Lecture 11
More Arrays

Comp_Sci 213

Introduction to Computer Systems

Wednesday, Nov. 2, 2026

Kate Smith – Fall 2026

Slides adapted from:
Ghena, St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Topics

- Multi-dimensional Arrays
- Multi-level Arrays

Topics

- **Multi-dimensional Arrays**
- Multi-level Arrays

Review: Arrays (1D)

- Arrays in C aggregate scalar data into larger data types
- For data type T consider:

```
T A [N];
```

- Starting location will be at x_A , declaration above:
  - Allocates a contiguous region of L * N bytes in memory, L is the size in bytes of data type T
  - An identifier A is created that can be used as a pointer to the beginning of the array (value is x_A)
- Array elements can be accessed with an integer index ranging between [0 :N-1]

```
char    A[12];
char   *B[8];
int     C[6];
double *D[5];
```



Array	Element size	Total size	Start address	Element i
A	1	12	x_A	$x_A + i$
B	8	64	x_B	$x_B + 8i$
C	4	24	x_C	$x_C + 4i$
D	8	40	x_D	$x_D + 8i$

Review: Arrays (1D)

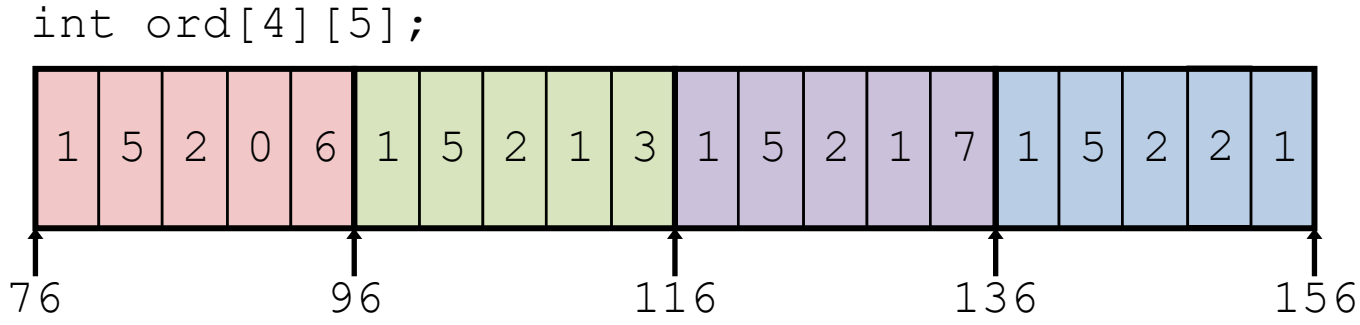
- Suppose `E` is an array of values of type `int`
- We wish to evaluate `E[i]` where:
  - E address is stored in `%rdx` and
  - i is stored in `%rcx`

```
movl (%rdx, %rcx,4), %eax
```

- The instruction above will perform the address computation x_E +4*i, read that memory location, and copy the result to `%eax`

Multidimensional (Nested) Array Example

```
int ord[4][5] =
  /* 4 rows, 5 cols */
  {{1, 5, 2, 0, 6 },
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



Addresses of each part of the array
Random starting address: decimal 76

- Let's decipher "`int ord[4][5]`"

  `int` **`ord[4]`**`[5]`: **`ord`** is an array of **4** elements, allocated contiguously

  `int` `ord[4]` **`[5]`**: Each element is an array of **5** **`int`**'s, allocated contiguously

- "Row-Major" ordering of all elements is guaranteed
  - Entire row (all columns in it) will be placed in memory before the next row starts

Multi-dimensional array in memory

Address	0	1	2	3	4	5	6	7
0x1040	2E	E2	BD	62	EF	A0	CD	93
0x1048	A4	75	61	2F	0F	DB	64	A4
0x1050	54	7A	F2	60	6E	47	B0	92
0x1058	DA	72	8F	A8	E5	15	18	CE
0x1060	86	BF	6A	6A	92	99	CF	6C

```
int array[4][2];
```

```
int array[4][2] =
  /* 4 rows, 2 cols */
  {{A, B},
   {A, B},
   {A, B},
   {A, B}};
```

- Assume array starts at address 0x1040
- How many bytes is the entire array?

Multi-dimensional array in memory

Address	0	1	2	3	4	5	6	7
0x1040	2E	E2	BD	62	EF	A0	CD	93
0x1048	A4	75	61	2F	0F	DB	64	A4
0x1050	54	7A	F2	60	6E	47	B0	92
0x1058	DA	72	8F	A8	E5	15	18	CE
0x1060	86	BF	6A	6A	92	99	CF	6C

