Arrays

CSE 351 Winter 2021

Instructor:

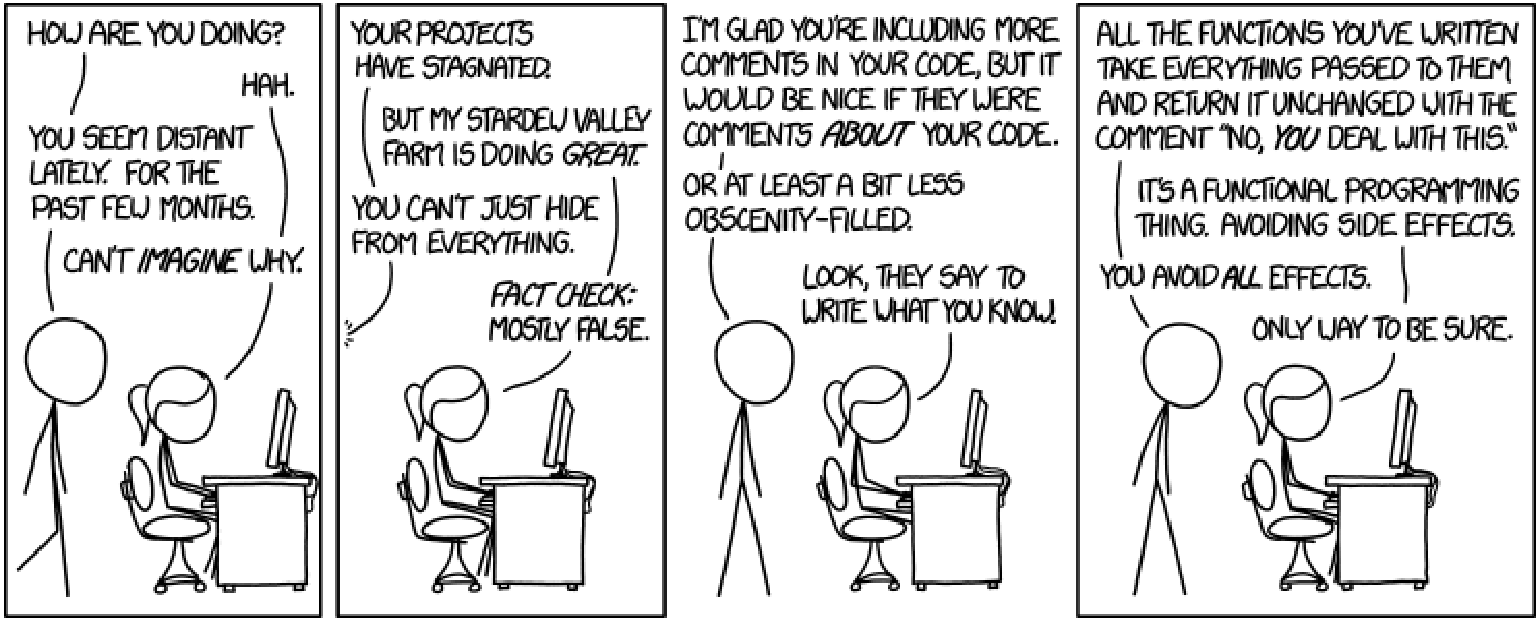
Mark Wyse

Teaching Assistants:

