Original slides: Bryant and O'Halloran, Computer Systems: A Programmers Perspective, 3rd Edition

CSE 2421

Array and Structure Storage and Access

Today

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment

Pointer arithmetic

- If p is a pointer to data type T
- And, the value of p (i.e., an address) is x_p
- Then, then p+i has value x_p + L*i
  - where, L is the size of data type T
- Thus for an array A of elements, A[i] == *(A+i)

- Example
  - int E[10]; /*Assume int is 4 bytes long */
  - Suppose rdx holds starting address of array E
  - Suppose rcx holds integer index i

C expression	Type	Assembly code… result in eax	Comment
E	int *	movq %rdx, %rax	
E[i]	int	movl (%rdx,%rcx,4),%eax	Reference memory
&(E[i])	int *	leaq (%rdx,%rcx,4),%rax	Generate address
E+i-1	int *	leaq -4(%rdx,%rcx,4),%rax	Generate address
*(E+i-3)	int	movl -12(%rdx,%rcx,4),%eax	Reference memory

Arrays

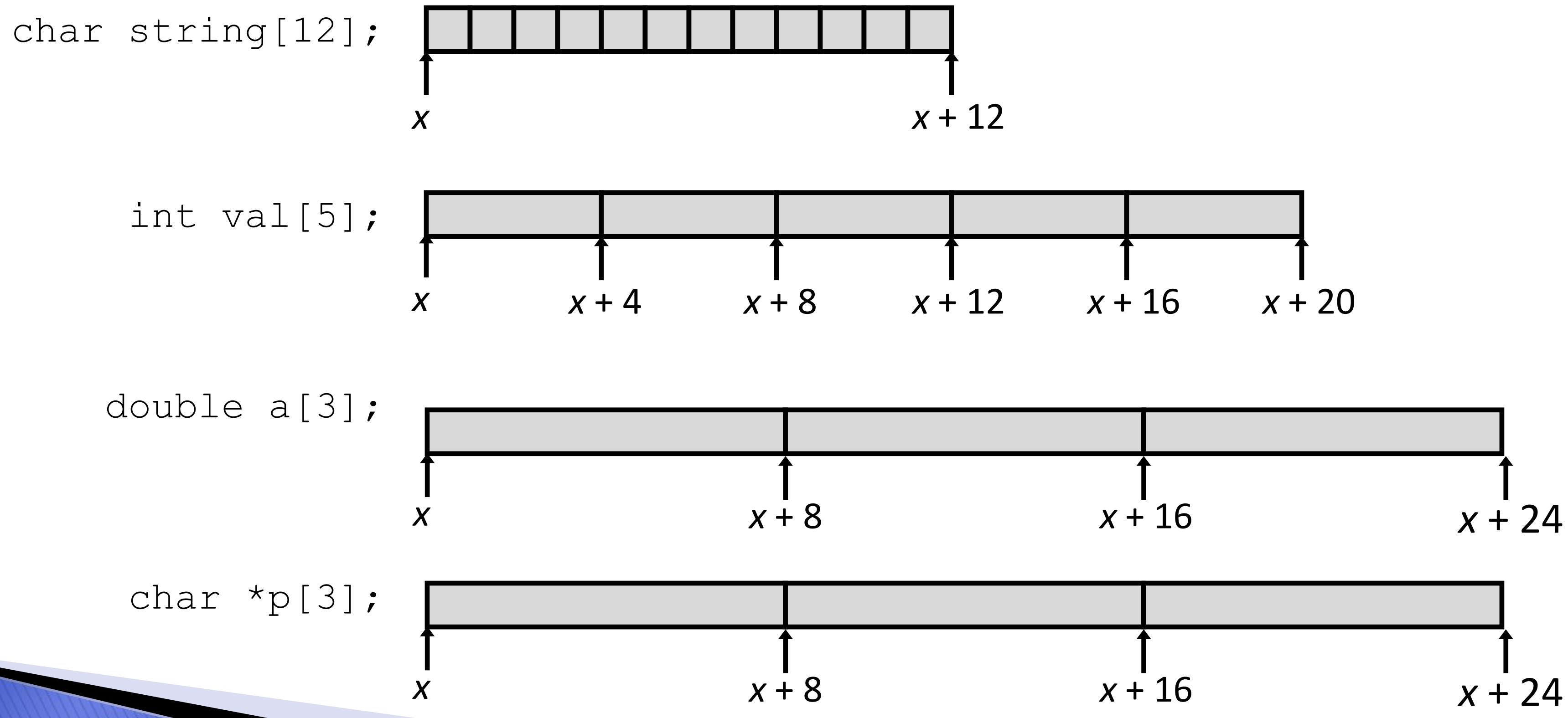
- C declaration: *type* array[length]
- Arrays are means for storing multiple data objects of the same type
- Stored sequentially, often accessed as an offset from a pointer which points to the beginning of the array.
  - size = length*sizeof(type)
- If x is the address of the first byte of the first element in the array, then array element i will be stored at address x+sizeof(type)*i