```
int array[4][2];
```

```
int array[4][2] =
  /* 4 rows, 2 cols */
  {{A, B},
   {A, B},
   {A, B},
   {A, B}};
```

- Assume array starts at address 0x1040
- How many bytes is the entire array? 4*2*4 = 32 bytes
- Which bytes in memory does each row occupy?

Multi-dimensional array in memory

Address	0	1	2	3	4	5	6	7
0x1040	2E	E2	BD	62	EF	A0	CD	93
0x1048	A4	75	61	2F	0F	DB	64	A4
0x1050	54	7A	F2	60	6E	47	B0	92
0x1058	DA	72	8F	A8	E5	15	18	CE
0x1060	86	BF	6A	6A	92	99	CF	6C

```
int array[4][2];
```

```
int array[4][2] =
  /* 4 rows, 2 cols */
  {{A, B},
   {A, B},
   {A, B},
   {A, B}};
```

- Assume array starts at address 0x1040
- How many bytes is the entire array? 4*2*4 = 32 bytes
- Which bytes in memory does each row occupy?
  - Shaded above: **darker color is first column**, lighter color is second column
- What is the value of `array[2][1]`?

Multi-dimensional array in memory

Address	0	1	2	3	4	5	6	7
0x1040	2E	E2	BD	62	EF	A0	CD	93
0x1048	A4	75	61	2F	0F	DB	64	A4
0x1050	54	7A	F2	60	6E	47	B0	92
0x1058	DA	72	8F	A8	E5	15	18	CE
0x1060	86	BF	6A	6A	92	99	CF	6C

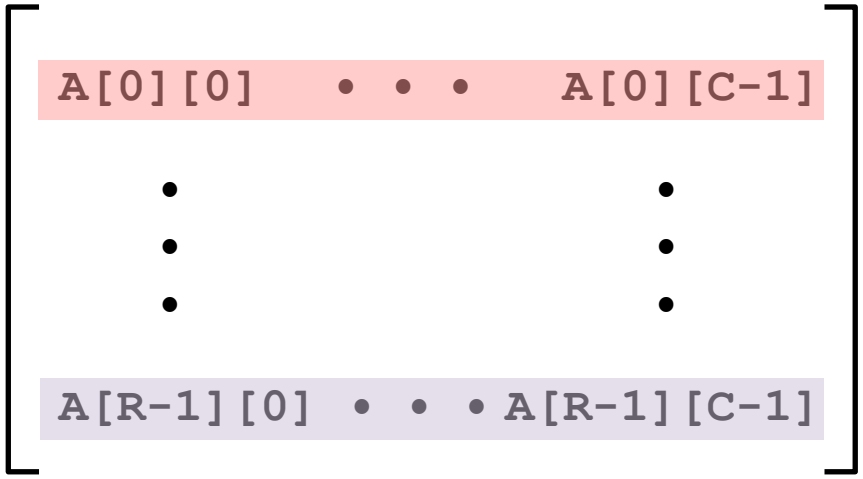
```
int array[4][2];
```