Kyrie Dowling Catherine Guevara Ian Hsiao

Jim Limprasert Armin Magness Allie Pfleger

Cosmo Wang Ronald Widjaja



http://xkcd.com/1790/

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();
```

Assembly language:

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

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 10

OS X Yosemite

Computer system:

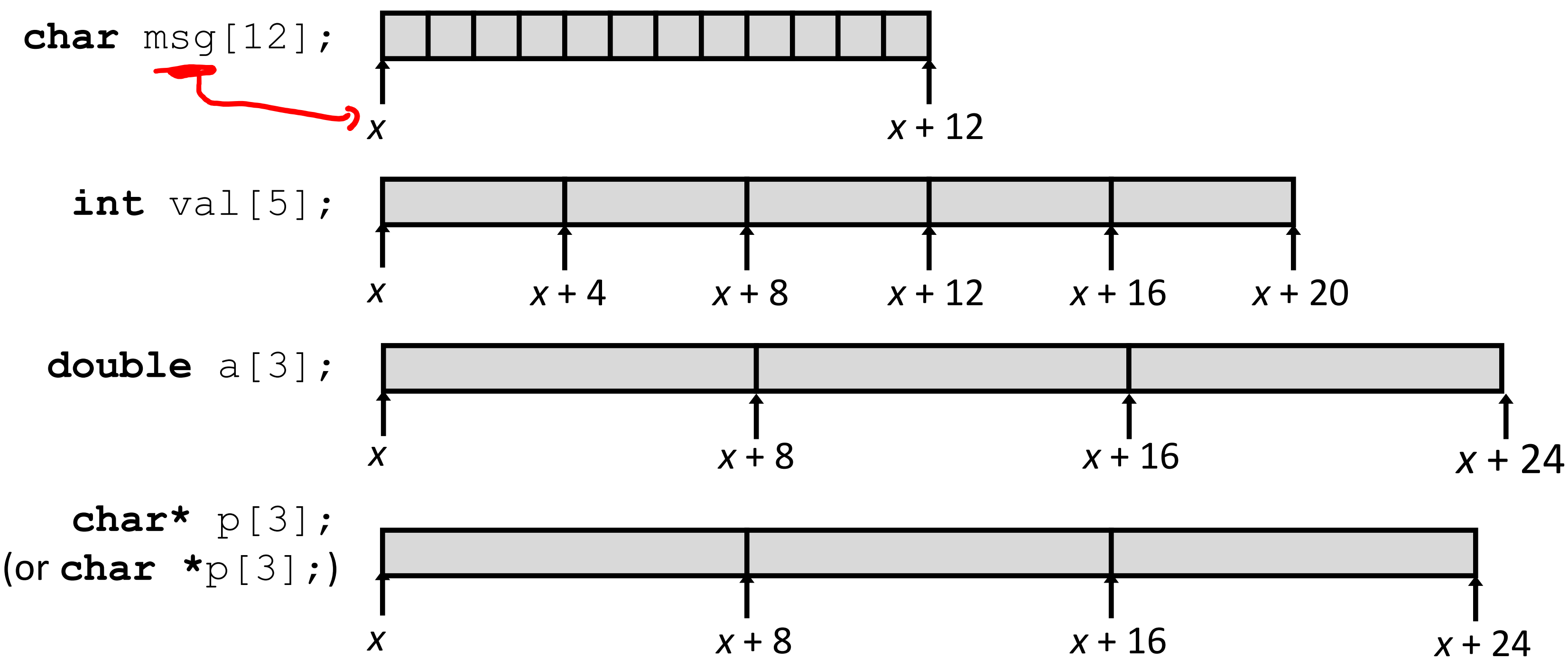
- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- **Arrays & structs**
- Memory & caches
- Processes
- Virtual memory
- Memory allocation
- Java vs. C

Data Structures in Assembly

- **Arrays**
  - **One-dimensional**
  - Multidimensional (nested)
  - Multilevel
- Structs – on Friday w/ Cosmo!
  - Alignment
- ~~Unions~~

Review: Array Allocation

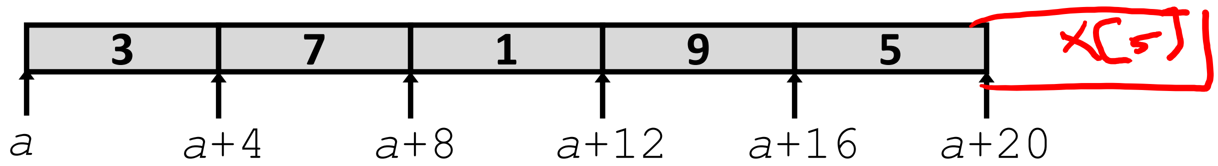
- Basic Principle
  - `T A[N];` → array of data type **T** and length `N`
  - *Contiguously* allocated region of `N*sizeof(T)` bytes
  - Identifier `A` returns address of array (type **T***)



Review: Array Access

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

`int x[5];`



- Reference Type Value

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

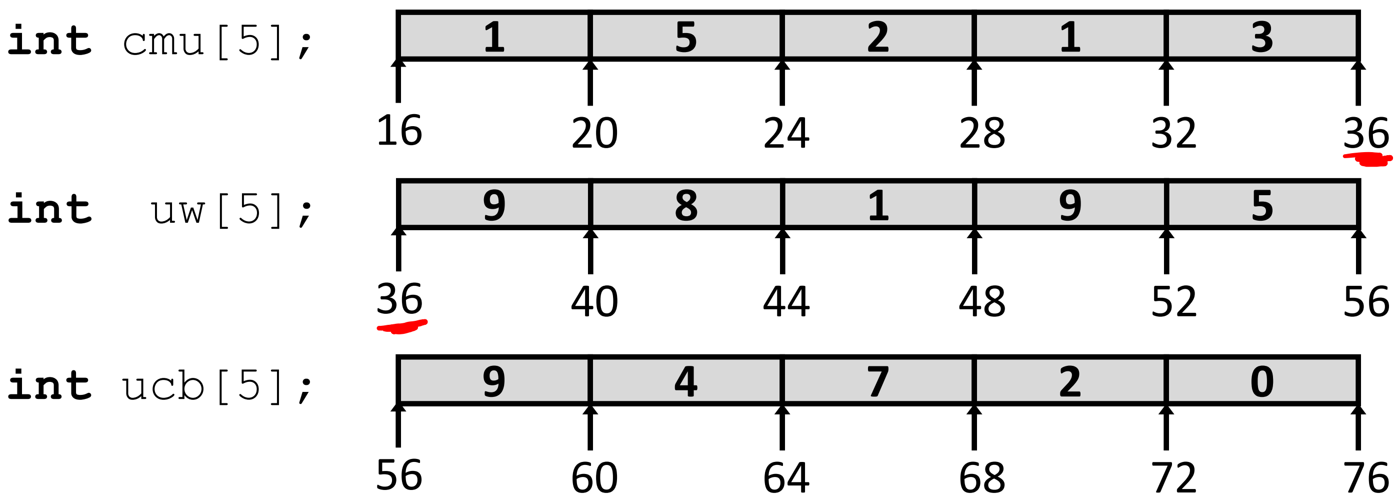
Array Example

