April 25: Arrays

- **Announcements**
  - Lab 2 due Wednesday
  - HW 2 due Friday
  - Midterm next Monday
    - Done with material in lecture (goes through last Friday)
  - Extra office hours tomorrow?

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:

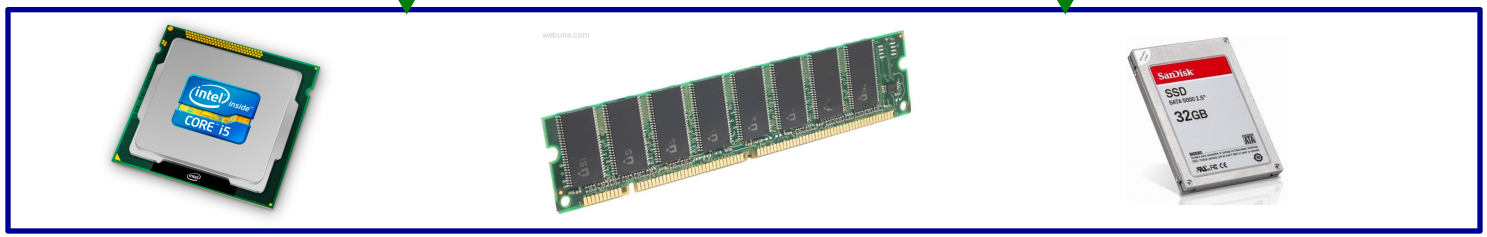
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100001111
```

OS:

Computer system:



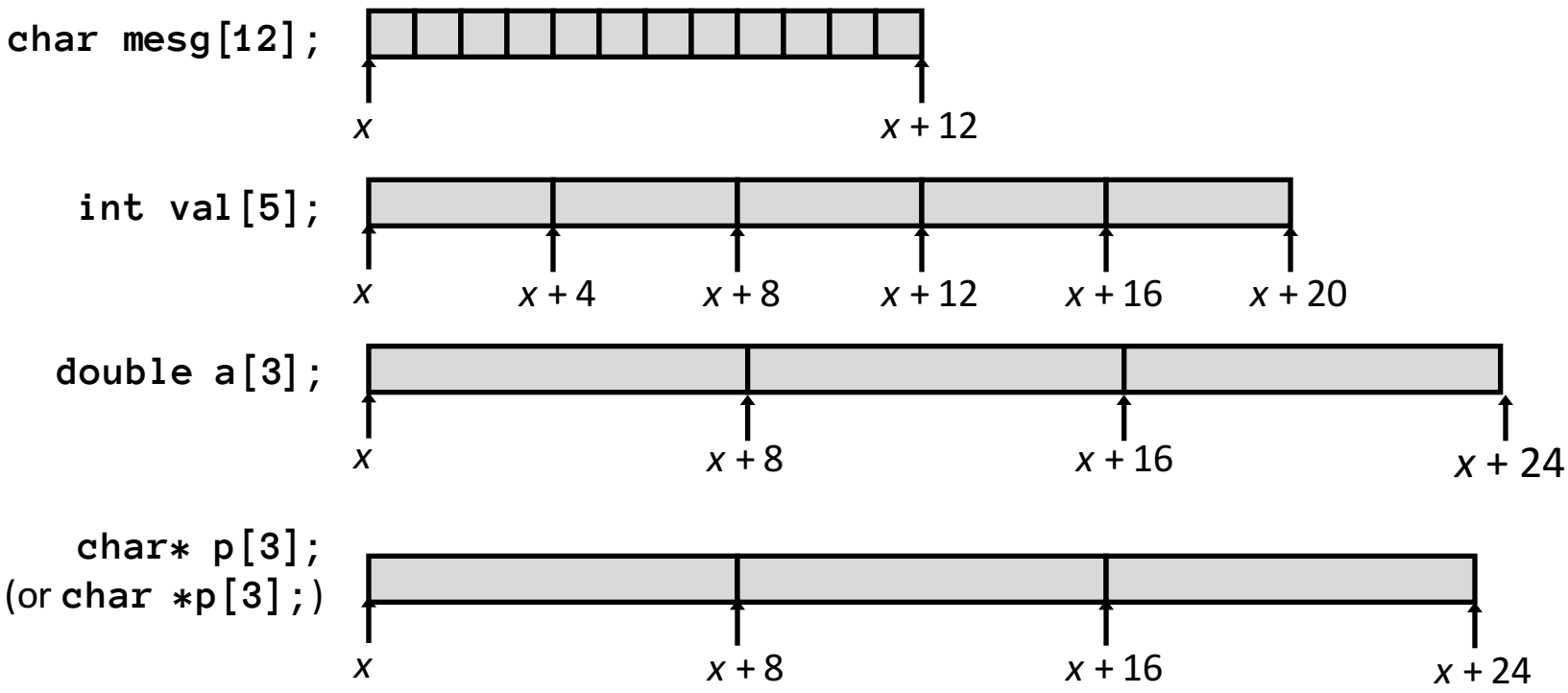
Data Structures in Assembly

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

- **Also: Some C details and how they relate to Java and assembly**
  - C arrays are convenient but with some unique/strange rules

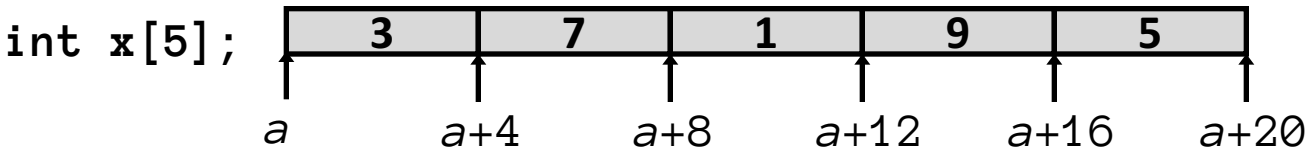
Array Allocation

- **Basic Principle**
  - T A[N];
  - Array of data type T and length N
  - *Contiguously* allocated region of N * sizeof(T) bytes
  - Identifier A can be used as a pointer to array element 0: Type T*



Array Access

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



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

Passing Arrays To Functions

- **Arrays are passed as a pointer**
  - Remember arrays are *implicitly* converted to `&a[0]` when needed
  - Must specify length as well
  - *Confusing* alternative syntax *looks* like it's passing an array, but it's *still* just passing a pointer (*size* is just a "hint")

