

Arrays

CSE 351 Autumn 2017

Instructor:

Justin Hsia

Teaching Assistants:

Lucas Wotton

Michael Zhang

Parker DeWilde

Ryan Wong

Sam Gehman

Sam Wolfson

Savanna Yee

Vinny Palaniappan

Administrivia

- ❖ Lab 2 due Friday (10/27)
- ❖ Homework 3 due next Friday (11/3)
- ❖ Lab 3 released next Wednesday (11/1)

- ❖ **Midterm** (Monday 10/30, 5:10-6:20pm)
 - 70 minute exam (longer than usual)
 - ID check, so come at 5pm
 - Midterm details Piazza post: [@163](#)
 - **Review session:** 5:30-7:30pm on Friday (10/27) in EEB 105
 - Take a look at midterm review packet before section

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs**
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

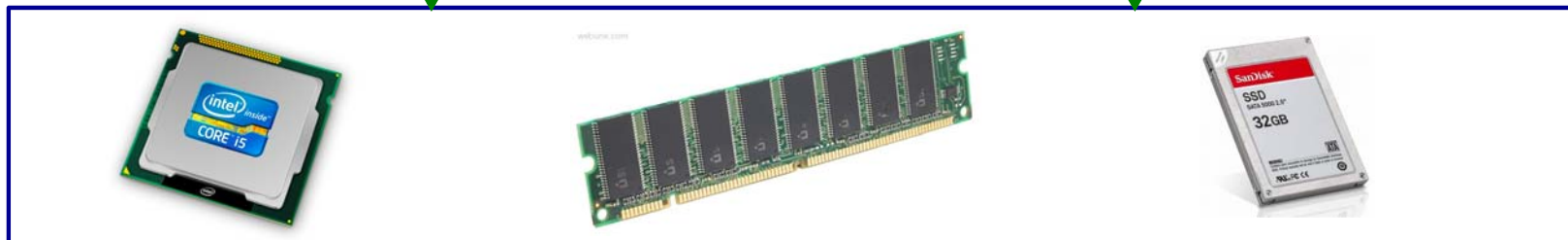
Assembly language:

```
get_mpg:
    pushq    %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

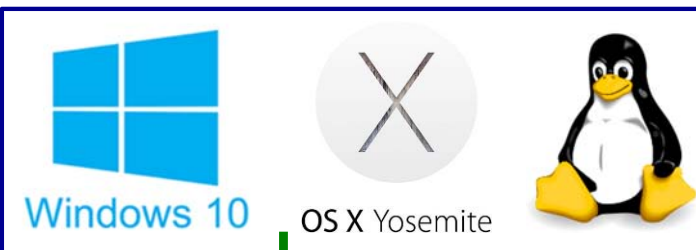
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:



OS:



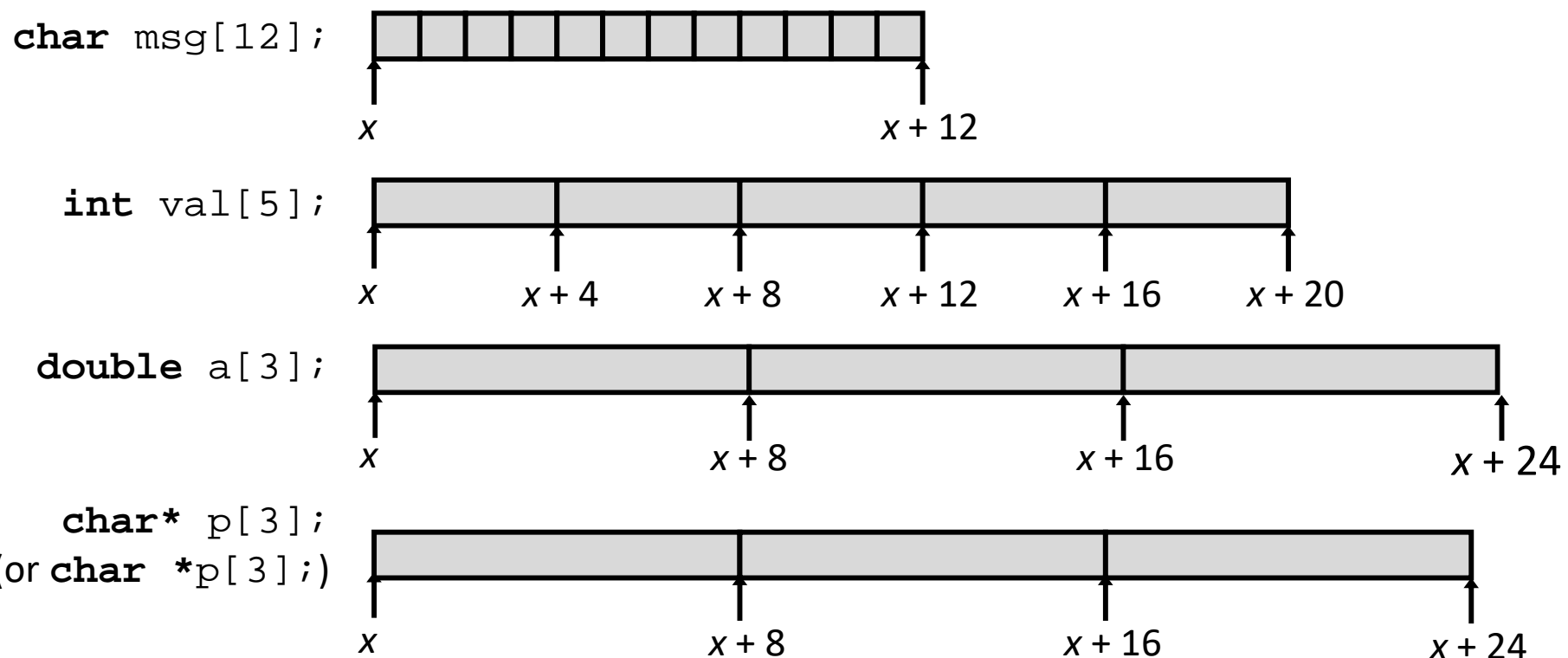
Data Structures in Assembly

- ❖ **Arrays**
 - **One-dimensional**
 - Multi-dimensional (nested)
 - Multi-level
- ❖ **Structs**
 - Alignment
- ❖ **Unions**

Array Allocation

❖ Basic Principle

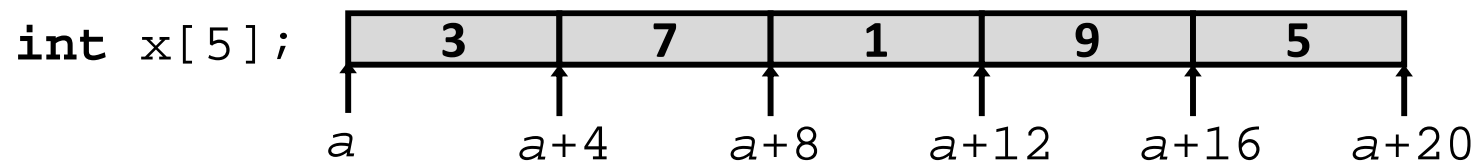
- $\mathbf{T} \ A[N]; \rightarrow$ array of data type \mathbf{T} and length N
- *Contiguously* allocated region of $N * \text{sizeof}(\mathbf{T})$ bytes
- Identifier A returns address of array (type \mathbf{T}^*)



Array Access

❖ Basic Principle

- **T** A[N]; → array of data type **T** and length N
- Identifier A returns address of array (type **T***)



❖ Reference

<u>Reference</u>	<u>Type</u>	<u>Value</u>
<code>x[4]</code>	<code>int</code>	5
<code>x</code>	<code>int*</code>	a
<code>x+1</code>	<code>int*</code>	$a + 4$
<code>&x[2]</code>	<code>int*</code>	$a + 8$
<code>x[5]</code>	<code>int</code>	?? (whatever's in memory at addr $x+20$)
<code>*(x+1)</code>	<code>int</code>	7
<code>x+i</code>	<code>int*</code>	$a + 4*i$

Array Example

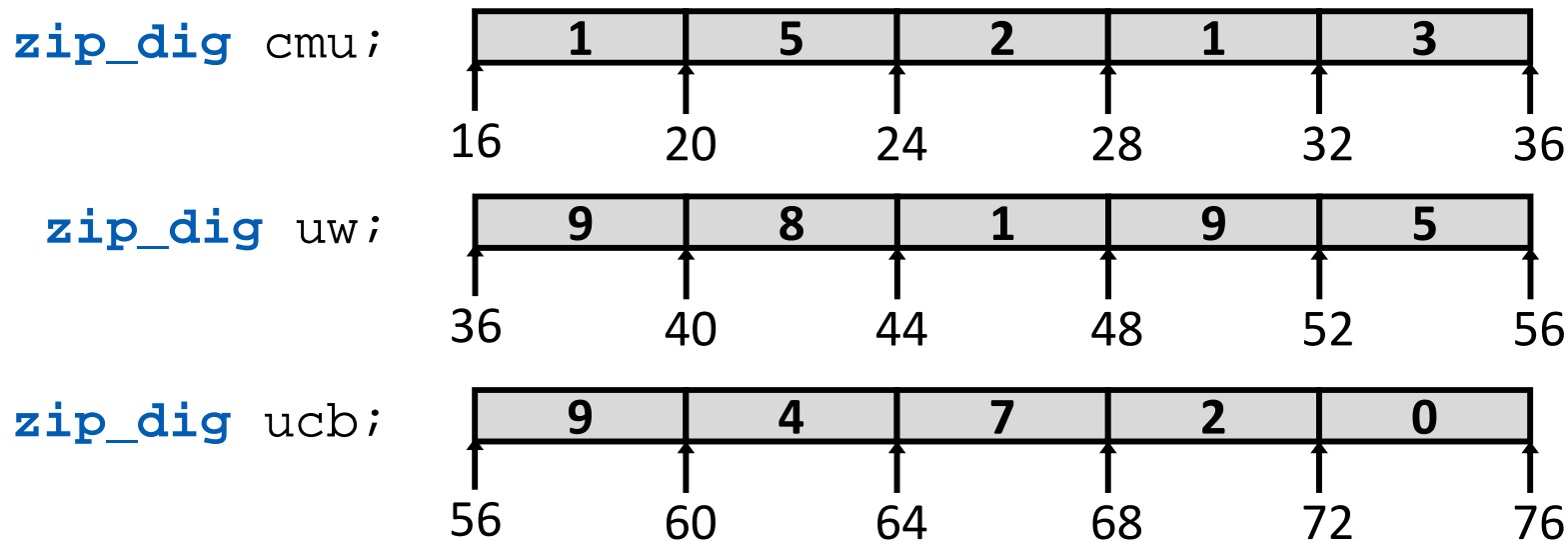
```
typedef int zip_dig[5];  
  
zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig uw  = { 9, 8, 1, 9, 5 };  
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

initialization

- ❖ typedef: Declaration “**zip_dig** uw” equivalent to “**int** uw[5]”

Array Example

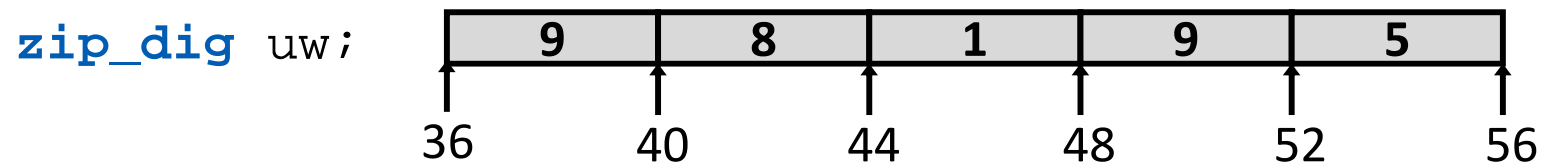
```
typedef int zip_dig[5];  
  
zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig uw  = { 9, 8, 1, 9, 5 };  
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- ❖ Example arrays happened to be allocated in successive 20 byte blocks
 - Not guaranteed to happen in general

Array Accessing Example