```
// arrays of ZIP code digits
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

brace-enclosed list initialization

Array Example

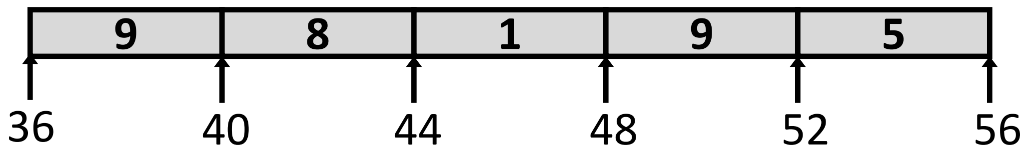
```
// arrays of ZIP code digits
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```



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

Array Accessing Example

```
int   uw[5];
```



D(Rb, Ri, S)

Rb = array start

Ri = array index

S = sizeof(T)

```
// return specified digit of ZIP code
int get_digit(int z[5], int digit) {
  return z[digit];
}
```

sizeof(int)

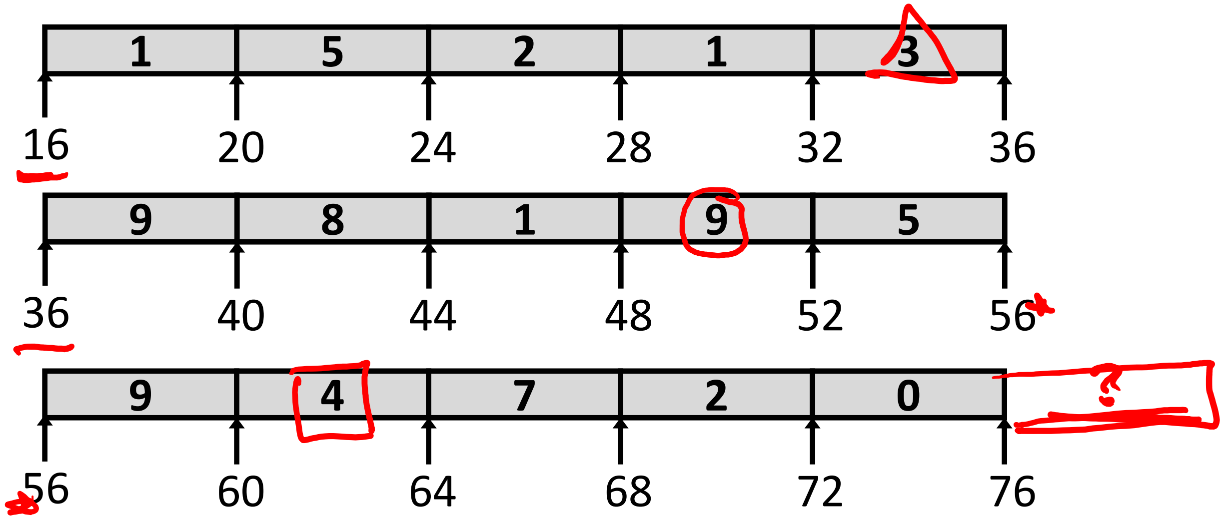
```
get_digit:
  movl  (%rdi,%rsi,4), %eax   # z[digit]
```

ret

- 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)`

Referencing Examples