```
int last(int * a, int len)
{
  return a[len-1];
}
```

```
int last(int a[5], int len)
{
  return a[len-1];
}
```

```
int last(int a[], int len)
{
  return a[len-1];
}
```

```
int main() {
  int x[5];
  printf("%d\n", last(x, 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 };
```

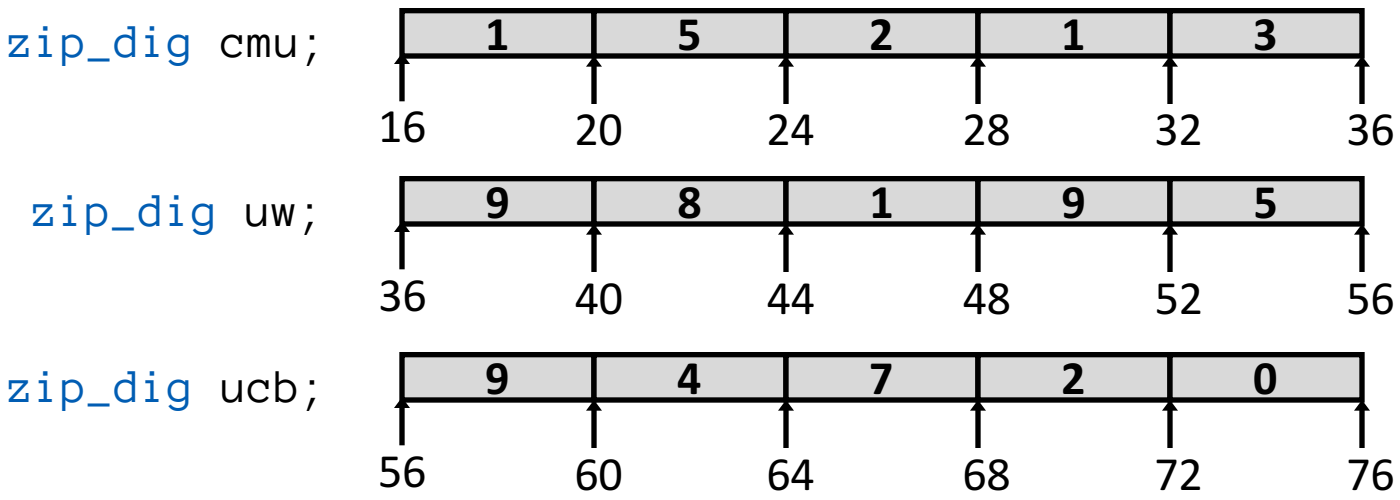
initialization

`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 };
```

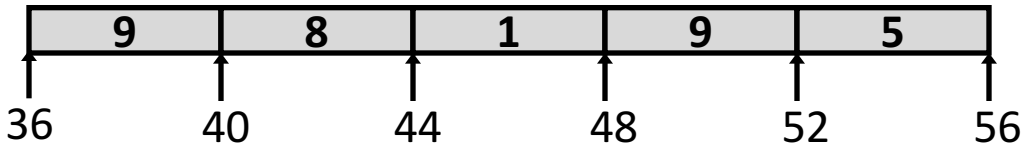


- **Declaration "`zip_dig uw`" equivalent to "`int uw[5]`"**
- **Example arrays happened to be allocated in successive 20 byte blocks**
  - Not guaranteed to happen in general