```
int array[4][2] =
  /* 4 rows, 2 cols */
  {{A, B},
   {A, B},
   {A, B},
   {A, B}};
```

- Assume array starts at address 0x1040
- How many bytes is the entire array? 4*2*4 = 32 bytes
- Which bytes in memory does each row occupy?
  - Shaded above: darker color is first column, lighter color is second column
- What is the value of `array[2][1]`? 0x92B0476E (little endian)

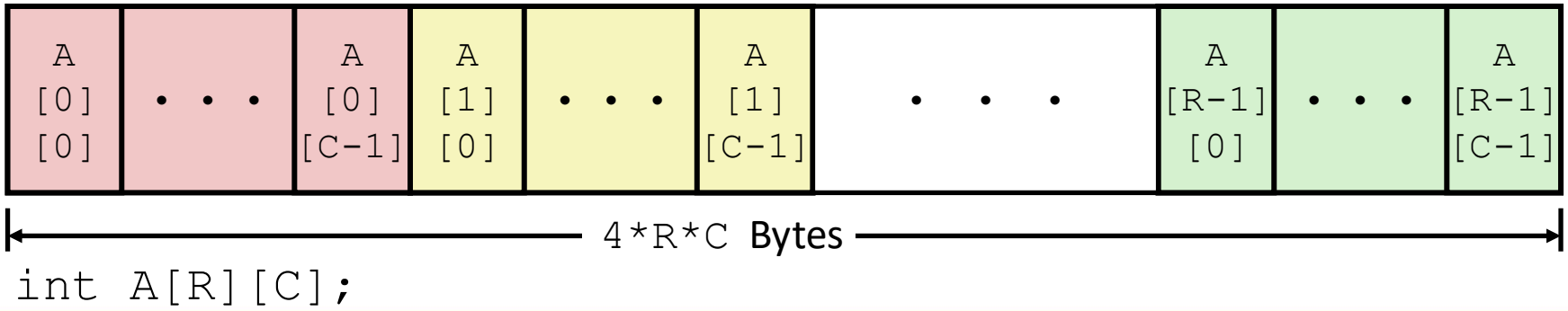
Multidimensional (Nested) Arrays

- Declaration
  - *T* **A**[*R*][*C*];
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes

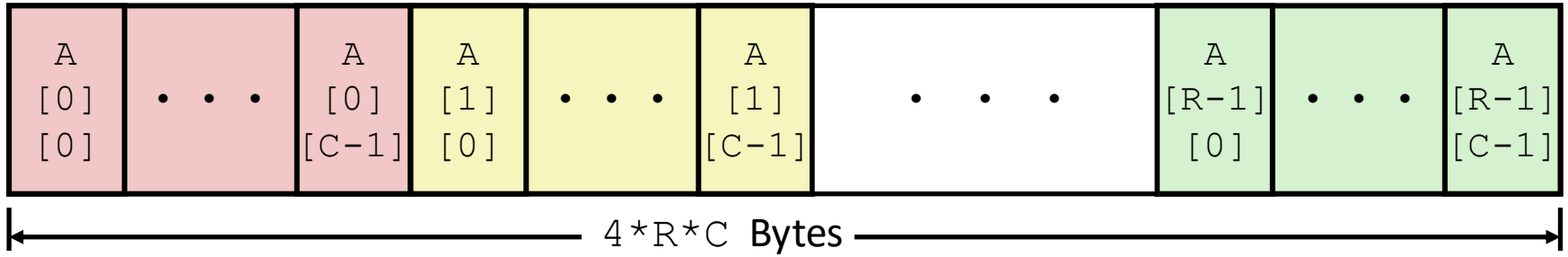


- Types
  - *What is A?* *The array object with a pointer to the first element &A[0] whose type is T (*)[C]*
  - *What is A[i][j]?* *T*

- Arrangement
  - Row-Major Ordering



Accessing items in the array



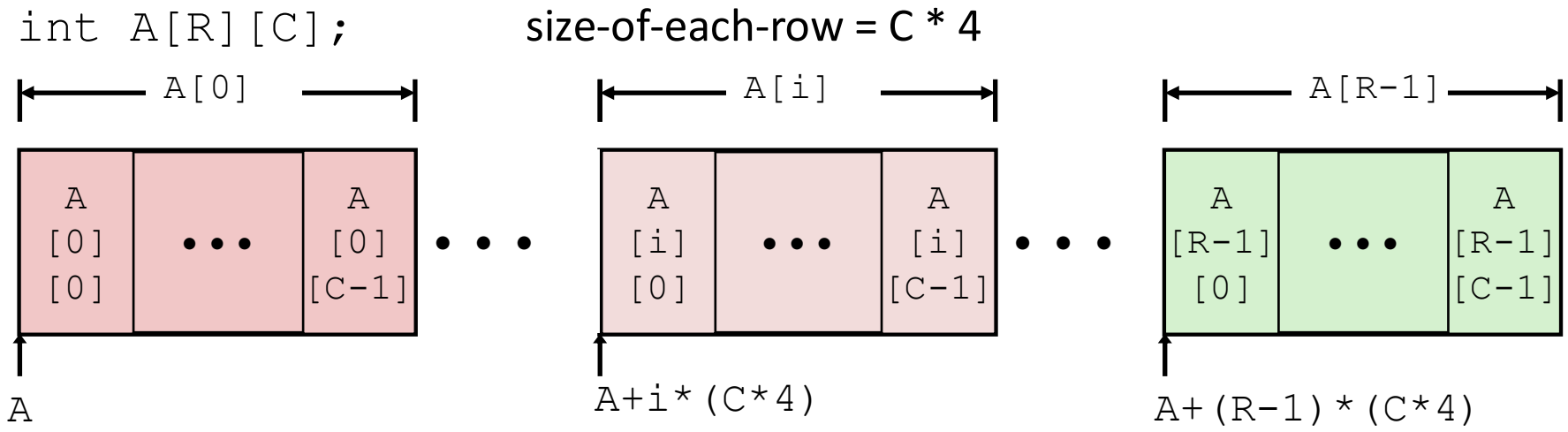