```
int cmu[5];
int  uw[5];
int ucb[5];
```



$R_b + R_i * S$

Reference	Address	Value	Guaranteed?
uw[3]	36 + 3*4 = 48	9	Yes
uw[6]	36 + 6*4 = 60	4	No
uw[-1]	36 + (-1)*4 = 32	3	No
cmu[15]	16 + 15*4 = 76	?	No

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

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 name is an expression (not a variable) that returns the address of the array
  - It *looks* like a pointer to the first (0^{th}) element
    - *ar same as ar[0], *(ar+2) same as ar[2]
  - An array name is read-only (no assignment) because it is a *label*
    - 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!

Really `int *ar`

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

Must explicitly pass the size!

Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - **Multidimensional (nested)**
  - Multilevel
- Structs
  - Alignment
- ~~Unions~~

Nested Array Example

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

Remember, **T** `A[N]` is an array with elements of type **T**, with length `N`

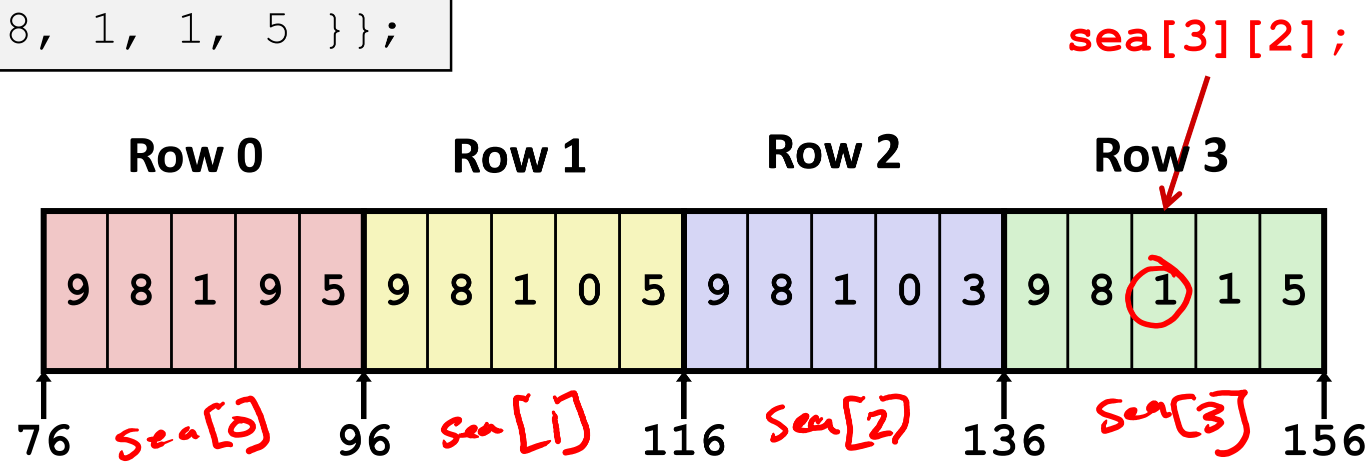
- What is the layout in memory?

Nested Array Example

R C

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

Remember, **T** A[N] is an array with elements of type **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?

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

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

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

- Array size:
  - `R*C*sizeof(T)` bytes
- Arrangement: **row-major** ordering

`int A[R][C];`

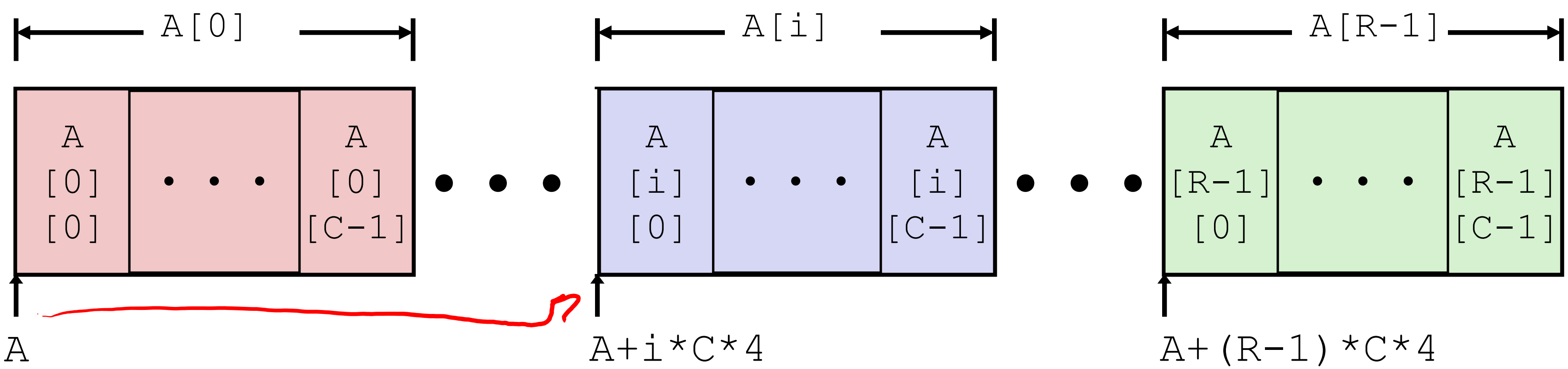
A [0] [0]	• • •	A [0] [C-1]	A [1] [0]	• • •	A [1] [C-1]	• • •	A [R-1] [0]	• • •	A [R-1] [C-1]

|← 4*R*C bytes →|

Nested Array Row Access

- Row vectors
  - Given **T** `A[R][C]`,
    - `A[i]` is an array of `C` elements ("row `i`") A[row]
    - `A` is address of array
    - Starting address of row `i` = $\underset{\text{array addr}}{\texttt{A}}$ + $\underbrace{\texttt{i*(C * sizeof(T))}}_{\text{row * size of row}}$

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



Nested Array Row Access Code

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

pointer to some row

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

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

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

index → 8 byte

D

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]`? int*
- What is its value? A + C * sizeof(T) * index = A + 5 * 4 * index

Rows = 4
Cols = 5