```
typedef int zip_dig[5];
```



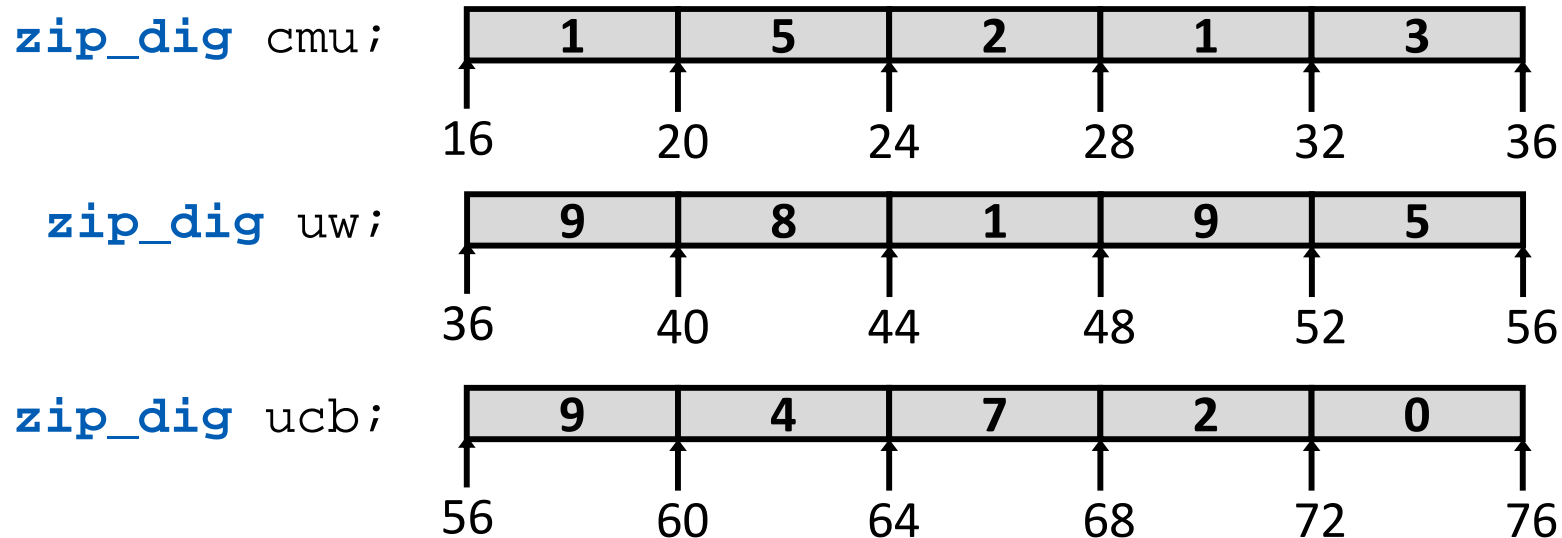
```
int get_digit(zip_dig z, int digit)
{
    return z[digit];
}
```

```
get_digit:
    movl (%rdi,%rsi,4), %eax # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi+4*%rsi`, so use memory reference `(%rdi,%rsi,4)`

```
typedef int zip_dig[5];
```

Referencing Examples



Reference **Address** **Value** **Guaranteed?**

- `uw[3]`
- `uw[6]`
- `uw[-1]`
- `cmu[15]`

- ❖ No bounds checking
- ❖ Example arrays happened to be allocated in successive 20 byte blocks
 - Not guaranteed to happen in general

Array Loop Example

```
typedef int zip_dig[5];
```

$$zi = 10 * 0 + 9 = 9$$

$$zi = 10 * 9 + 8 = 98$$

$$zi = 10 * 98 + 1 = 981$$

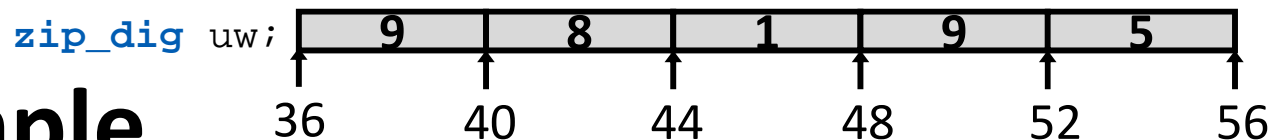
$$zi = 10 * 981 + 9 = 9819$$

$$zi = 10 * 9819 + 5 = 98195$$

```
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

9	8	1	9	5
---	---	---	---	---

Array Loop Example



❖ Original:

```
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

❖ Transformed:

- Eliminate loop variable *i*, use pointer *zend* instead
- Convert array code to pointer code
 - Pointer arithmetic on *z*
- Express in do-while form (no test at entrance)

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 5;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z < zend);
    return zi;
}
```

address just past 5th digit

← Increments by 4 (size of int)

Array Loop Implementation

gcc with -O1

❖ Registers:

```
%rdi z
%rax zi
%rcx zend
```

❖ Computations

-
-

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 5;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z < zend);
    return zi;
}
```

```
# %rdi = z
leaq 20(%rdi),%rcx #
movl $0,%eax #
.L17:
leal (%rax,%rax,4),%edx #
movl (%rdi),%eax #
leal (%rax,%rdx,2),%eax #
addq $4,%rdi #
cmpq %rdi,%rcx #
jne .L17 #
```

C Details: Arrays and Pointers

- ❖ Arrays are (almost) identical to pointers
 - `char *string` and `char string[]` are nearly identical declarations
 - Differ in subtle ways: initialization, `sizeof()`, etc.
- ❖ An array variable looks like a pointer to the first (0th) element
 - `ar[0]` same as `*ar`; `ar[2]` same as `*(ar+2)`
- ❖ An array variable is read-only (no assignment)
 - Cannot use `"ar = <anything>"`

C Details: Arrays and Functions

- ❖ Declared arrays only allocated while the scope is valid:

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

BAD!

