UNM SCHOOL *of* ENGINEERING
Department of Computer Science

Lecture #12 – part 2 or L13

Complex Data and Control Structures: Single/Multi-Dimensional Arrays + structs and alignment Sections 3.8 and 3.9

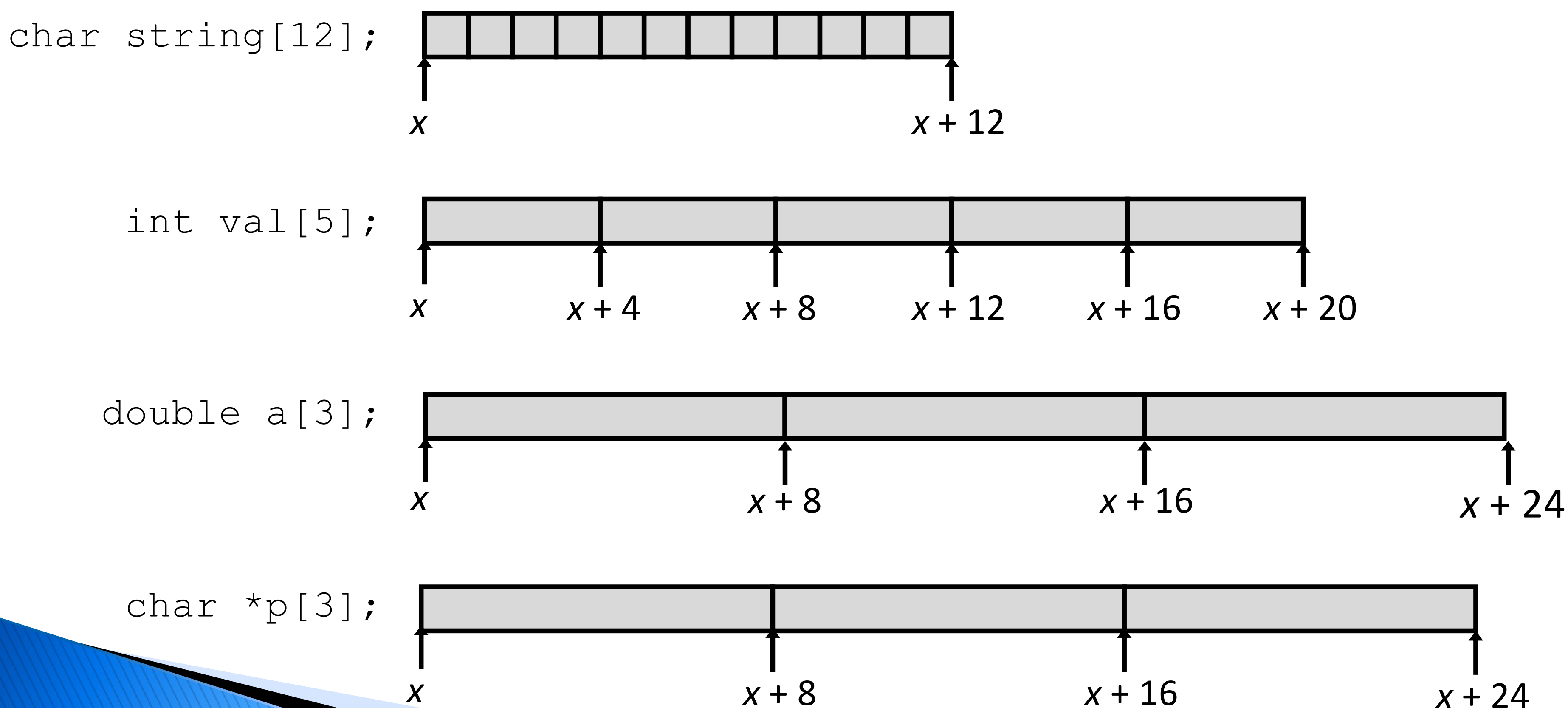
Prof. Soraya Abad-Mota, PhD

Learning objectives

- Describe how are arrays stored and accessed. Practice access to arrays in assembly code. The arrays include one-dimensional, fixed length, multi-dimensional (nested), fixed and variable size. (Section 3.8, 19 slides)
- Describe how are structures stored and accessed. (section 3.9)
- Describe alignment principles and their application to structures. (Subsection 3.9.3)