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



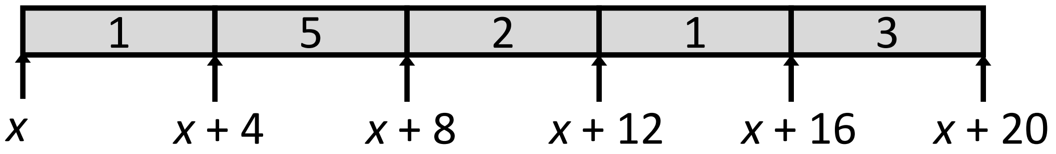
Array Access

- 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];
```

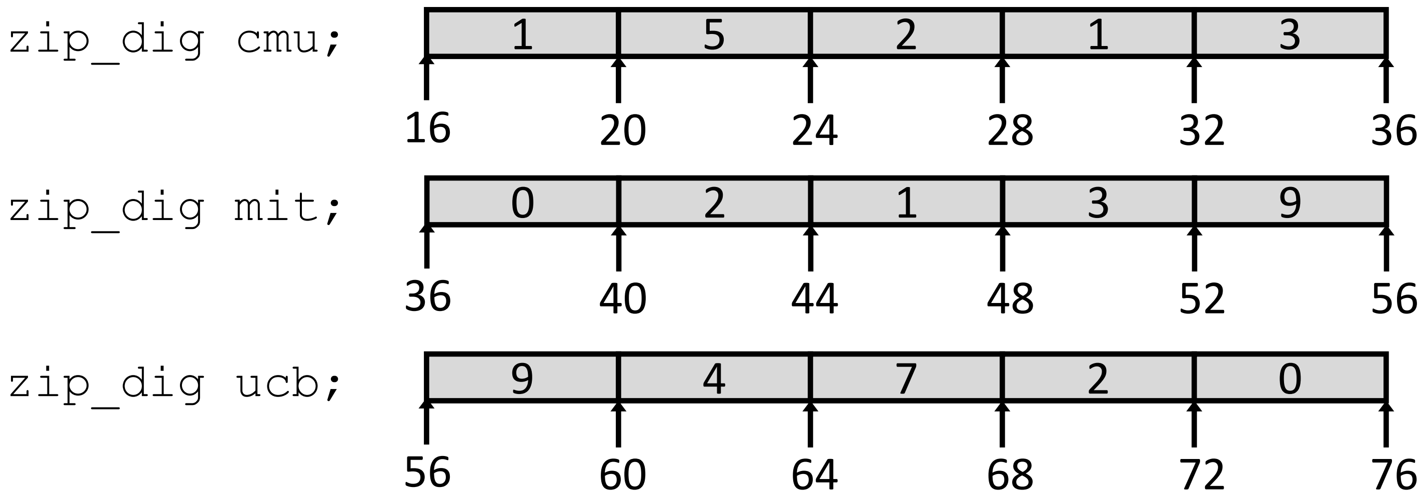


- Reference / Type / Value

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

Array Example

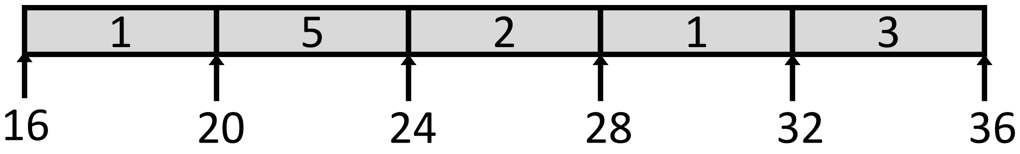
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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



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

Array Accessing Example