Array Accessing Example

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

```
zip_dig uw;
```




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

Assembly

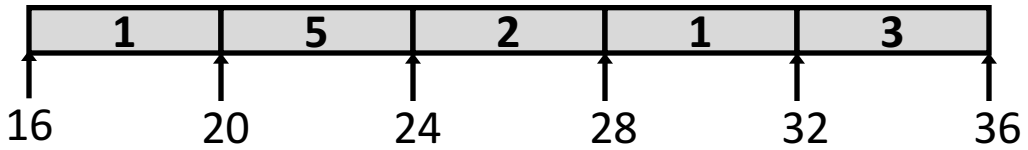
```
  # %rdi = z
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

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

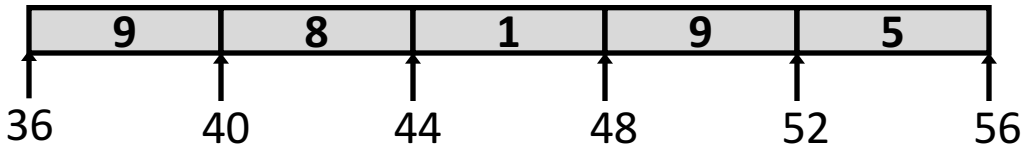
```
typedef int zip_dig[5];
```

Referencing Examples

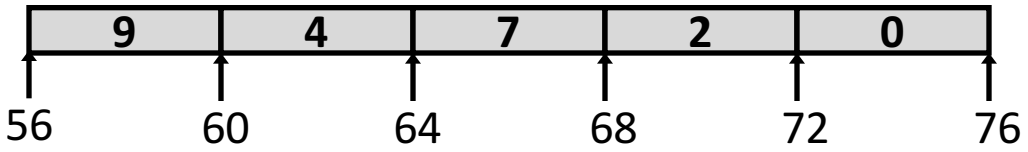
```
zip_dig cmu;
```




```
zip_dig uw;
```




```
zip_dig ucb;
```



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

  `uw[3]`

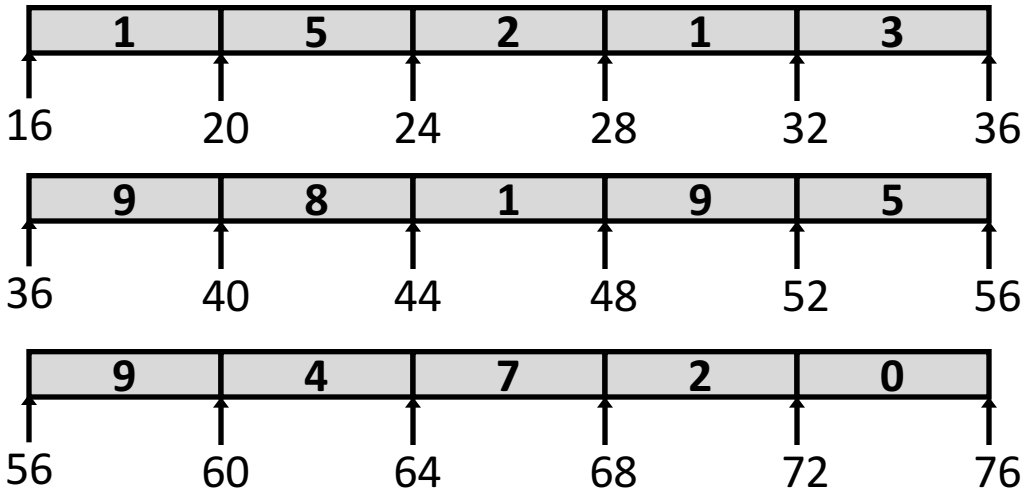
  `uw[6]`

  `uw[-1]`

  `cmu[15]`

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

Referencing Examples



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

- 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];
```

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

zi = 10*0 + 9 = 9

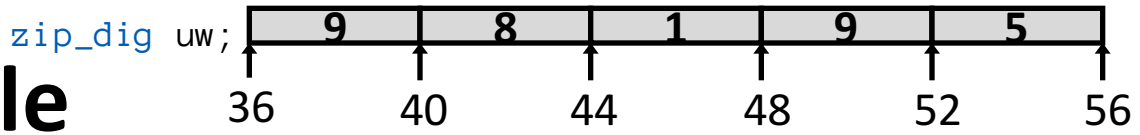
zi = 10*9 + 8 = 98

zi = 10*98 + 1 = 981

zi = 10*981 + 9 = 9819

zi = 10*9819 + 5 = 98195

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**
  - 10*zi + *z implemented as: *z + 2*(5*zi)
  - z++ increments by 4 (size of int=4)