```
# %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 }};
```

A + 5 * 4 * index

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

D

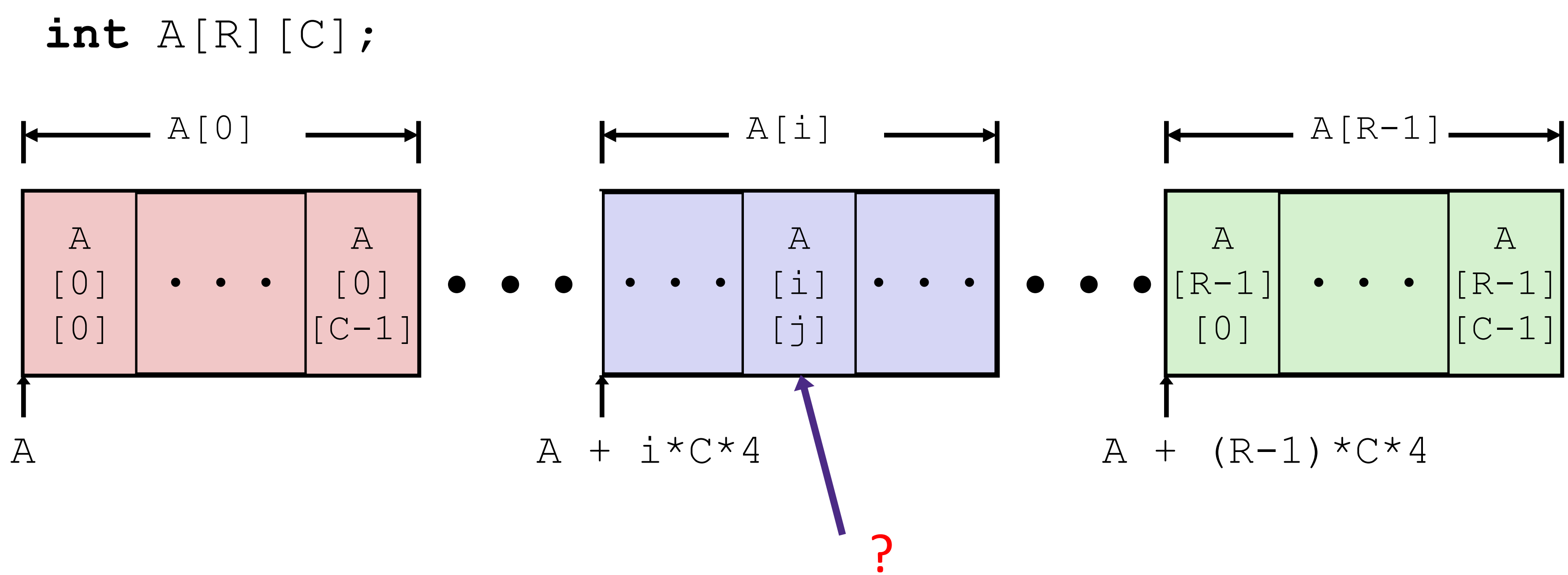
A + (4 * (5 * index))

- Row Vector
  - `sea[index]` is array of 5 `int`s
  - Starting address = `sea+20*index`
- Assembly Code
  - Computes and returns address
  - Compute as: `sea+4*(index+4*index)= sea+20*index`

sizeof a row

Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type **`T`**, which requires K bytes
  - Address of `A[i][j]` is

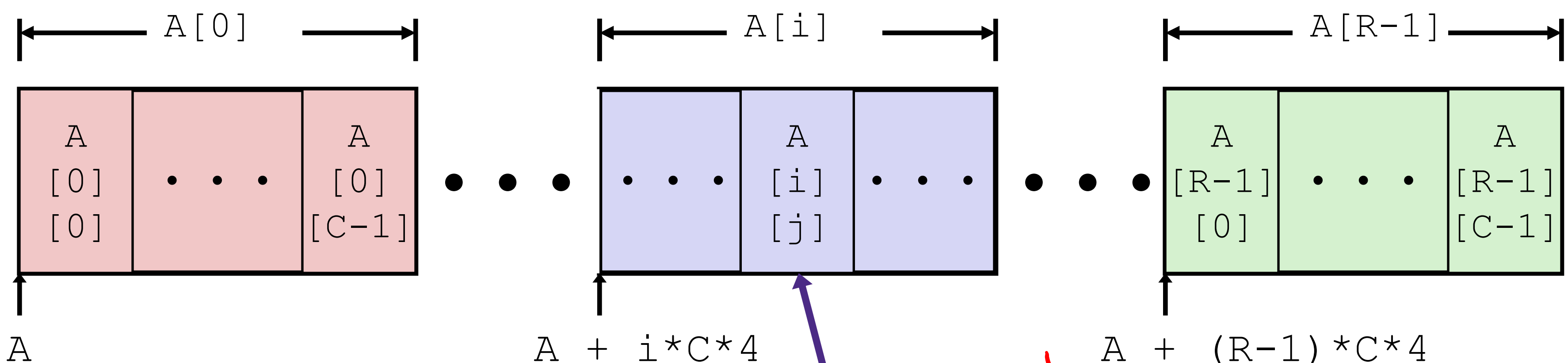


Nested Array Element Access

- 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];
```