- ❖ An array is passed to a function as a pointer:
 - Array size gets lost!

```
int foo(int ar[], unsigned int size) {  
    ... ar[size-1] ...  
}
```

*Really int *ar*

*Must explicitly
pass the size!*

Data Structures in Assembly

- ❖ **Arrays**
 - One-dimensional
 - **Multi-dimensional (nested)**
 - Multi-level
- ❖ **Structs**
 - Alignment
- ❖ **Unions**

Nested Array Example

```
typedef int zip_dig[5];
```

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

Remember, $\mathbf{T} \ A[N]$ is
an array with elements
of type \mathbf{T} , with length N

same as:

```
int sea[4][5];
```

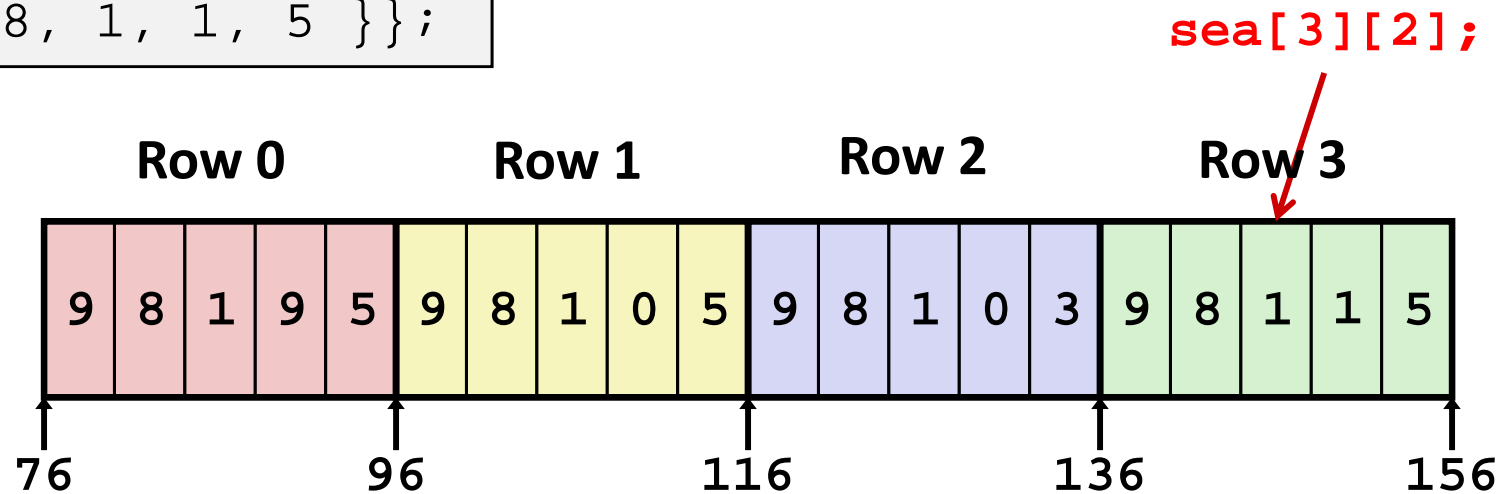
What is the layout in memory?

Nested Array Example

```
typedef int zip_dig[5];
```

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

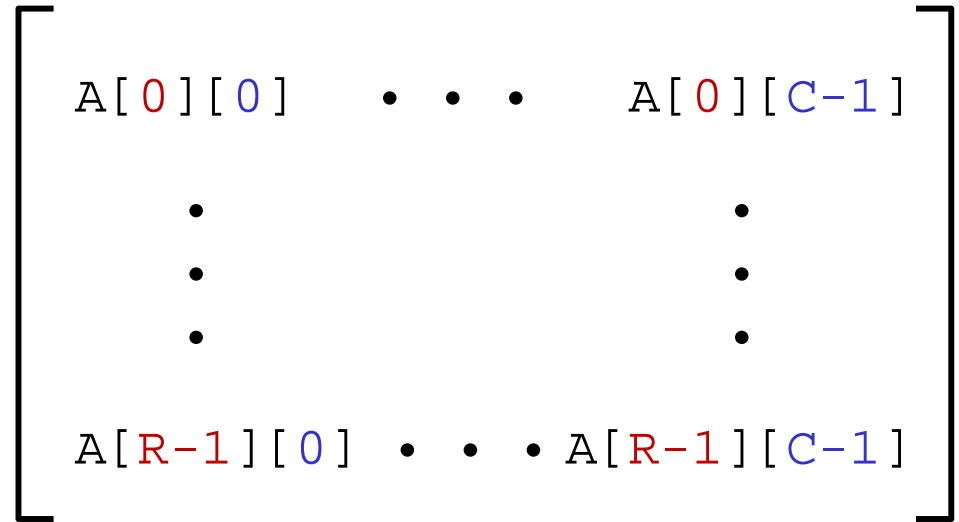
Remember, $\mathbf{T} \ A[N]$ is an array with elements of type \mathbf{T} , with length N



- ❖ “Row-major” ordering of all elements
- ❖ Elements in the same row are contiguous
- ❖ Guaranteed (in C)