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

```
   # %ecx = z
   movl $0,%eax                 # rax = zi = 0
   leaq 20(%rdi),%rcx           # rcx = zend  = z+5
.L17:
   leal (%rax,%rax,4),%edx      # zi + 4*zi = 5*zi
   movl (%rdi),%eax             # eax = *z
   leal (%rax,%rdx,2),%eax      # zi = *z + 2*(5*zi)
   addq $4,%rdi                 # z++
   cmpq %rcx,%rdi               # z : zend
   jne .L17                     # if != goto loop
```

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

same as:

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

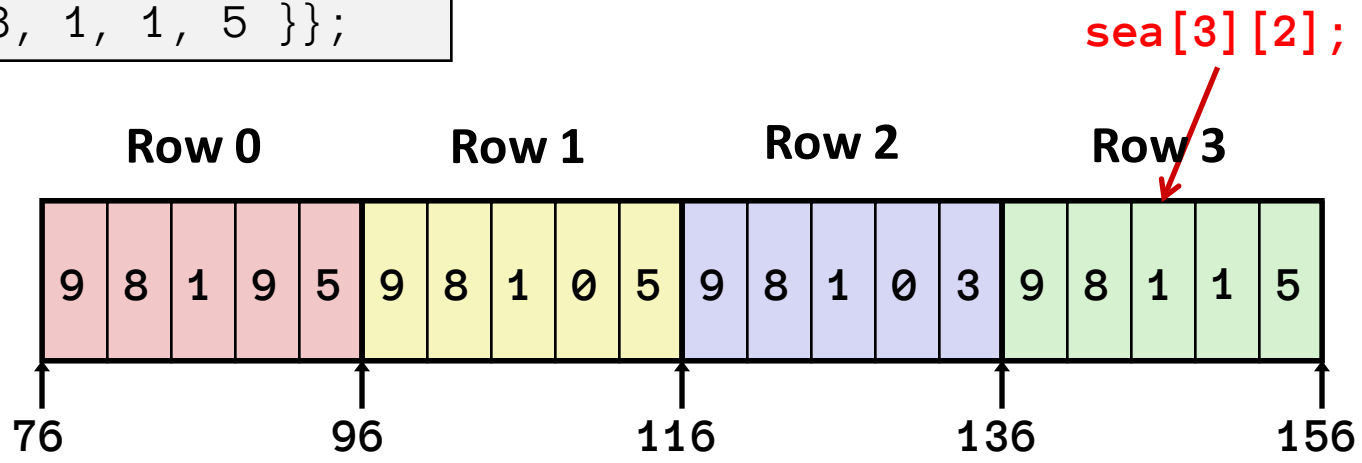
What is the layout in memory?