$\underbrace{\overset{\text{row}}{\texttt{A + i*C*4}}} + \overbrace{\texttt{j*4}}^{\text{element in row}}$

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 }};
```

index, digit, size of (int)

```
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)
  - Address = sea + 5*4*index + 4*digit
- Assembly Code
  - 5*index + colindy
  - Computes address as: sea + ((index+4*index) + digit)*4
  - movl performs memory reference

Multidimensional Referencing Examples

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

9	8	1	9	5	9	8	1	0	5	9	8	1	0	3	9	8	1	1	5

72 76 96 116 136 140 144 148 152 156

Reference	Address	Value	Guaranteed?
`sea[3][3]`	$76 + 20*3 + 4*3 = 148$	1	Yes
`sea[2][5]`	$76 + 20*2 + 4*5 = 136$	9	Yes
`sea[2][-1]`	$76 + 20*2 + (-1)*4 = 112$	5	Yes
`sea[4][-1]`	$76 + 20*4 + (-1)*4 = 152$	5	Yes
`sea[0][19]`	$76 + 19*4 = 152$	5	Yes
`sea[0][-1]`	$76 + (-1)*4 = 72$	??	No

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

Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multidimensional (nested)
  - **Multilevel**
- Structs
  - Alignment
- ~~Unions~~