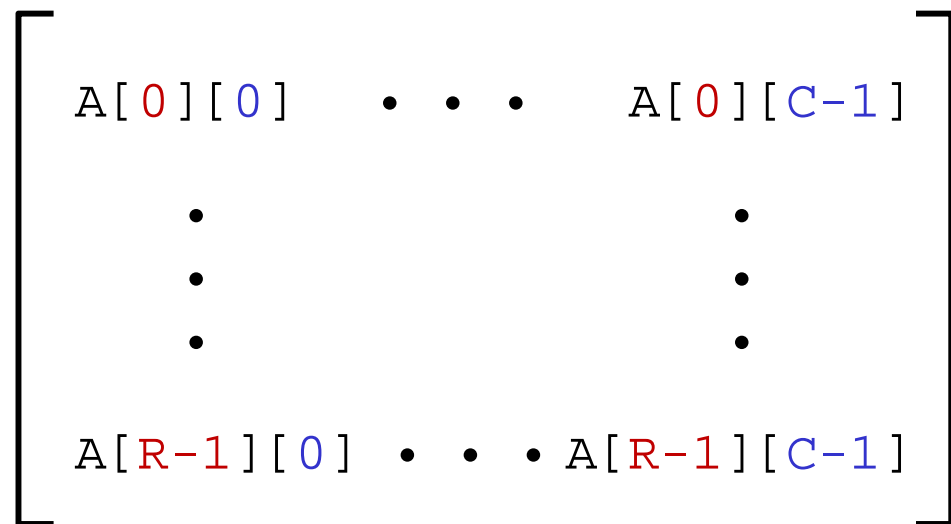
Two-Dimensional (Nested) Arrays

- ❖ Declaration: `T A[R][C];`
 - 2D array of data type `T`
 - `R` rows, `C` columns
 - Each element requires `sizeof(T)` bytes
- ❖ Array size?



Two-Dimensional (Nested) Arrays

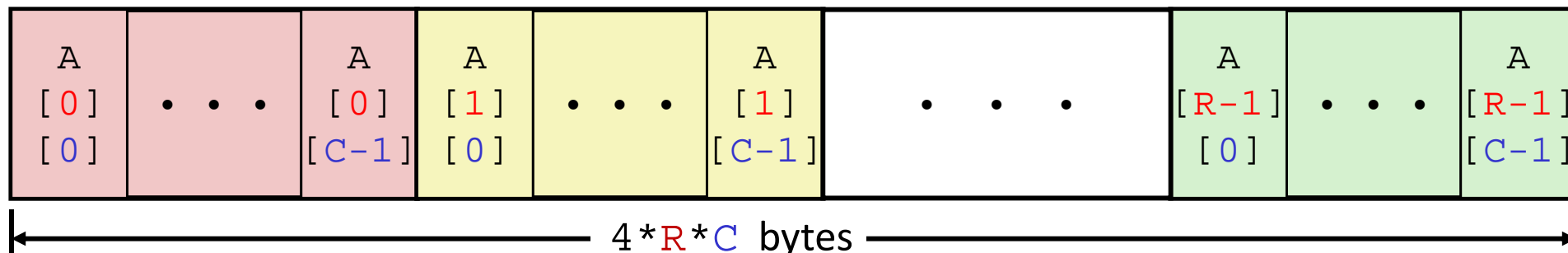
- ❖ Declaration: `T A[R][C];`
 - 2D array of data type `T`
 - `R` rows, `C` columns
 - Each element requires `sizeof(T)` bytes



- ❖ Array size:
 - `R * C * sizeof(T)` bytes

- ❖ Arrangement: **row-major** ordering

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



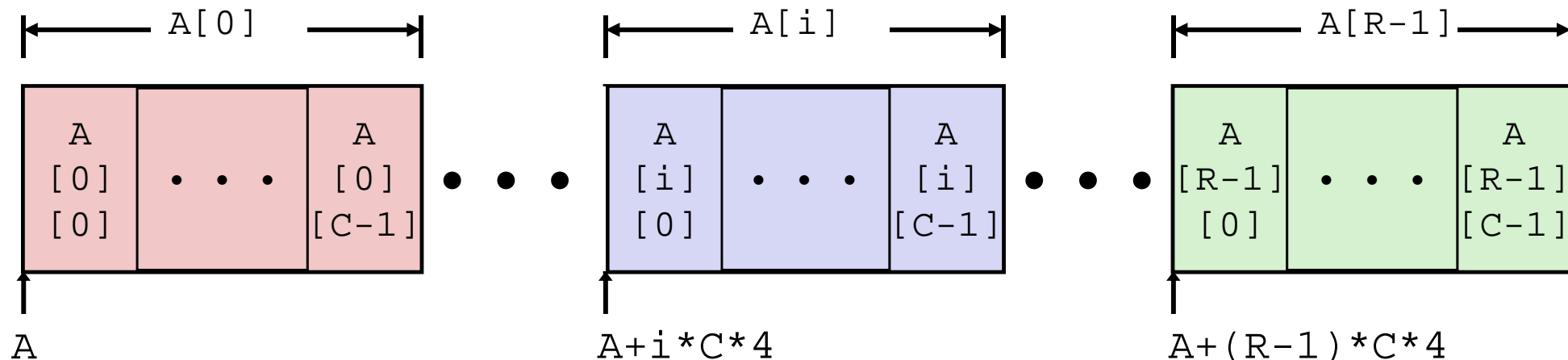
Nested Array Row Access

❖ Row vectors

■ Given $\mathbf{T} \ A[R][C]$,

- $A[i]$ is an array of C elements (“row i ”)
- Each element of type \mathbf{T} requires K bytes
- A is address of array
- Starting address of row $i = A + i * (C * K)$

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



Nested Array Row Access Code

```
int* get_sea_zip(int index)
{
    return sea[index];
}
```

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

- What data type is `sea[index]`?
- What is its starting address?

```
get_sea_zip(int):
    movslq    %edi, %rdi
    leaq     (%rdi,%rdi,4), %rdx
    leaq     0(,%rdx,4), %rax
    addq     $sea, %rax
    ret

sea:
    .long    9
    .long    8
    .long    1
    .long    9
    .long    5
    .long    9
    .long    8
    ...
```

Nested Array Row Access Code

```
int* get_sea_zip(int index)
{
    return sea[index];
}
```

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

- What data type is `sea[index]`?
- What is its starting address?