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 }};
```

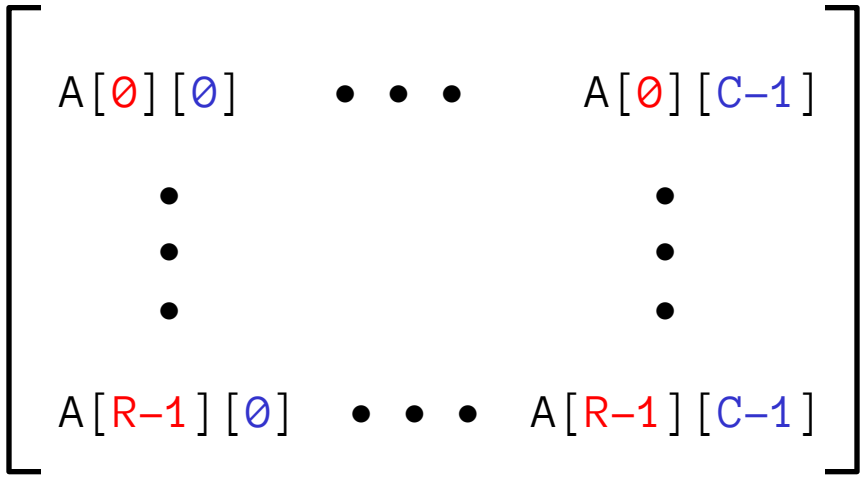
Remember, `T A[N]` is an array with elements of type `T`, with length `N`



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

Two-Dimensional (Nested) Arrays

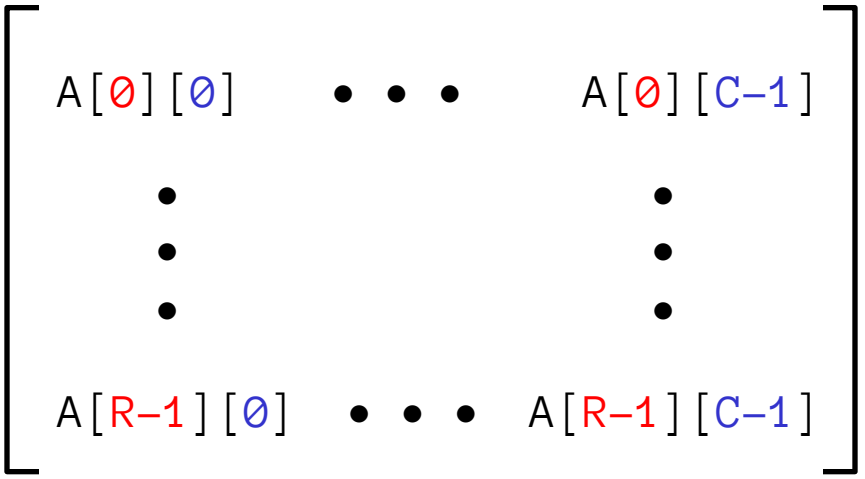
- **Declaration**
  - `T A[R][C];`
  - 2D array of data type T
  - R rows, C columns
  - Each element requires `sizeof(T)` bytes
- **Array size?**

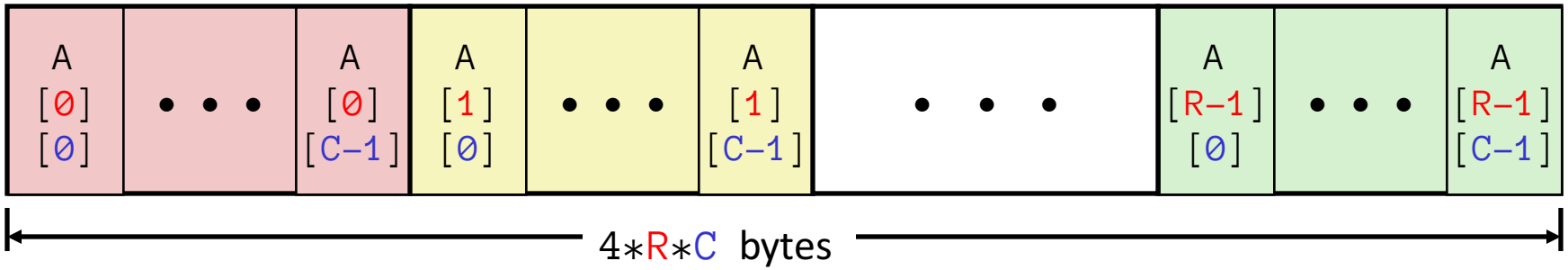