Multilevel Array Example

Multilevel 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};
```

array of int ptr

Is a multilevel array the same thing as a 2D array?

NO

2D Array Declaration:

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

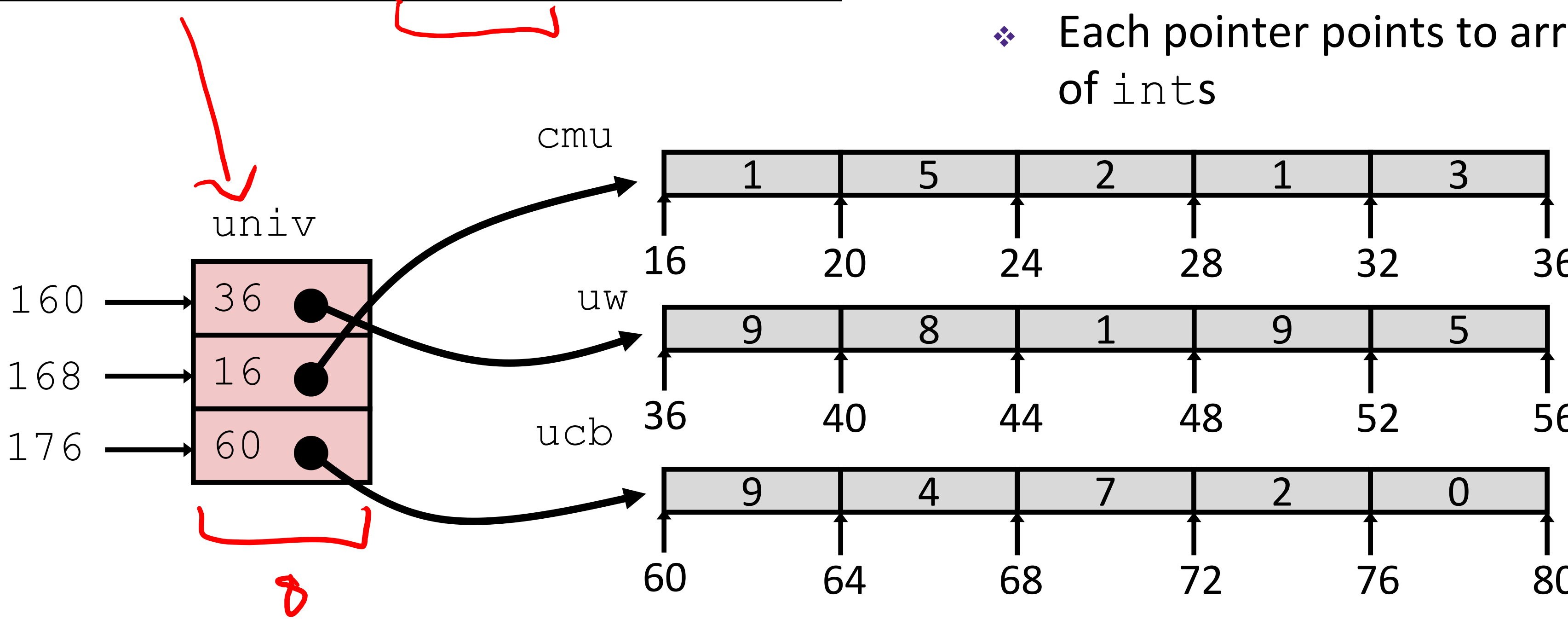
One array declaration = one contiguous block of memory

Multilevel 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 `int`s

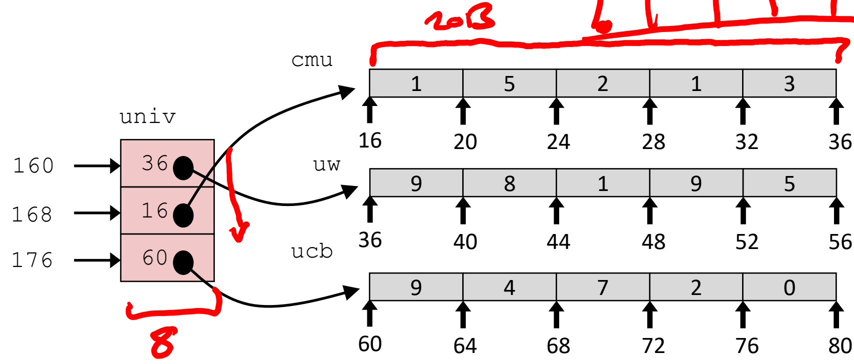


Note: this is how Java represents multidimensional arrays

Element Access in Multilevel Array

A[i] 2D 20B