```
int A[R][C];
```

1. Figure out which row you want to access
   Skip over previous rows

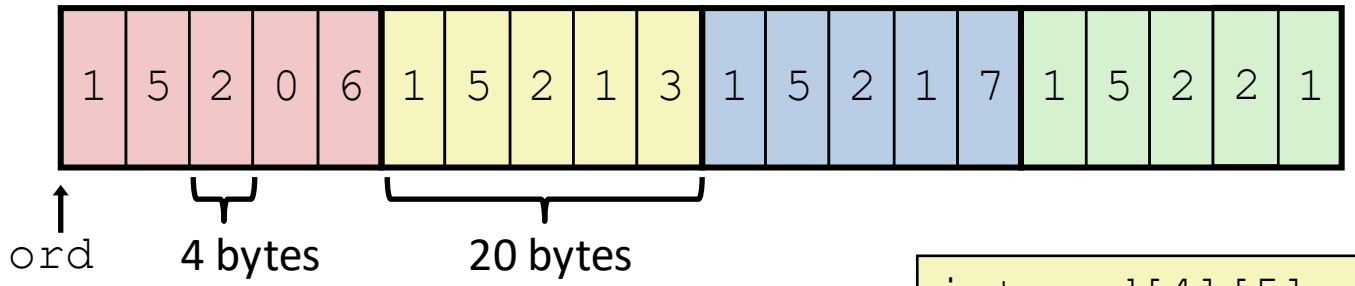
2. Figure out which column you want to access in that row
   Skip over previous columns in that row

Nested Array **Row** Access

- To figure out how to get the element we want
  - Let's first figure out how to get the **row** we want (its starting address)
- Calculating the address of a row
  - Row size: number-of-columns * element-size
  - Row offset: row-index * row-size
  - Row address: initial-address + row-offset



Nested Array **Row** Access Code

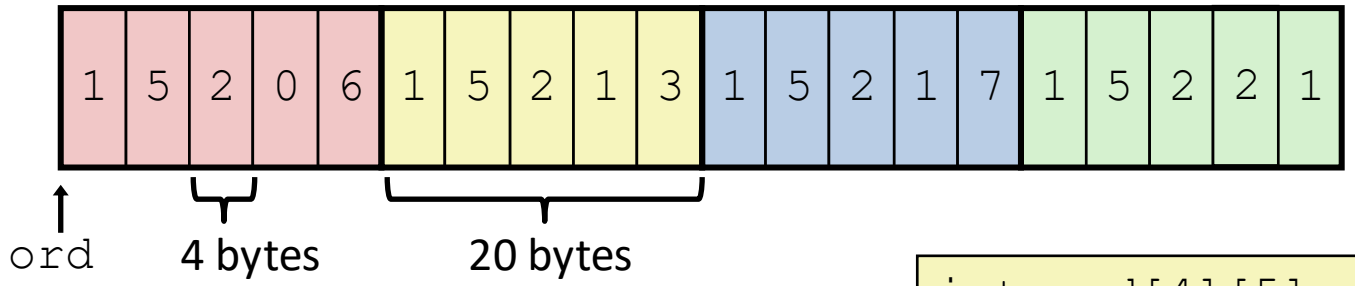



```
int* get_ord_row(size_t index)
{
  return ord[index];
}
```

```
int ord[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

```
  # %rdi = index
   leaq (%rdi,%rdi,4),%rax    # %rax = 5 * index
   leaq ord(,%rax,4),%rax     # %rax = ord + 4*(5*index)
```

Nested Array **Row** Access Code



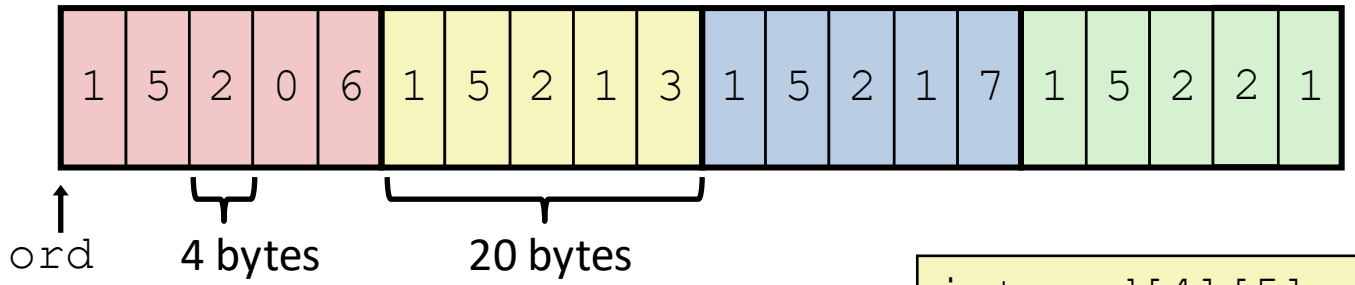

```
int* get_ord_row(size_t index)
{
  return ord[index];
}
```

```
int ord[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