Two-Dimensional (Nested) Arrays

- **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];`

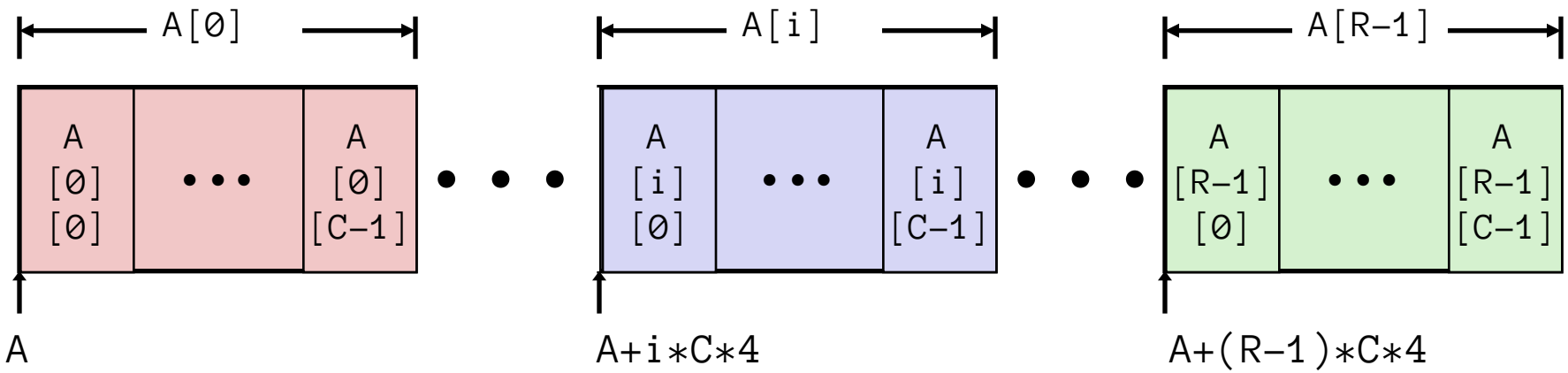




Nested Array Row Access

- **Row vectors**
  - Given: `T A[R][C]`:
    - `A[i]` is an array of `C` elements, "row i"
    - Each element of type `T` requires *K* bytes
    - `A` is starting 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

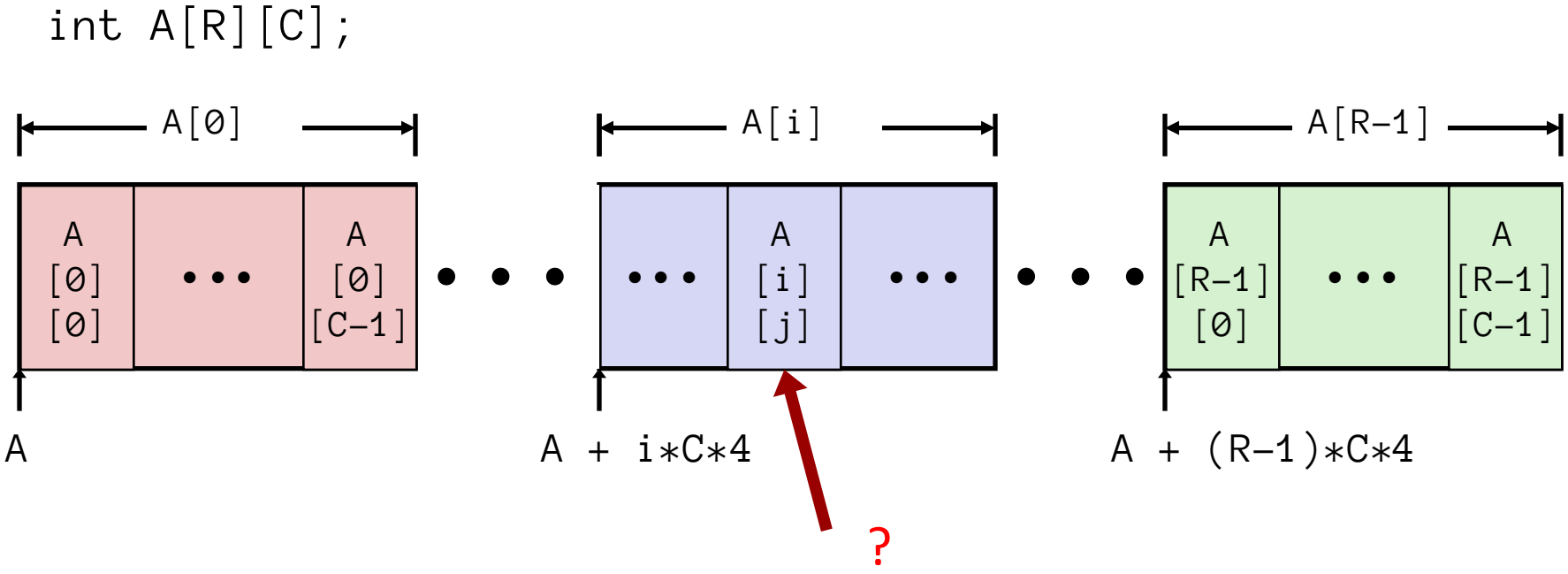
```
int* get_sea_zip(int index)
{
  return sea[index];
}
```