```
zip_dig cmu;
```




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

X86-64

```
  # %rdi = z
  # %rsi = 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`
- Use memory reference `(%rdi,%rsi,4)`
- use movl instruction to move 4 bytes
- Use 4 byte register %eax

Array Loop Example

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

```
  # %rdi = z
  movq    $0, %rax          #   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
  jle     .L4               #   if <=, goto loop
  ret                       # ret
```

Arrays – Example 1

- Consider the following C code:

```
static int x, array[30];
x = array[25];
```

- Which is equivalent to assembly code:

REMINDER: $ in assembly language with a label gives an address. array and x must have been defined in the data segment (.data section) of the program.

```
movq $array, %rbx              #rbx is base register
movq $25, %rcx                 #rcx is index register
movl (%rbx,%rcx,4),%eax        #eax = array[25]
movl %eax, x                   #x = array[25] Note: no $ - Why?
```

Arrays – Example 2 – dynamic arrays

```
C code:
int MyFunction1()
{
  int data[20];
  ...
}

MyFunction1:
pushq %rbp
movq %rsp, %rbp
subq $80,%rsp       #Allocate space for array on the stack: 20 elements,
                    #4 bytes each
leaq (%rsp), %rax   #using %rax as base register
                    #OR movq %rsp, %rax
...
```

Arrays – Example 3

```
C code
void MyFunction2()
{
      char buffer[12];
       ...
}

MyFunction2:
 pushq %rbp
 movq %rsp, %rbp
 subq $12,%rsp           #allocate 12 bytes for array
 leaq (%rsp), %rax       #rax is base register
                         #OR movq %rsp, %rax
...
```

Array – Example 3

```
MyFunction2:
 pushq %rbp
 movq %rsp, %rbp
 subq $12, %rsp
 leaq (%rsp), %rax  #OR movq %rsp, %rax
 ...
```

What happens if rcx has 16, and the code tries to access (%rax,%rcx,1)? For example, suppose dl has 5, and this instruction is executed:

movb %dl, (%rax,%rcx,1)

Array – Example 3

```
MyFunction2:
 pushq %rbp
 movq %rsp, %rbp
 subq $12, %rsp
 leaq (%rsp), %rax   #OR movq %rsp, %rax
 ...
```

What happens if rcx equals16, and the code tries to access (%rax,%rcx,1)? For example, suppose %dl equals 5, and this instruction is executed:

movb %dl, (%rax,%rcx,1)

Buffer Overflow!

Arrays On the Stack – cont

- Look for large allocation on the stack
- Look for data references using a register other than rsp/rbp as the base

```
StackArrayEx:
pushq %rbp
movq %rsp, %rbp
subq $520, %rsp
pushq %rbx      # save callee-saved registers before use
leaq 8(%rsp), %rbx  # the array is higher than pushed rbx
                # leaq -520(%rbp), %rbx also works
movl $0x0,(%rbx)    #set first element to 0
```

Arrays On the Stack – initialization

- For the dynamic array on the preceding slide, how could the compiler generate code for a loop to initialize all of the array elements to 0?

```
StackArrayEx:
    pushq %rbp
    movq %rsp, %rbp
    subq $520, %rsp        # make space for the array
    pushq %rbx                     #push callee saved register
    leaq 8(%rsp), %rbx             #base register (array is above pushed rbx)
    movq $0, %rcx                  #index register
initialize:
    cmpq $130, %rcx                # alternative loop saves 260 instructions
    je      next                   # movq $129, %rcx
    movl $0x0,(%rbx,%rcx,4)        # backwards:
    incq    %rcx                   # movl $0x0,(%rbx,%rcx,4)
    jmp     initialize             # decq %rcx
next:                              # jge backwards
```

Arrays On the Stack – cleanup

- For the dynamic array on the preceding slides, what needs to happen with respect to cleanup before return?

```
StackArrayEx:
        pushq %rbp
        movq %rsp, %rbp
        subq $520, %rsp
        pushq %rbx
        leaq 8(%rsp), %rbx          #base register for array.
. . . .                        #other code (omitted)
Return:
        popq %rbx     # last thing we pushed, restore old value
        movq %rbp, %rsp     # these 2 are the same as leave
        popq %rbp
        ret
```

Why order things in the stack that way?