Array Allocation

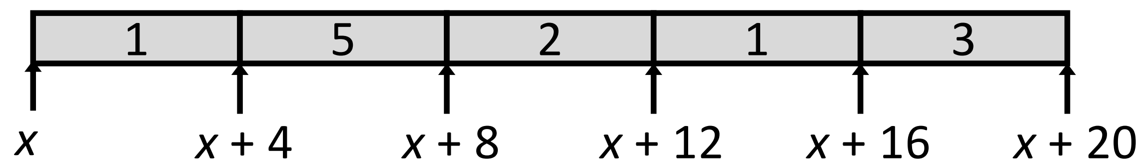
- Basic Principle: *T* **A[*L*];**
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof**(*T*) bytes in memory
  - x is beginning address (of 1st byte) of array



Array Access

- Basic Principle: *T* **A[*L*];**
  - Array of data type *T* and length *L*
  - Identifier **A** can be used as a pointer to array element 0: Type *T**

`int val[5];`



Reference	Type	Value
`val[4]`	`int`	3 (the value at index 4 of array val)
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4 (x + 1 * L)
`&val[2]`	`int *`	*x* + 8 (the address of element at index 2)
`val[5]`	`int`	??
`*(val+1)`	`int`	5 (content of address val + 1)
`val + `*`i`*	`int *`	*x* + 4 *i* *(address val + i * L)*

Practice exercises

- Problem 3.36, p. 256: to understand the size of the array and how to access each element.
- Problem 3.37, p. 258: to practice pointer arithmetic.
- Follow assembly code examples for subsection 3.8.4 p.261
- Follow examples in subsection 3.8.5

Problem 3.36

Array	Elem.size	Total size	Start add.	element i
short S[7]	2 bytes	14	x_s	$x_s + 2*i$
short *T[3]				
short **U[6]				
int V[8]				
double *W[4]				

Problem 3.36

Array	Elem.size	Total size	Start add.	element i
short S[7]	2 bytes	14	x_s	$x_s + 2*i$
short *T[3]	8	24	x_T	$x_T + 8*i$
short **U[6]	8	48	x_U	$x_U + 8*i$
int V[8]	4	32	x_V	$x_V + 4*i$
double *W[4]	8	32	x_W	$x_W + 8*i$

Array Example

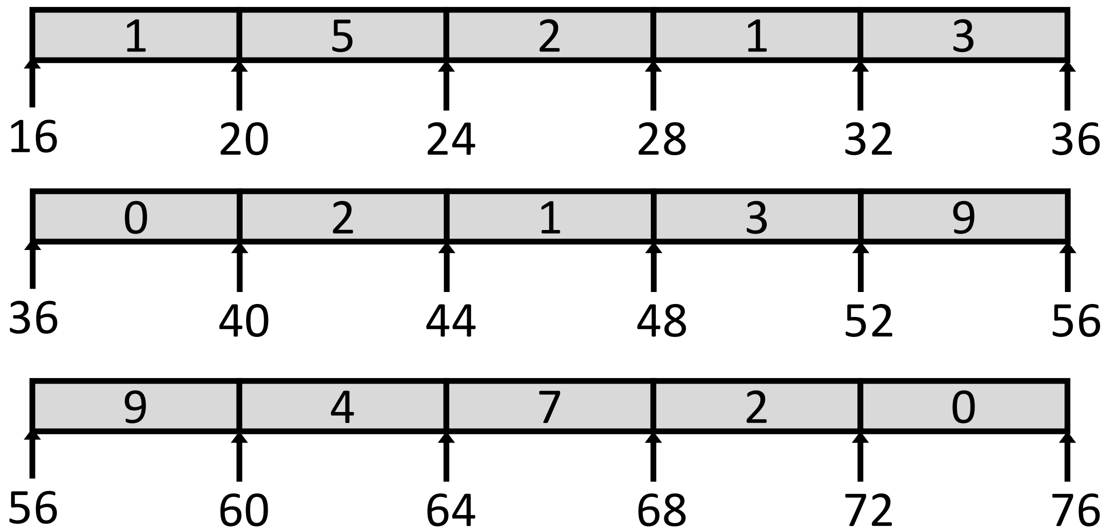
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

Each is one array of 5 integers

`zip_dig cmu;`

`zip_dig mit;`

`zip_dig ucb;`



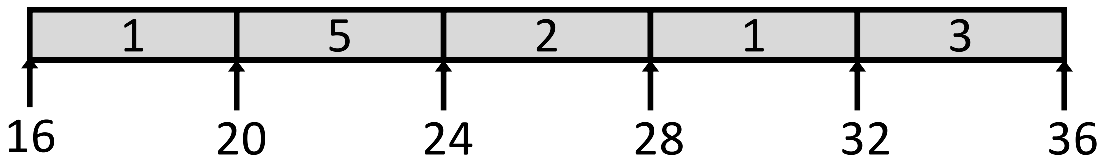
- Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Simple homework exercise