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax
leaq sea(,%rax,4),%rax
```

Translation?

Nested Array Row Access Code

```
int* get_sea_zip(int index)
{
    return sea[index];
}
```

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

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax # 5 * index
leaq sea(,%rax,4),%rax # sea + (20 * index)
```

❖ Row Vector

- `sea[index]` is array of 5 ints
- Starting address = `sea+20*index`

❖ Assembly Code

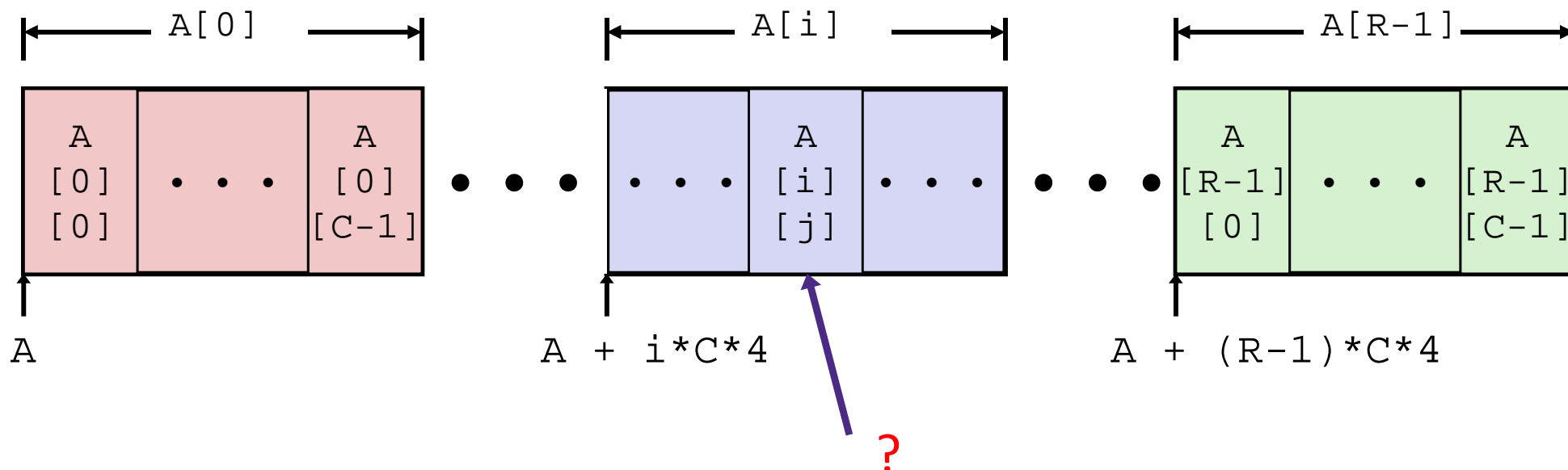
- Computes and returns address
- Compute as: `sea+4*(index+4*index) = sea+20*index`

Nested Array Element Access

❖ Array Elements

- $A[i][j]$ is element of type T , which requires K bytes
- Address of $A[i][j]$ is

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



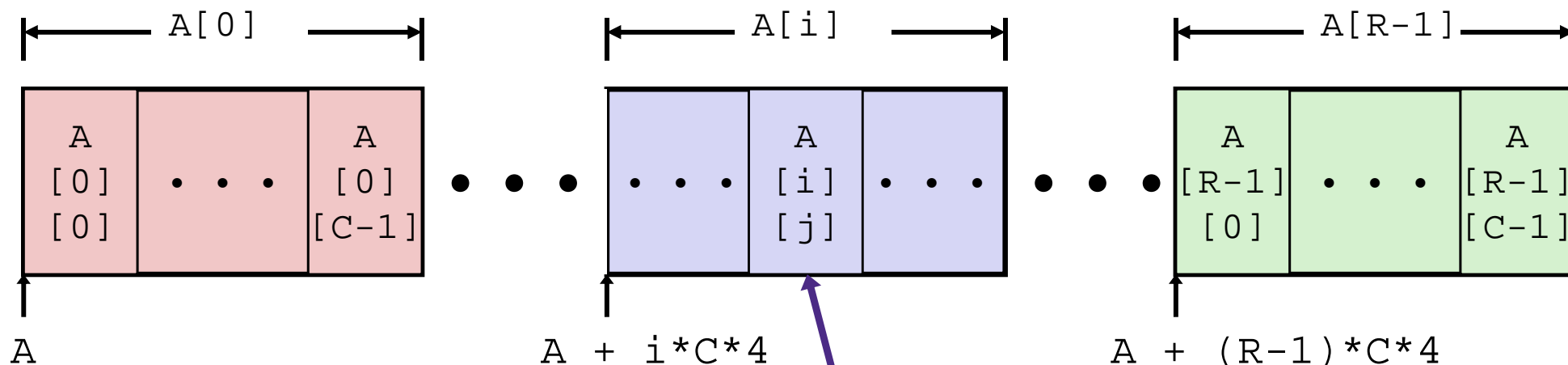
Nested Array Element Access

❖ Array Elements

- $A[i][j]$ is element of type T , which requires K bytes
- Address of $A[i][j]$ is

$$A + i * (C * K) + j * K == A + (i * C + j) * K$$

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



$$A + i * C * 4 + j * 4$$

Nested Array Element Access Code

```
int get_sea_digit
(int index, int digit)
{
    return sea[index][digit];
}
```

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

```
leaq    (%rdi,%rdi,4), %rax    # 5*index
addl    %rax, %rsi            # 5*index+digit
movl    sea(,%rsi,4), %eax    # *(sea + 4*(5*index+digit))
```

❖ Array Elements

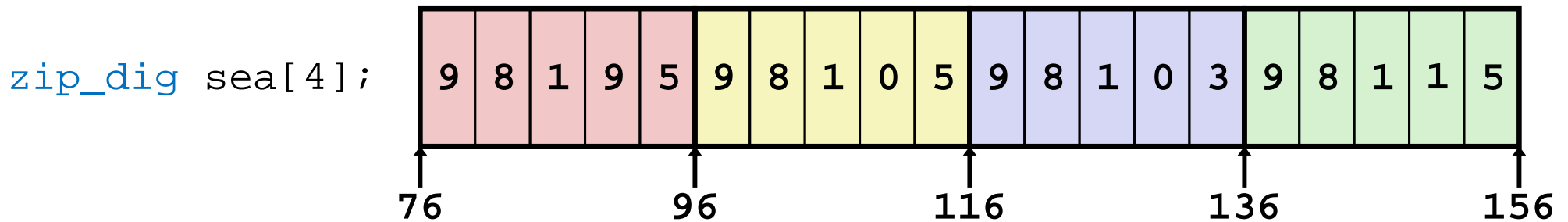
- `sea[index][digit]` is an `int` (`sizeof(int)=4`)
- $\text{Address} = \text{sea} + 5 \cdot 4 \cdot \text{index} + 4 \cdot \text{digit}$