```
  # %rdi = index
   leaq (%rdi,%rdi,4),%rax    # %rax = 5 * index
   leaq ord(,%rax,4),%rax     # %rax = ord + 4*(5*index)
```

- What's that displacement?
  - Constant address
    - How do we know this? We use `ord` without declaring it. `ord` is not a parameter (only `index` is), and it isn't defined inside the function body – it's declared at file scope
  - `ord` is a global. Always in a location known at compile-time. So constant address!

Nested Array **Row** Access Code




```
int* get_ord_row(size_t index)
{
  return ord[index];
}
```

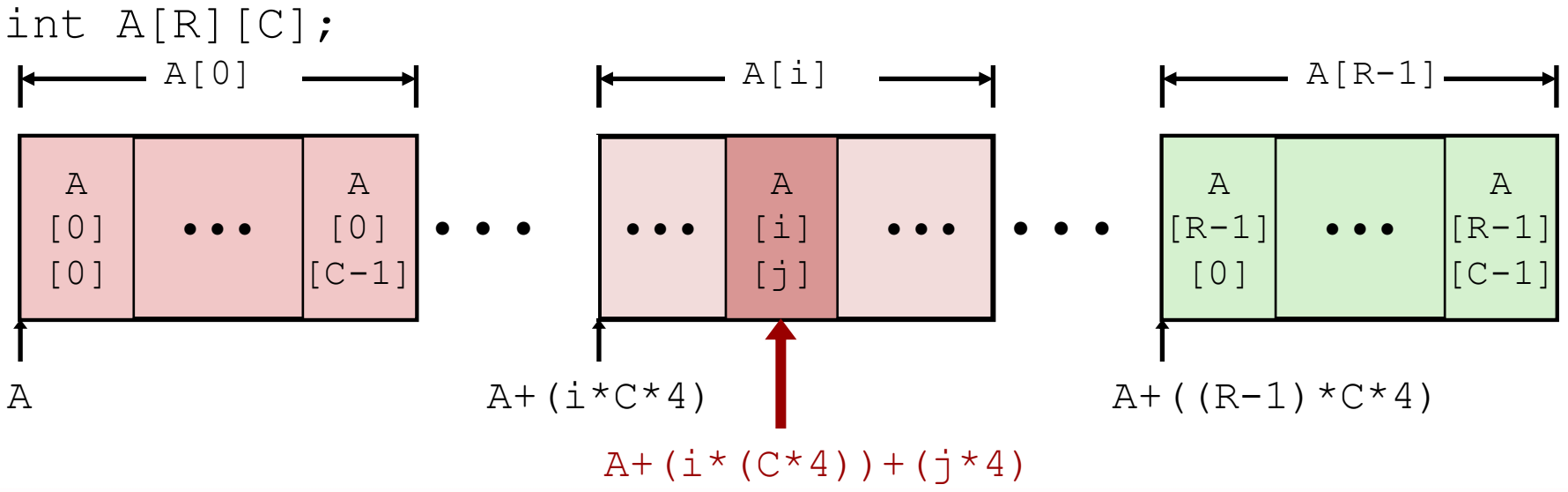
```
int ord[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

```
  # %rdi = index
   leaq (%rdi,%rdi,4),%rax    # %rax = 5 * index
   leaq ord(,%rax,4),%rax     # %rax = ord + 4*(5*index)
```

- Row Vector
  - **`ord[index]`** is array of 5 **`int`**'s
  - Row address: **`ord + 20*index`**
- Assembly Code
  - Computes and returns address
  - **`ord + 4*(5*index)`**

Nested Array **Element** Access

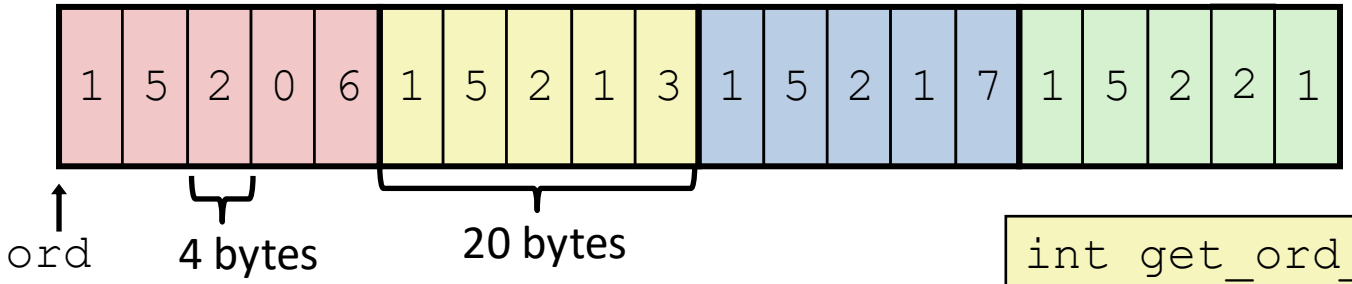
- Now, let's find the *element* that we want
- Calculating the address of an element
  - Row size: number-of-columns * element-size
  - Row offset: row-index * row-size
  - Element offset: column-index * element-size
  - Element address: initial-address + row-offset + element-offset