- Write a C program that declares these 3 arrays of any type, make them same size, and check which addresses are used to store them.
- It is very likely that they will not be assigned contiguous blocks, but check the addresses of each array and all its elements.
- Which function can you use to see the address?
  - of each array
  - of each element of an array (notice datatype size)

Array Accessing Example

```
zip_dig cmu;
```




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

IA32

```
  # %rdi = z      (in book x_z )
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

Pointer Arithmetic

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

Array Loop Example (study on your own)

```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   z[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle
  cmpq    $4, %rax            #   i:4
  jbe     .L4                 #   if <=, goto loop
  rep; ret
```

Multidimensional (Nested) Arrays

- Declaration

  T **A**[R][C];

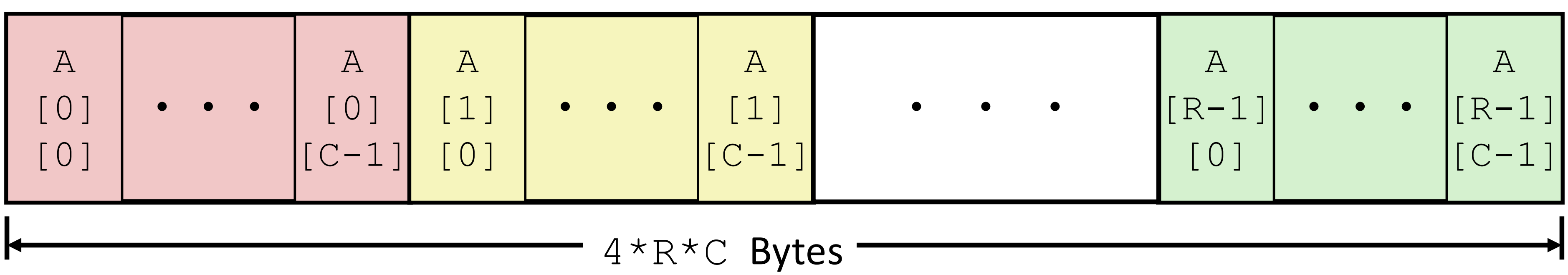
  - 2D array of data type T
  - R rows, C columns
  - Type T element requires K bytes

- Array Size
  - R * C * K bytes

- Arrangement: Row-Major Ordering

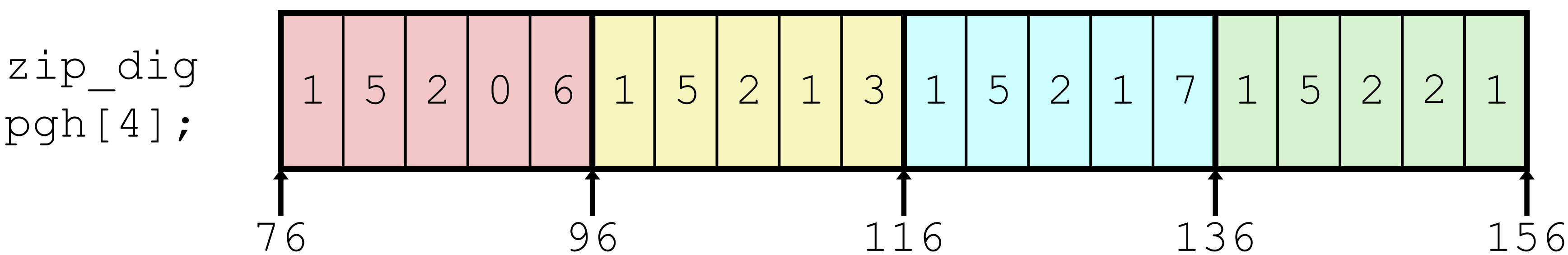
$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

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



Nested Array Example (study on your own)

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

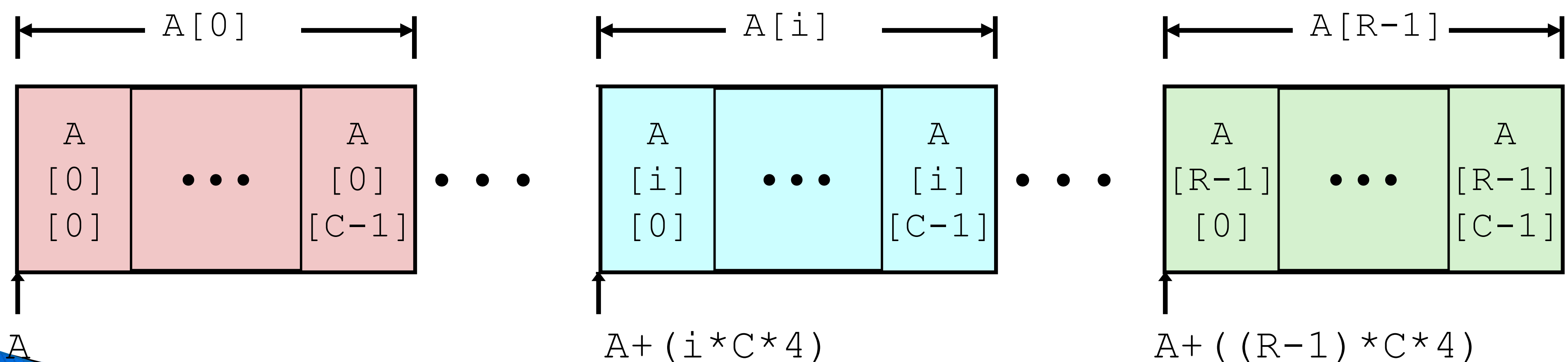


- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- "Row-Major" ordering of all elements in memory

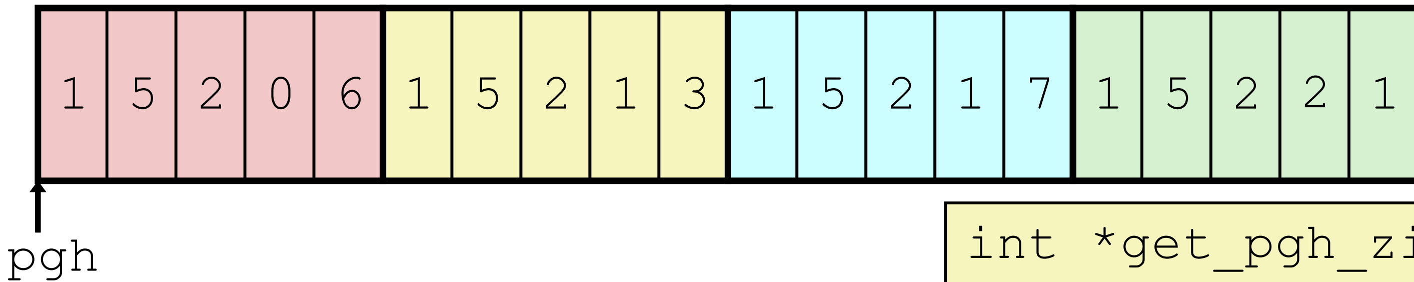
Nested Array Row Access

- Row Vectors
  - **A[i]** is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address **A +** $i * (C * K)$

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



Nested Array Row Access assembly code