- We put RBP, the array, and then the other callee-saved registers (RBX) in the stack in that order
- It makes cleanup simpler
  - The callee-saved registers (other than rbp) are at the top of the stack so we can pop them off in reverse order of the pushes without doing any math to figure out where they are
  - Then the leave instruction (or the 2 instruction equivalent) cleans up the array and gets the pushed RBP back in place – without caring about how many bytes were allocated for the array
- We paid for simple cleanup by having a more complex assignment for the array pointer
  - leaq 8(%rsp), %rbx   #specific to this example -or-
  - leaq -520(%rbp), %rbx # general method for any allocated space
- Assumes we need to use a callee-saved register

Arrays on the heap

- "Global" Arrays
- Arrays of elements with initial values of 0 by default
  - If stored in the data section of application (i.e., static arrays)
- Accessed through a memory address

```
MemArrayEx:
 pushq %rbp
 movq %rsp, %rbp
 pushq %rbx
 pushq %r12
 movq $staticArray, %r12   #base register
 movq $0x0, %rbx           #index register
 movl $0x0,(%r12,%rbx,4)   #set 1st element to 0
```

Arrays on the heap – alternative

- "Global" Arrays – the compiler knows where they go in the heap
- So a label is available to use for the address

```
MemArrayEx:
 pushq %rbp
 movq %rsp, %rbp
 pushq %rbx
 movq $0x0, %rbx             #index register
     # use the label as a displacement (no $)
     # and we won't need a base register
movl $0x0, staticArray(,%rbx,4) #set 1st element to 0
```

Arrays

- If an array holds elements larger than 1 byte, the index will need to be multiplied by the size of the element

```
#access to array of elements of size 4, with
#scaling, where rax holds the index i, and rbx is
#the base register:
#e.g., arr[i] = 11223344
movl $11223344,(%rbx,%rax,4)
...
```

Arrays of size larger than 8 bytes

- What if the array holds elements larger than 8 bytes? For example, what if it is an array of structures?
- Recall that, in x86-64, scaling factors are to 1, 2, 4, or 8
- Therefore, for arrays with elements larger than 8 bytes, manual scaling must be used

```
#Here, two index registers are used, one
#for the conventional index (here, rcx),
#and one for a scaled index register,
#here, rax. This is “manual scaling.”
movq $0, %rcx
                        # signed multiply
                        # imulq aux, src, Dest
imulq $20,%rcx,%rax           # manually scale index
#Suppose we want: ptr = &arr[i]
leaq (%rbx,%rax), %rdx
```