Element access calculation in words

- Don’t bother memorizing an equation. Think about how it works instead!
- To get a row, we skip over the prior rows in the array.
  - Each row is the size of the number of columns * the size of the Type.
- To get a column, we skip over the prior columns within that row.
  - Each column is the size of the Type.
- Add to base address of the array to get actual memory address.

Nested Array **Element** Access Code




```
int get_ord_digit(size_t index, size_t digit)
{
  return ord[index][digit];
}
```

```
# %rdi = index
leaq   (%rdi,%rdi,4), %rax    # 5*index
addq   %rax, %rsi             # 5*index + digit
movl   ord(,%rsi,4), %eax     # M[ord + 4*(5*index+digit)]
```

- Array Elements
  - **`ord[index][digit]`** is type **`int`**
  - Address: **`ord + 20*index + 4*digit`** = **`ord + 4*(5*index + digit)`**

Practice

- Find the addresses (assume array starts at address 0)
  - *Start + Offset to Row + Offset to element*
- int A[16][16]; A[1][3]
- char B[16][16]; B[10][7]
- char* B[10][10]; B[0][2]

Practice

- Find the addresses (assume array starts at address 0)
  - *Start + Offset to Row + Offset to element*
- int A[16][16]; A[1][3]
  - *0 + (1 * 16 * 4) + (3 * 4) = 64 + 12 = 76*
- char B[16][16]; B[10][7]
- char* B[10][10]; B[0][2]

Practice

- Find the addresses (assume array starts at address 0)
  - *Start + Offset to Row + Offset to element*
- int A[16][16]; A[1][3]
  - *0 + (1 * 16 * 4) + (3 * 4) = 64 + 12 = 76*
- char B[16][16]; B[10][7]
  - *0 + (10 * 16 * 1) + (7 * 1) = 160 + 7 = 167*
- char* B[10][10]; B[0][2]

Practice

- Find the addresses (assume array starts at address 0)
  - *Start + Offset to Row + Offset to element*
- int A[16][16]; A[1][3]
  - *0 + (1 * 16 * 4) + (3 * 4) = 64 + 12 = 76*
- char B[16][16]; B[10][7]
  - *0 + (10 * 16 * 1) + (7 * 1) = 160 + 7 = 167*
- char* B[10][10]; B[0][2]
  - *0 + (0 * 10 * 8) + (2 * 8) = 16*