```c
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

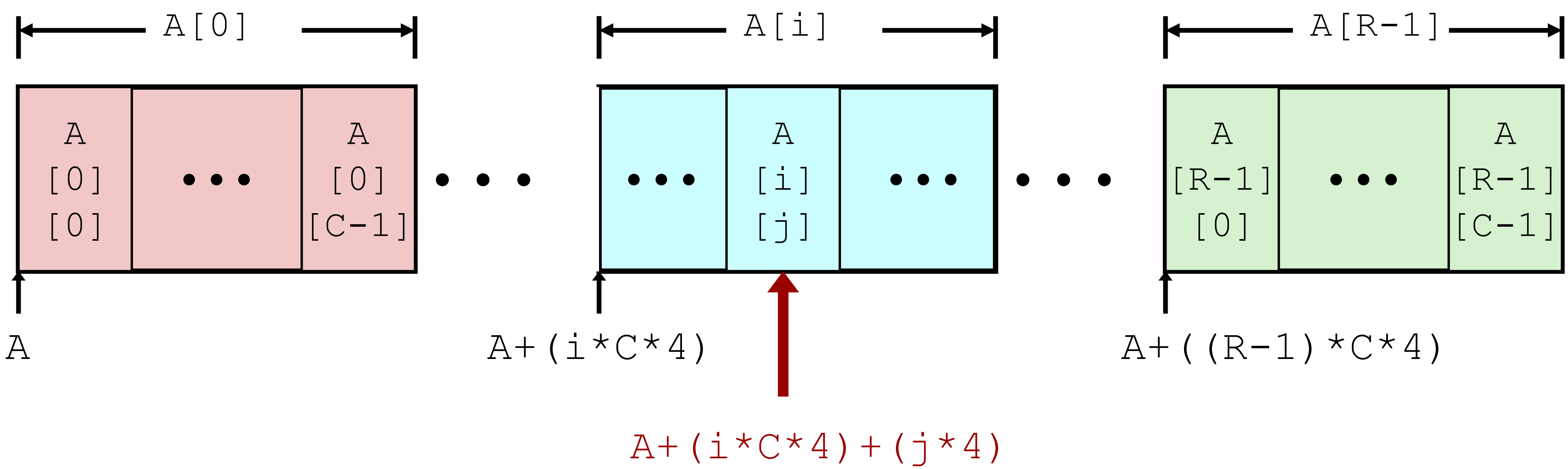
```
  # %rdi = index
  leaq (%rdi,%rdi,4),%rax  # 5 * index
  leaq pgh(,%rax,4),%rax   # pgh + (20 * index)
```