```
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 **int**s
  - 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

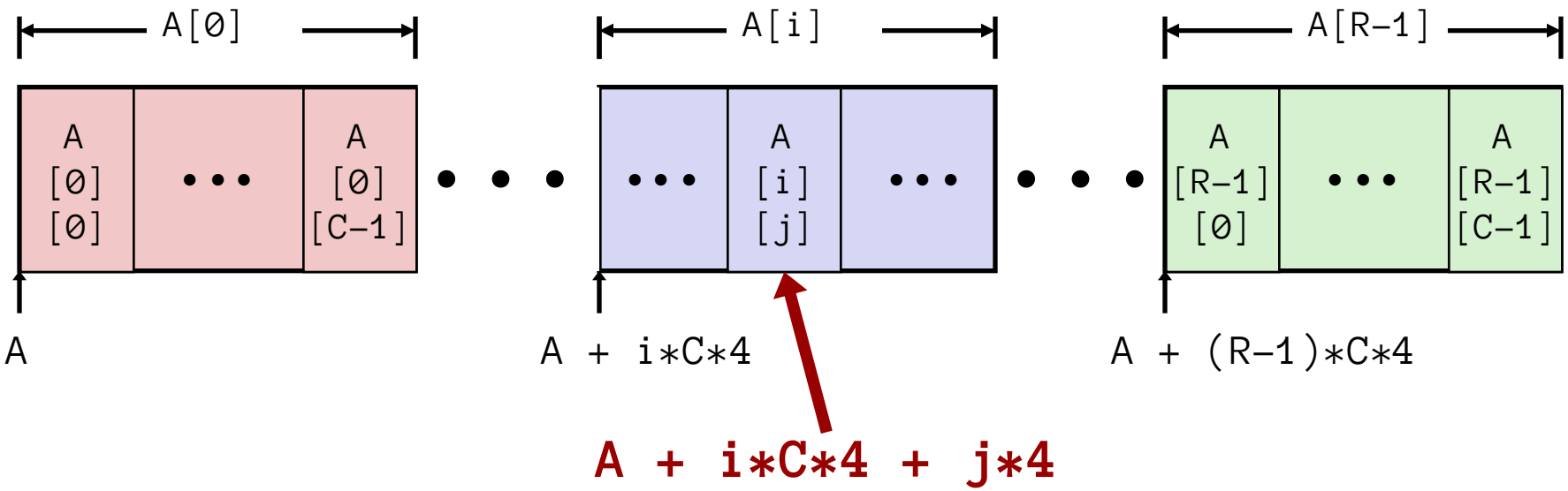


Nested Array Element Access

- **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];
```



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 }};
```

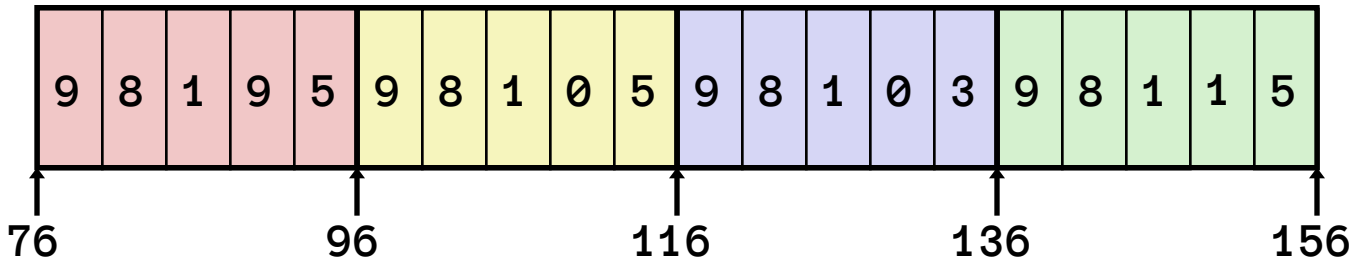
```
leaq   (%rdi,%rdi,4), %rax     # 5*index
addl   %rax, %rsi              # 5*index+digit
movl   sea(,%rsi,4), %eax      # *(sea + 4*(5*index+digit))
```

- **Array Elements**
  - `sea[index][digit]` is an `int` (`sizeof(int) = 4`)
  - Address = `sea + 5*4*index + 4*digit`
- **Assembly Code**
  - Computes address as: `sea + ((index+4*index) + digit)*4`
  - **`movl`** performs memory reference

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

Strange Referencing Examples

```
zip_dig sea[4];
```



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

  `sea[3][3]`

  `sea[2][5]`

  `sea[2][-1]`

  `sea[4][-1]`

  `sea[0][19]`

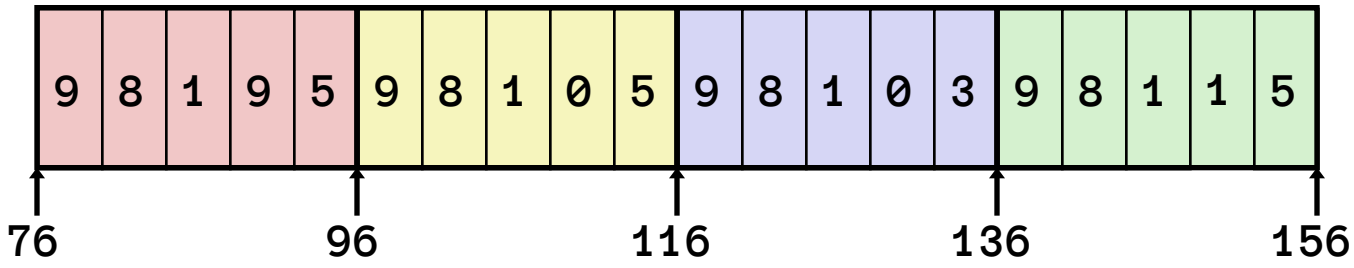
  `sea[0][-1]`

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

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

Strange Referencing Examples

```
zip_dig sea[4];
```



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+4*-1 = 112	5	Yes
sea[4][-1]	76+20*4+4*-1 = 152	5	Yes
sea[0][19]	76+20*0+4*19 = 152	5	Yes
sea[0][-1]	76+20*0+4*-1 = 72	??	No