```
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
```

address of "row"

- Computation
  - Element access Mem[Mem[`univ+8*index`]+`4*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

Multidimensional array

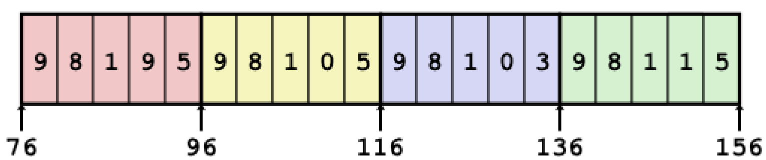
```
int get_sea_digit
  (int index, int digit)
{
  return sea[index][digit];
}
```

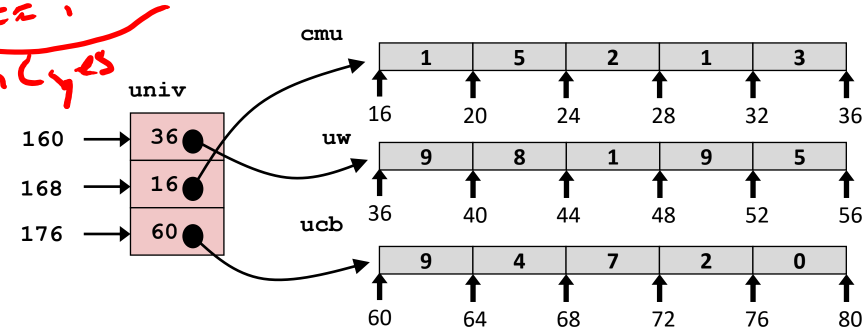
Multilevel array

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

== ?

in C, yes





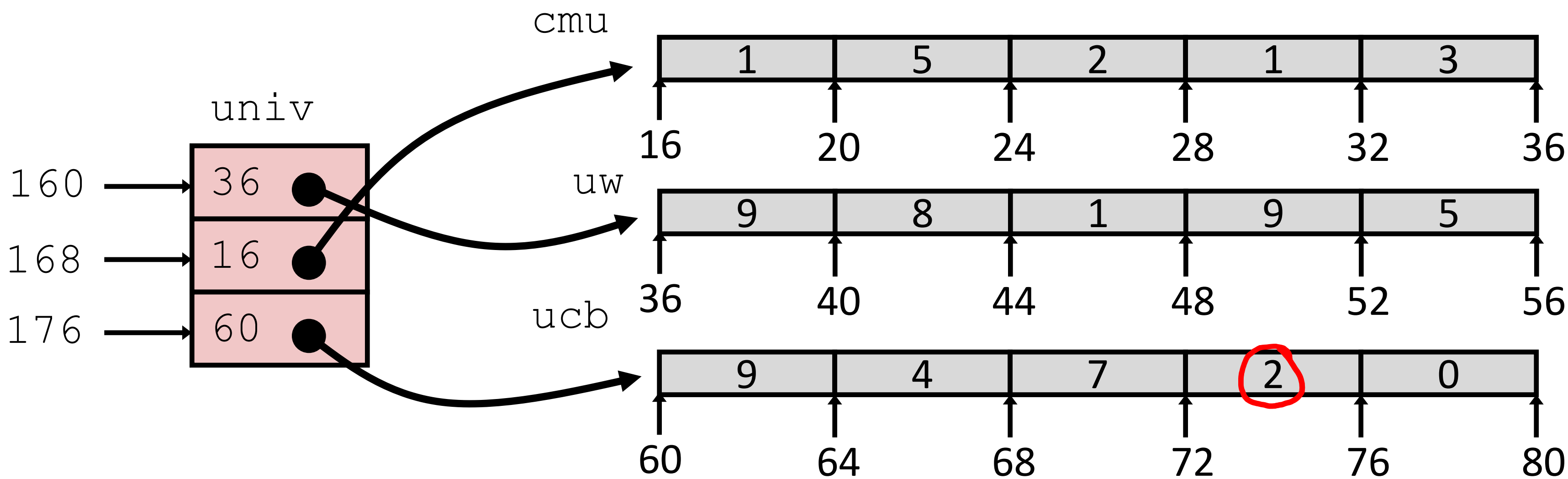
Access *looks* the same, but it isn't:

two dereferences

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

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

Multilevel Referencing Examples



Reference	Address	Value	Guaranteed?
univ[2][3]	Mem[176] + 3*4 = 60+12 = 72	2	Yes
univ[1][5]	M[168] + 5*4 = 16 + 20 = 36	9	No
univ[2][-2]	M[176] + (-2)*4 = 60 - 8 = 52	5	No
univ[3][-1]	M[184] + (-1)*4 = ?? - 4 = ??	???	No
univ[1][12]	M[168] + 12*4 = 16+48 = 64	4	No

- 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