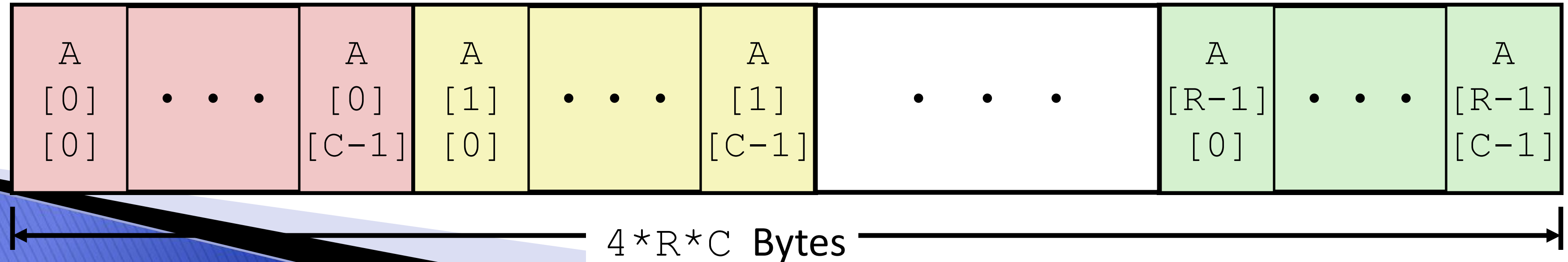
Multidimensional (Nested) Arrays

- 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

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

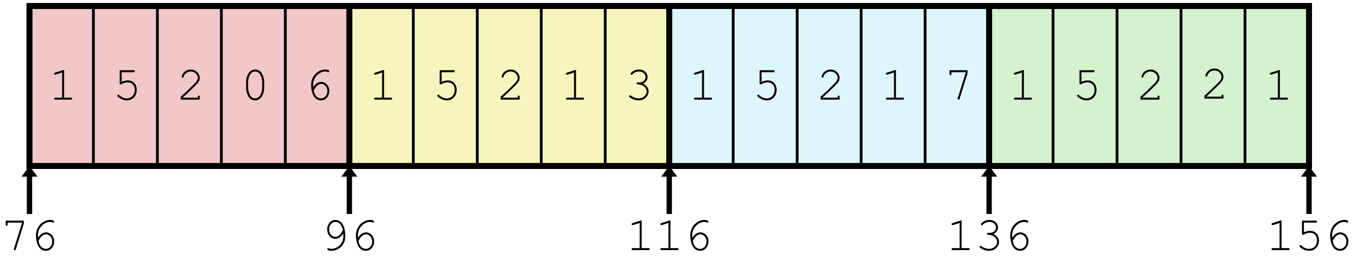
```
int A[R][C];
```



Nested Array Example

```
#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];`

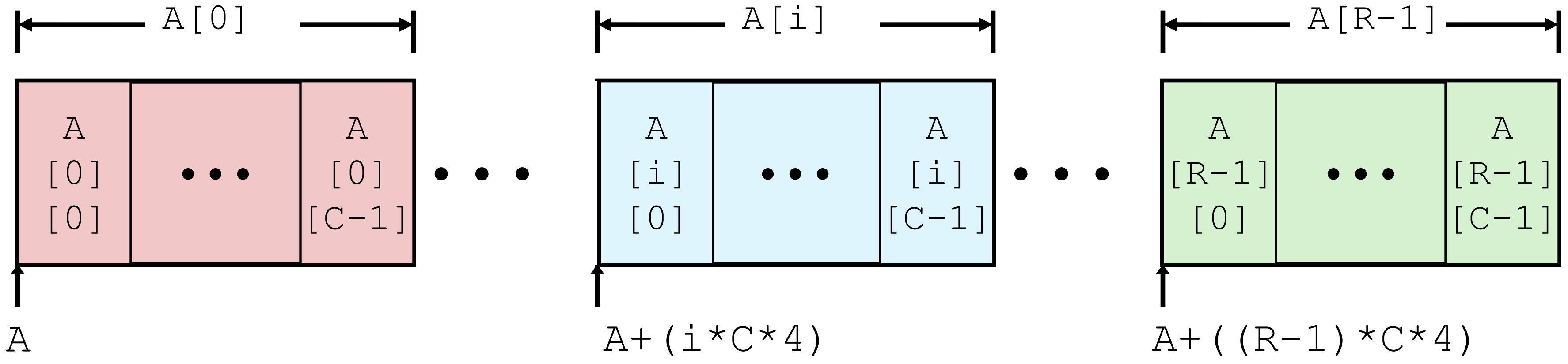


- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable **`pgh`**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **`int`**'s, allocated contiguously
- "Row-Major" ordering of all elements in memory

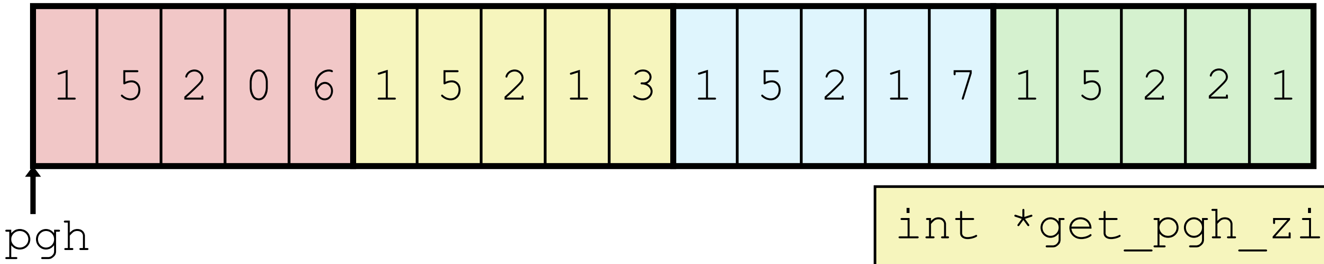
Nested Array Row Access

- 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 Code




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

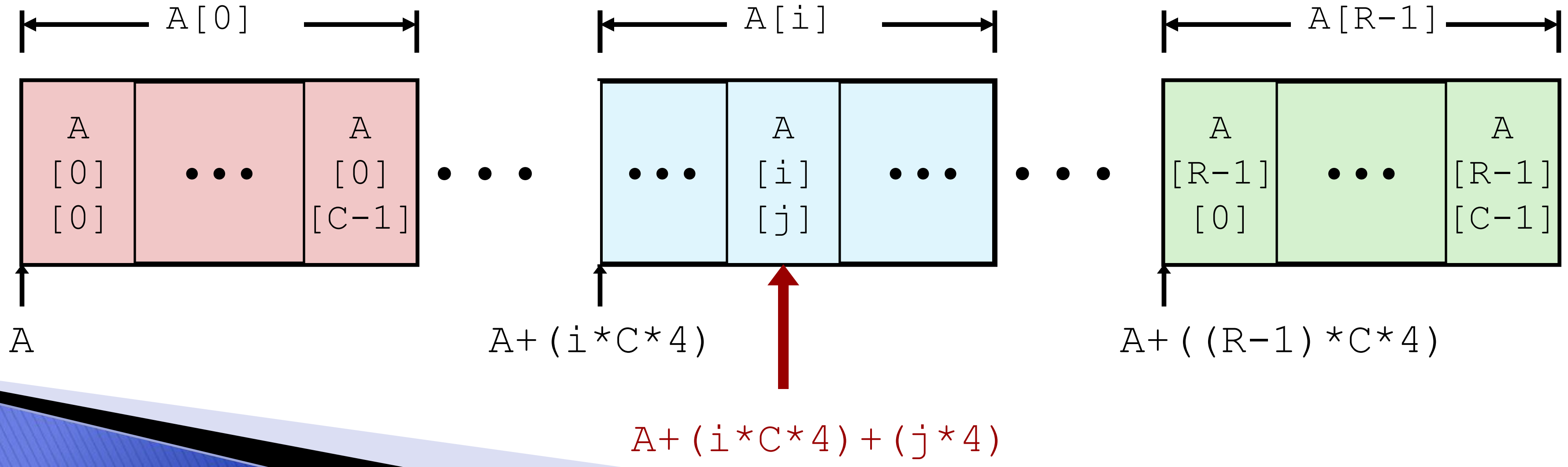