Topics

- Multi-dimensional Arrays
- **Multi-level Arrays**

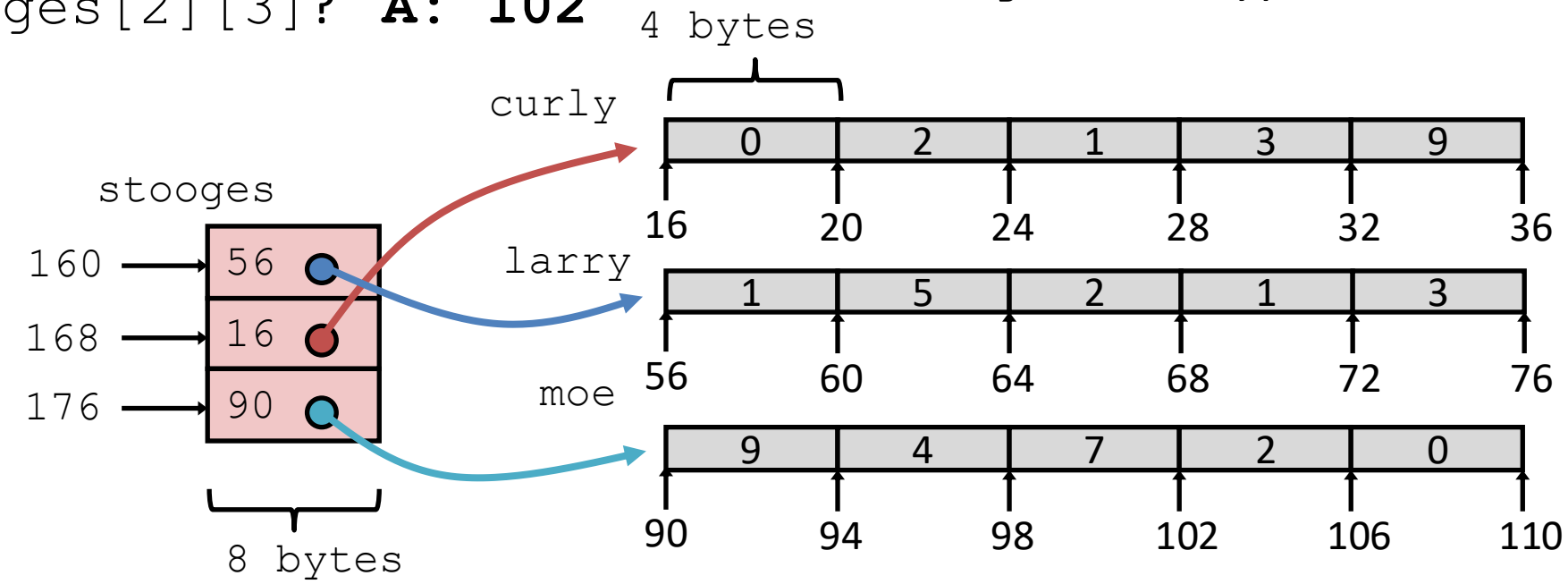
Multi-Level Array Example

```
int larry [5] = { 1, 5, 2, 1, 3 };
int curly [5] = { 0, 2, 1, 3, 9 };
int moe [5]   = { 9, 4, 7, 2, 0 };
```

```
int* stooges[3]={larry,curly,moe};
```

- Variable `stooges` denotes array of 3 elements
- Each element is a pointer (8 bytes)
- Each pointer points to array of `int`s
- `stooges` is of type `int* []`
- `stooges` is of type `int**`

Q: What is the address of `stooges[2][3]`? **A: 102**



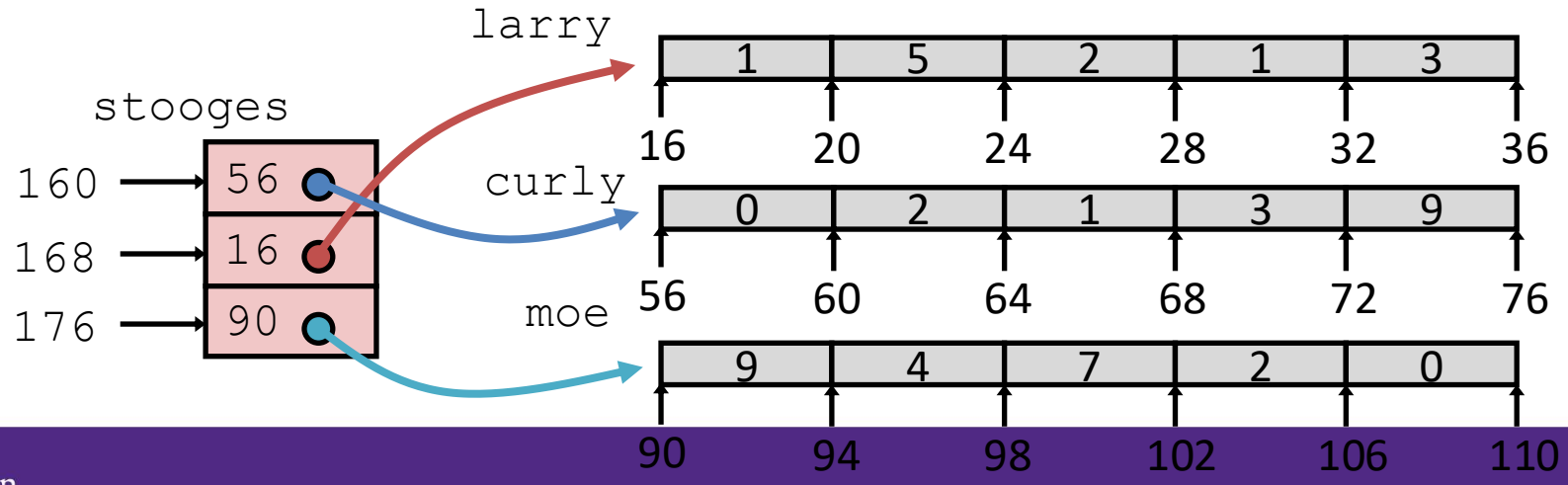
Multi-Level Array Element Access

```
int get_stooge_digit
  (size_t index, size_t digit){
  return stooges[index][digit];
}
```

- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

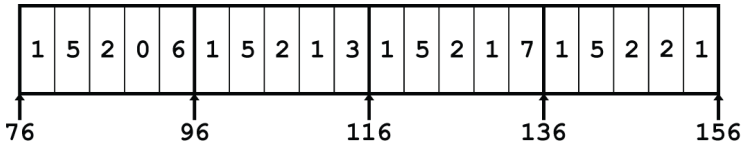
```
 salq    $2, %rsi                  # 4*digit
 addq    stooges(,%rdi,8), %rsi    # p = stooges[8*index] + 4*digit
 movl    (%rsi), %eax              # return *p
 ret
```

Element access `Mem[Mem[stooges+8*index] + 4*digit]`

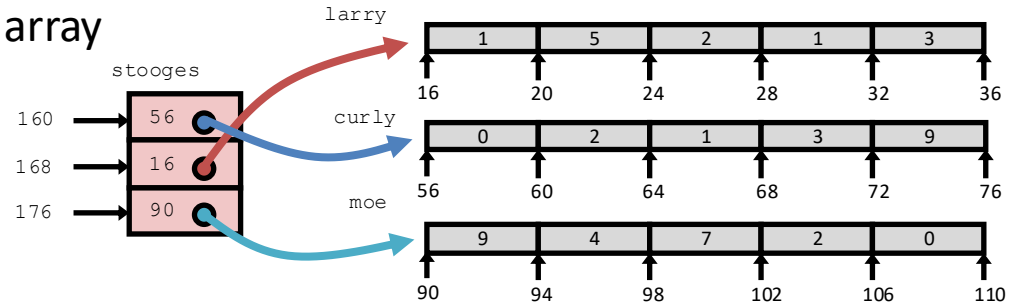


Nested vs. Multi-Level Array Element Accesses

Nested array



Multi-level array




```
int ord [4][5];


int get_ord_digit
  (size_t index, size_t digit){
  return ord[index][digit];
}
```

```
int larry[5], curly[5], moe[5];
int *stooges[3] = {larry, curly, moe};

int get_stooge_digit
  (size_t index, size_t digit){
  return stooges[index][digit];
}
```

Accesses look similar in C, but address computations are very different:

`ord` is sort of like `int*`

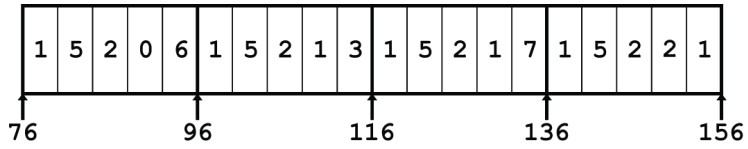
`stooges` is definitely `int**`

`Mem[ord+(20*index)+(4*digit)]`

`Mem[Mem[stooges+(8*index)]+(4*digit)]`

Nested vs. Multi-Level Arrays

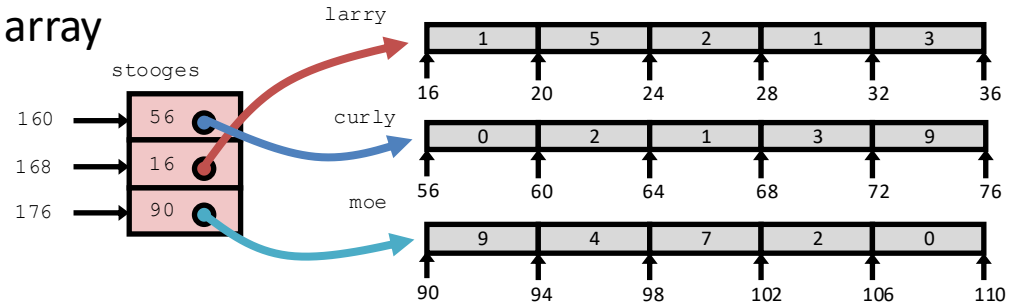
Nested array



`Mem[ord+(20*index)+(4*digit)]`

- Strengths
  - Fast element access
    - Single memory access
  - Efficient memory usage
    - Stored in contiguous memory
- Limitations
  - Requires fixed size rows
  - Large memory usage
    - All rows need to be allocated

Multi-level array



`Mem[Mem[stooges+(8*index)]+(4*digit)]`

- Strengths
  - Rows may be of different size
  - Rows could even be different types
    - First array would store `void*`
- Limitations
  - Slow element access
    - Two memory references
  - Memory fragmentation
    - Many small chunks allocated