❖ Assembly Code

- Computes address as: $\text{sea} + ((\text{index} + 4 \cdot \text{index}) + \text{digit}) \cdot 4$
- `movl` performs memory reference

```
typedef int zip_dig[5];
```

Strange Referencing Examples



<u>Reference</u>	<u>Address</u>	<u>Value</u>	<u>Guaranteed?</u>
------------------	----------------	--------------	--------------------

<code>sea[3][3]</code>			
------------------------	--	--	--

<code>sea[2][5]</code>			
------------------------	--	--	--

<code>sea[2][-1]</code>			
-------------------------	--	--	--

<code>sea[4][-1]</code>			
-------------------------	--	--	--

<code>sea[0][19]</code>			
-------------------------	--	--	--

<code>sea[0][-1]</code>			
-------------------------	--	--	--

- Code does not do any bounds checking
- Ordering of elements within array guaranteed

Data Structures in Assembly

- ❖ **Arrays**
 - One-dimensional
 - Multi-dimensional (nested)
 - **Multi-level**
- ❖ **Structs**
 - Alignment
- ❖ **Unions**

Multi-Level Array Example

Multi-Level Array Declaration(s):

```
int cmu[5] = { 1, 5, 2, 1, 3 };  
int uw[5] = { 9, 8, 1, 9, 5 };  
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {uw, cmu, ucb};
```

2D Array Declaration:

```
zip_dig univ2D[3] = {  
    { 9, 8, 1, 9, 5 },  
    { 1, 5, 2, 1, 3 },  
    { 9, 4, 7, 2, 0 }  
};
```

Is a multi-level array the same thing as a 2D array?

NO

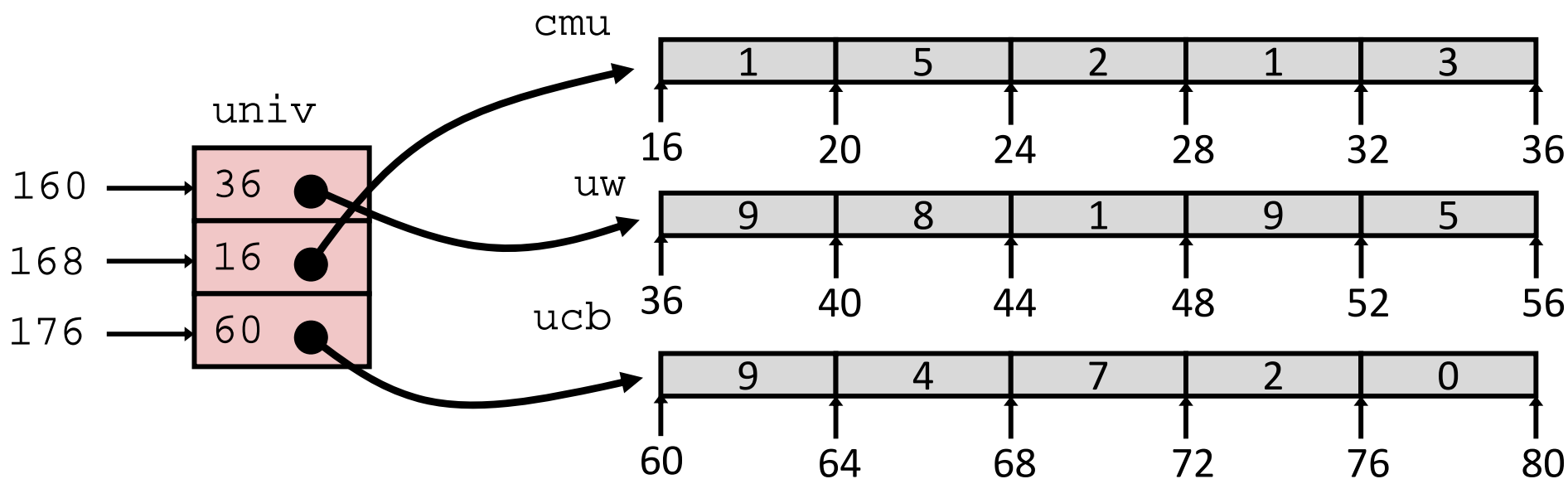
One array declaration = one contiguous block of memory

Multi-Level Array Example

```
int cmu[5] = { 1, 5, 2, 1, 3 };
int uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {uw, cmu, ucb};
```

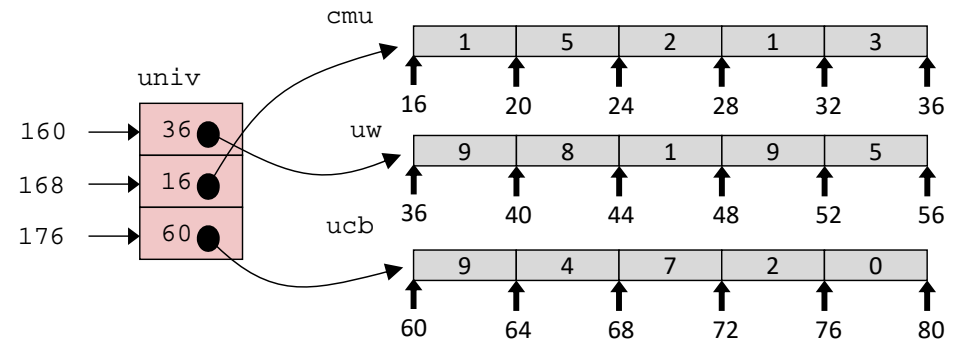
- ❖ Variable `univ` denotes array of 3 elements
- ❖ Each element is a pointer
 - 8 bytes each
- ❖ Each pointer points to array of `ints`