```
  # %rdi = index
  leaq (%rdi,%rdi,4),%rax  # 5 * index
  leaq pgh(,%rax,4),%rax   # pgh + (20 * index)
```

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

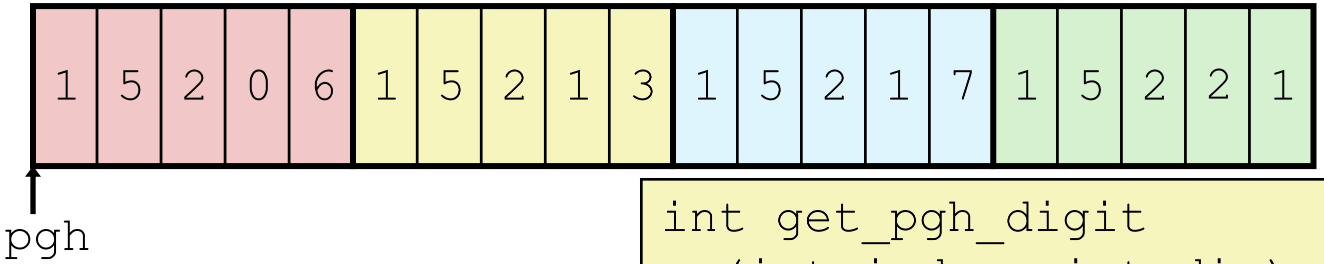
Nested Array Element Access

- 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 Code




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

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
                              # %rsi => 2nd parameter
addq   %rax, %rsi             # 5*index+dig
movl   pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

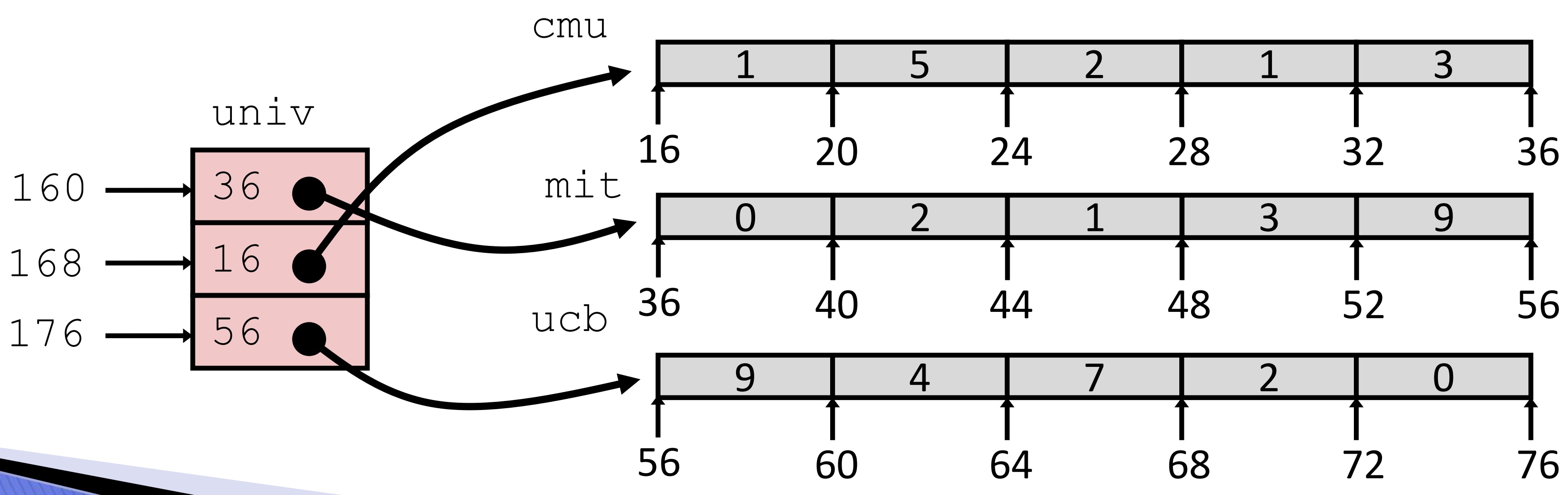
- Array Elements
  - **`pgh[index][dig]`** is **`int`**
  - Address: **`pgh + 20*index + 4*dig`**
    - = **`pgh + 4*(5*index + dig)`**

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 };
```