- Row Vector
  - **`pgh[index]`** is array of 5 **`int`**’s
  - Starting address **`pgh+20*index`**
- Machine Code (understand with leaq)
  - Computes and returns address
  - Compute as **`pgh + 4*(index+4*index)`**

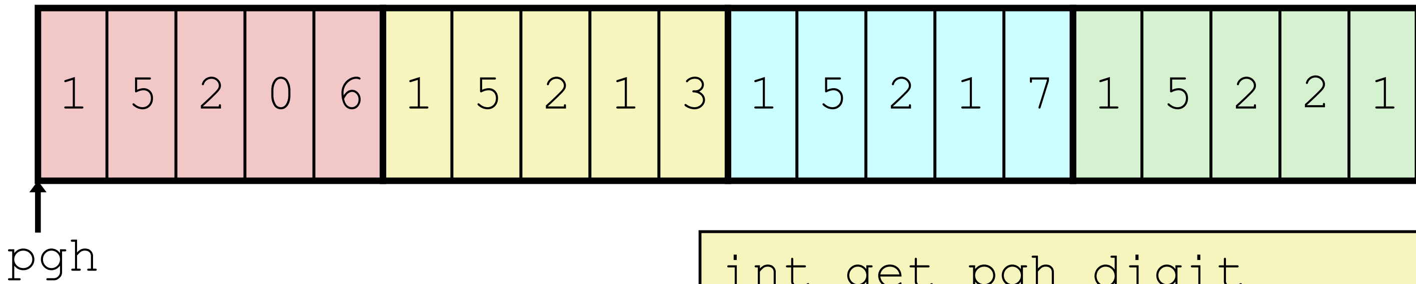
Nested Array Element Access
(derivation of how to access each)

- Array Elements
  - **`A[i][j]`** is element of type *T*, which requires *K* bytes
  - Address **`A +`** $i * (C * K) + j * K = A + (i * C + j) * K$

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



Nested Array Element Access assembly code




```
int get_pgh_digit
   (int index, int dig)
{
   return pgh[index][dig];
}
```

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+dig
movl   pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

- Array Elements
  - **`pgh[index][dig]`** is **`int`** `(each array has 5 elements)`
  - Address: **`pgh + 20*index + 4*dig`**
    - = **`pgh + 4*(5*index + dig)`**

