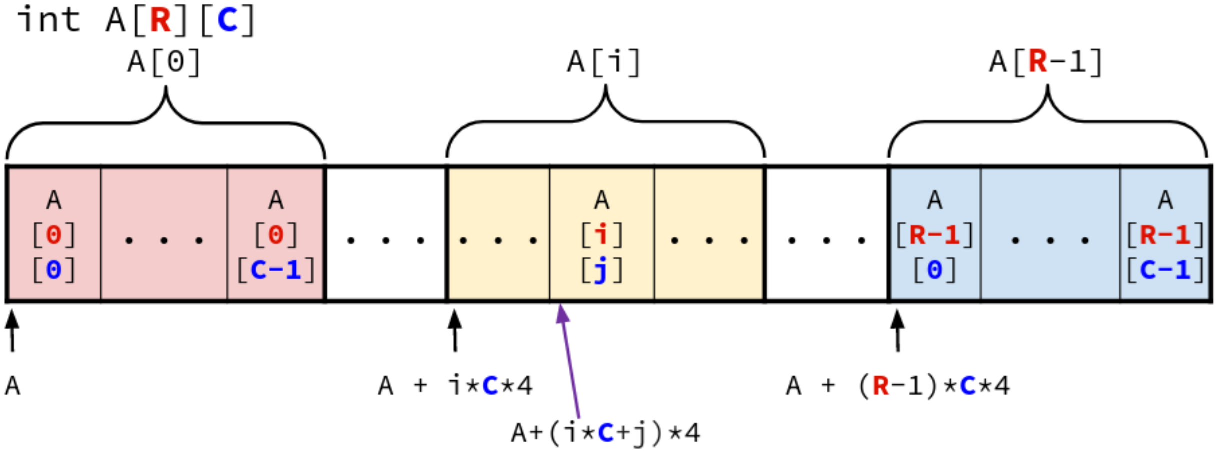
Nested Array Row Access

- 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)`



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

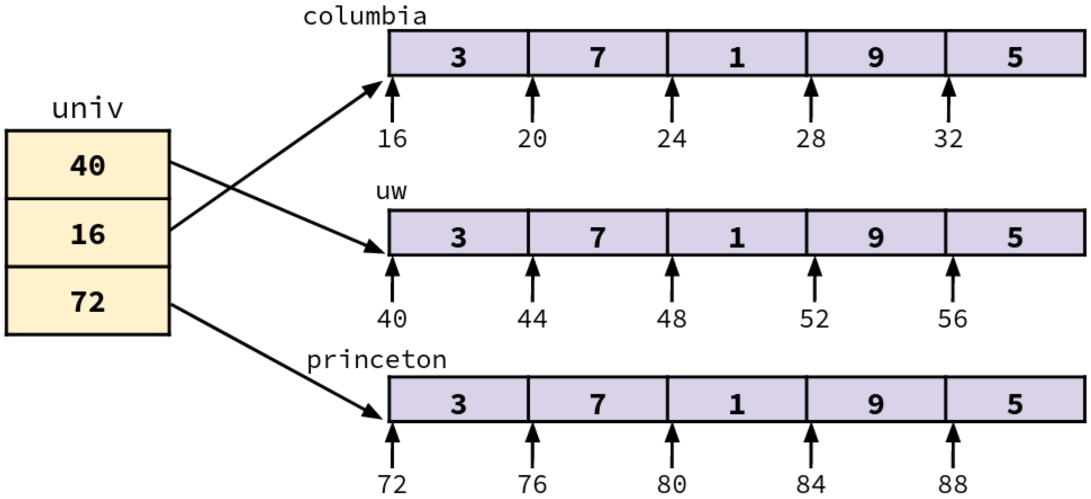


Multilevel Array

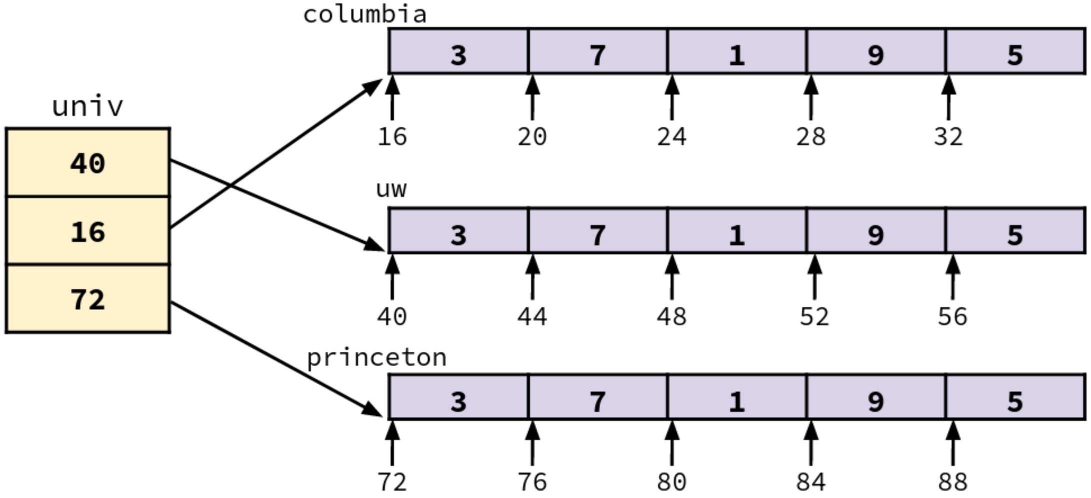
```
// 1-D arrays of ints
int columbia[5]  = { 1, 0, 0, 2, 7 };
int uw[5]        = { 9, 8, 1, 9, 5 };
int princeton[5] = { 0, 8, 5, 4, 0 };
```

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

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



Multilevel Array **Element** Access

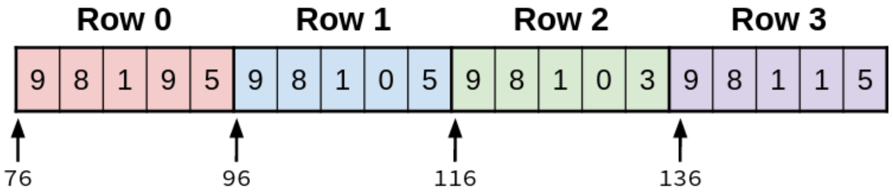


- Ex: `univ[1][3]`
  - **Requires two memory reads.** 1) to get pointer to row array. 2) to get element.
  - `Mem[Mem[univ + 1*8] + 3*4]`

TLDR: Array Element Accesses

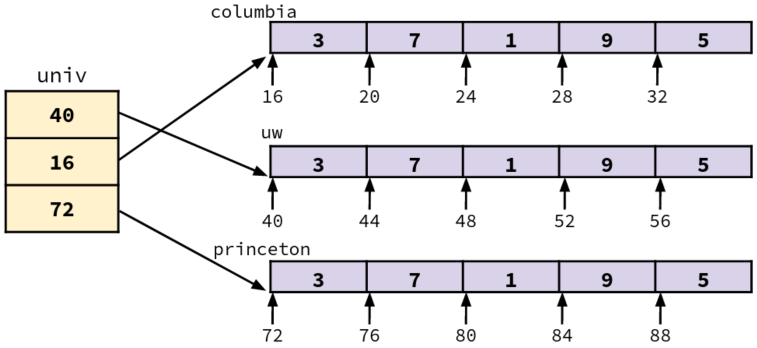
- Syntax looks the same, but memory layout is different

Multidimensional



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

Multilevel



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

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)]`