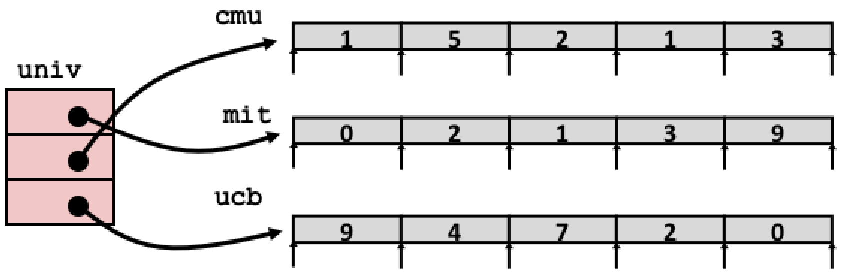
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- 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 (2nd parameter)
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

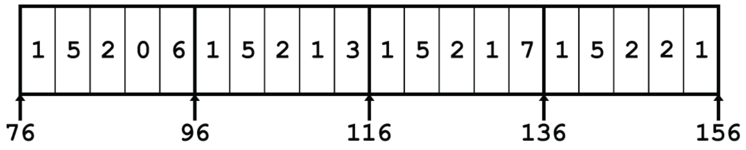
Array Element Accesses

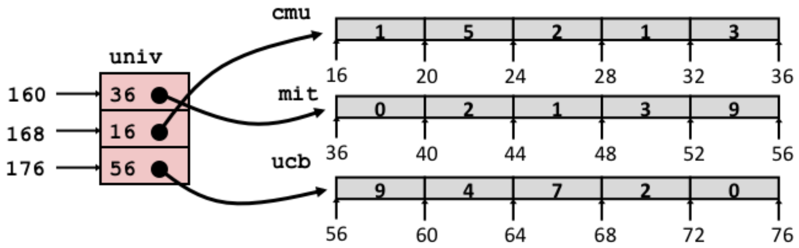
Nested array

```
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]`**