Alternative representation

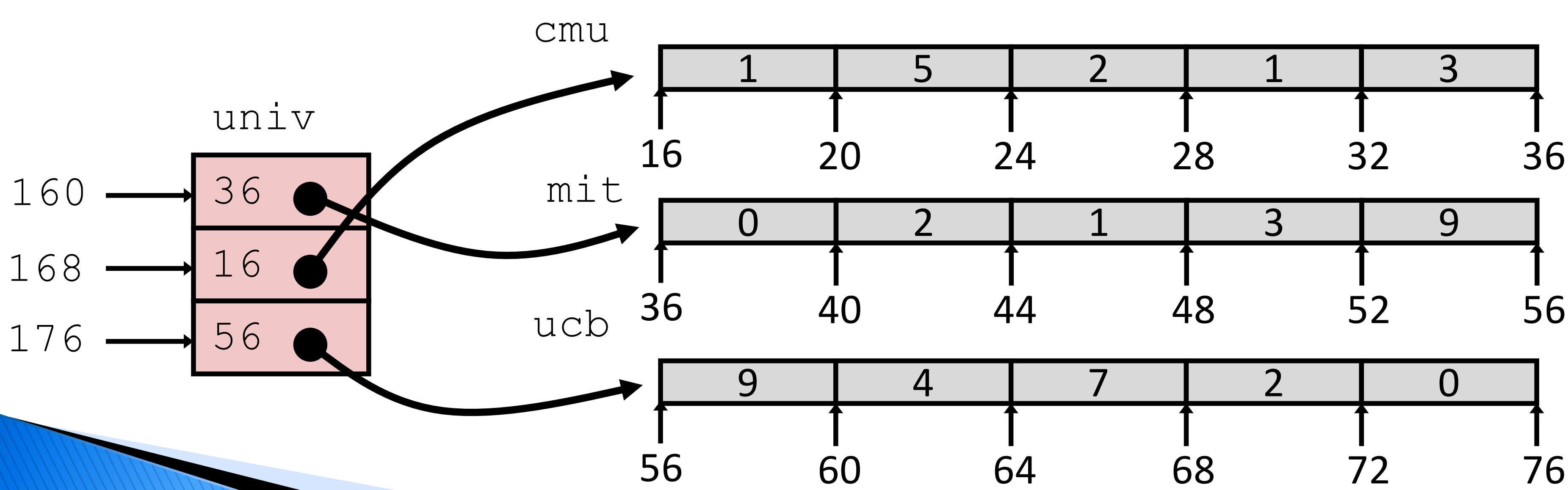
- Before: The vector of zip codes, each of which is an array of 5 elements
- Now: (on the next 3 slides) Multilevel array representation:
  - one vector of pointers
  - each pointer points to an array of 5 elements (zip code digits)
- Study these slides on your own: further practice on how to represent the addresses in the arrays in assembly code
- JUMP to slide 22

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

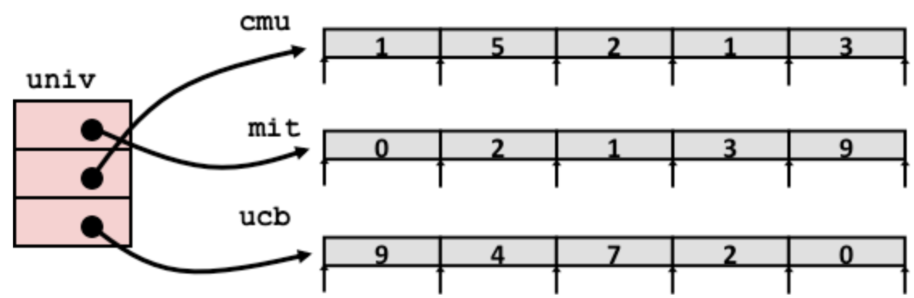
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of `int`'s



Element Access in Multi-Level Array

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




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

- 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

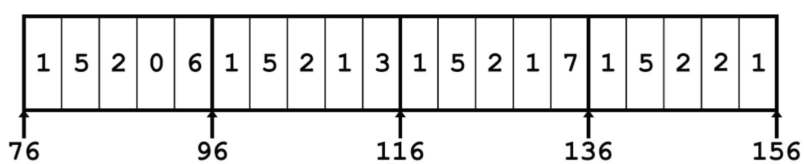
Array Element Accesses

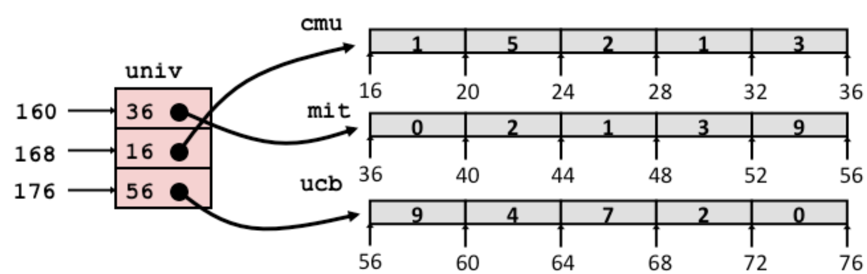
Nested array

```
int get_pgh_digit
  (size_t index, size_t digit)
{
  return pgh[index][digit];
}
```

Multi-level array

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





Accesses looks similar in C, but address computations very different:

Mem[pgh+20*index+4*digit]   Mem[Mem[univ+8*index]+4*digit]