Note: this is how Java represents multi-dimensional arrays

Element Access in Multi-Level Array

```
int get_univ_digit
(int index, int digit)
{
    return univ[index][digit];
}
```



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

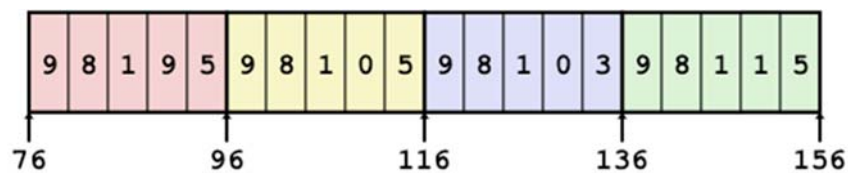
❖ Computation

- Element access $\text{Mem}[\text{Mem}[\text{univ} + 8 * \text{index}] + 4 * \text{digit}]$
- Must do **two memory reads**
 - First get pointer to row array
 - Then access element within array
- But allows inner arrays to be different lengths (not in this example)

Array Element Accesses

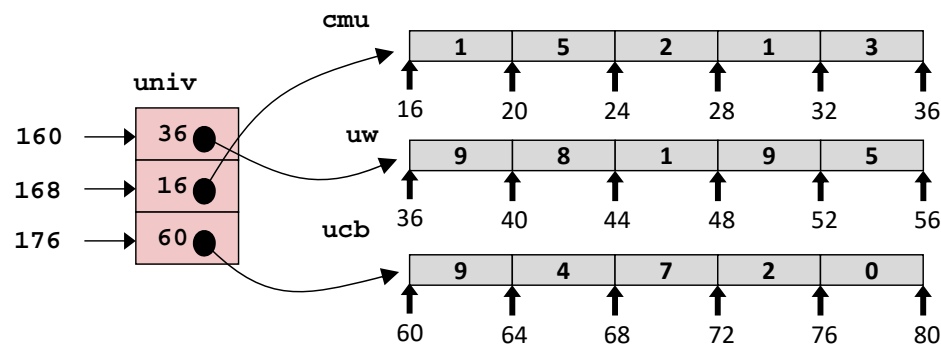
Nested array

```
int get_sea_digit
(int index, int digit)
{
    return sea[index][digit];
}
```



Multi-level array

```
int get_univ_digit
(int index, int digit)
{
    return univ[index][digit];
}
```

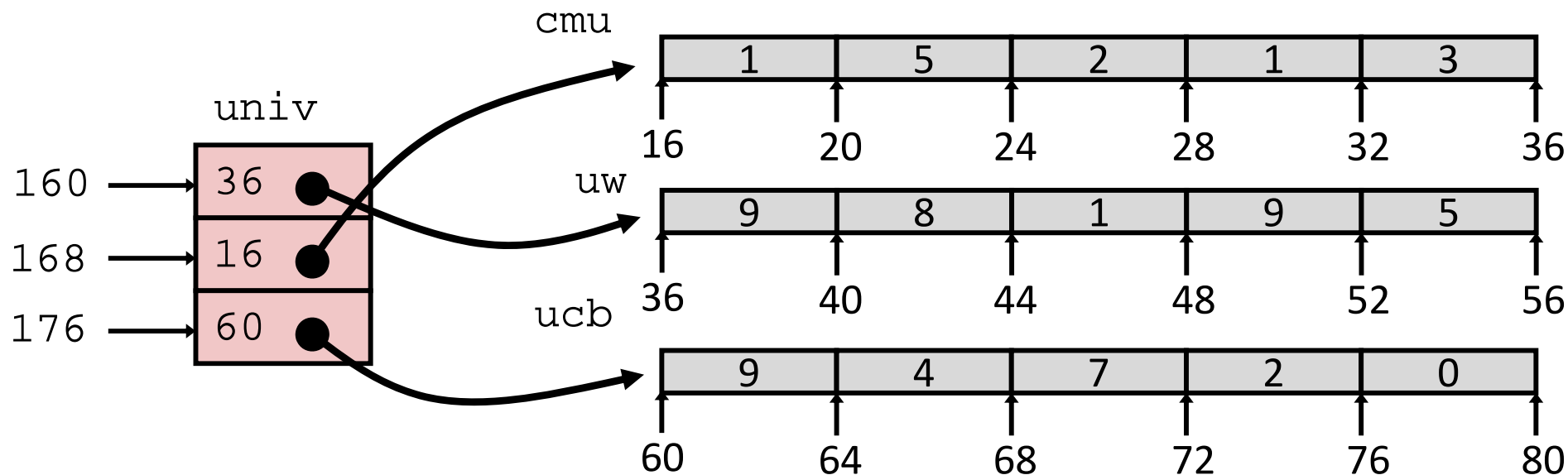


Access *looks* the same, but it isn't:

Mem[sea+20*index+4*digit]

Mem[**Mem**[univ+8*index]+4*digit]

Strange Referencing Examples



<u>Reference</u>	<u>Address</u>	<u>Value</u>	<u>Guaranteed?</u>
<code>univ[2][3]</code>			
<code>univ[1][5]</code>			
<code>univ[2][-2]</code>			
<code>univ[3][-1]</code>			
<code>univ[1][12]</code>			

- C code does not do any bounds checking
- Location of each lower-level array in memory is *not* guaranteed

Summary

- ❖ Contiguous allocations of memory
- ❖ **No bounds checking** (and no default initialization)
- ❖ Can usually be treated like a pointer to first element
- ❖ **int** a[4][5]; → array of arrays
 - all levels in one contiguous block of memory
- ❖ **int*** b[4]; → array of pointers to arrays
 - First level in one contiguous block of memory
 - Each element in the first level points to another “sub” array
 - Parts anywhere in memory