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

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

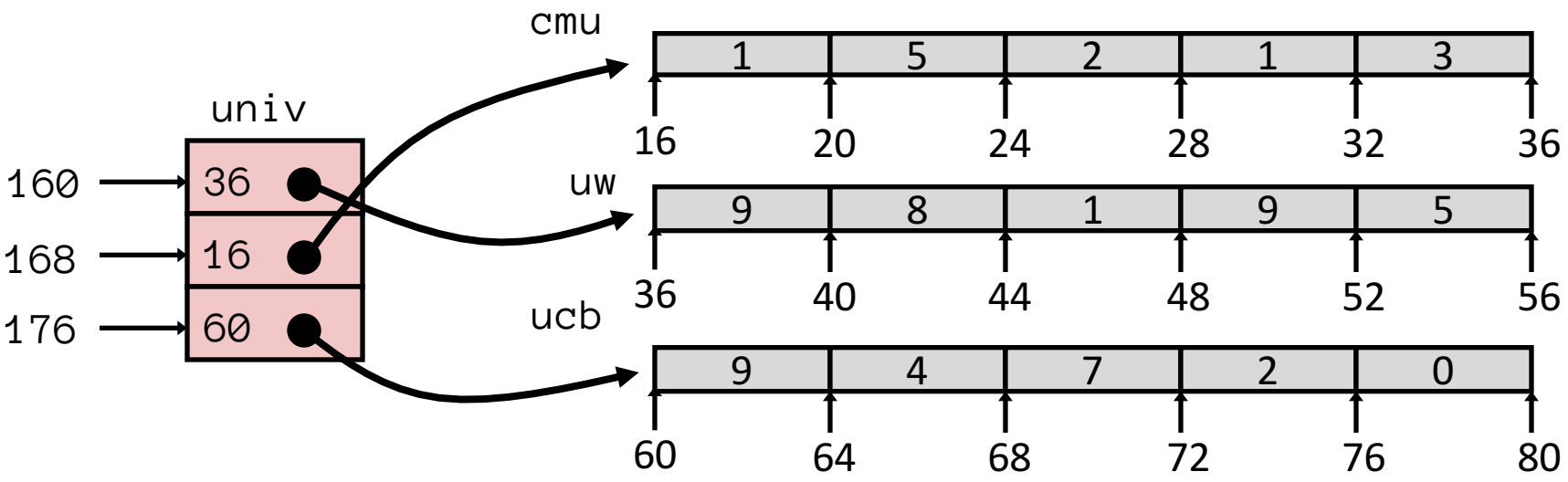
One array declaration = one contiguous block of memory

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};
```

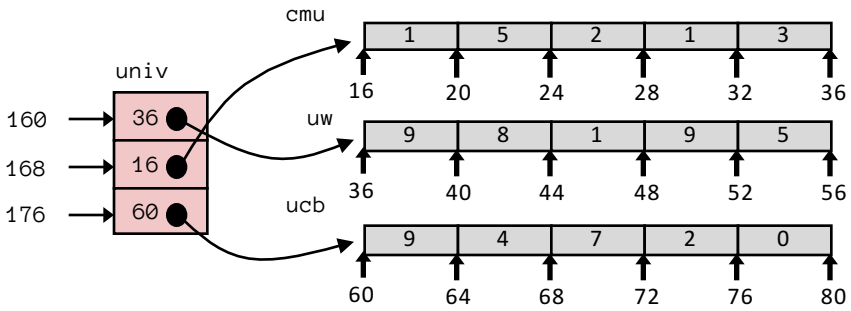
- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes each
- Each pointer points to array of `int`s



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];
}
```



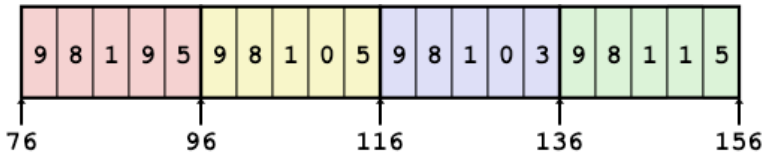

```
salq    $2, %rsi              # rsi = 4*digit
addq    univ(,%rdi,8), %rsi   # p = univ[index] + 4*digit
movl    (%rsi), %eax          # return *p
ret
```

- **Computation**
  - Element access **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

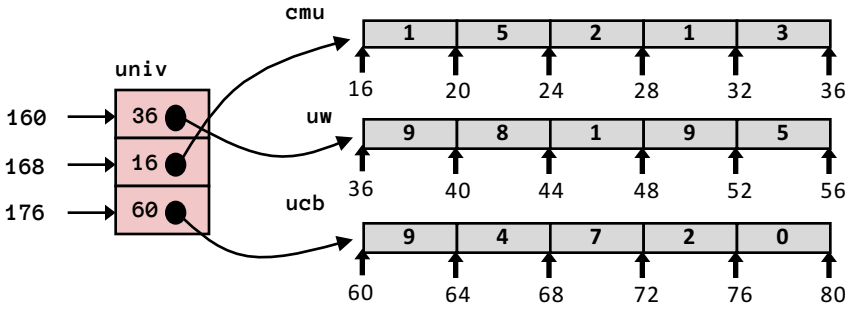
Nested array

```
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];
}
```

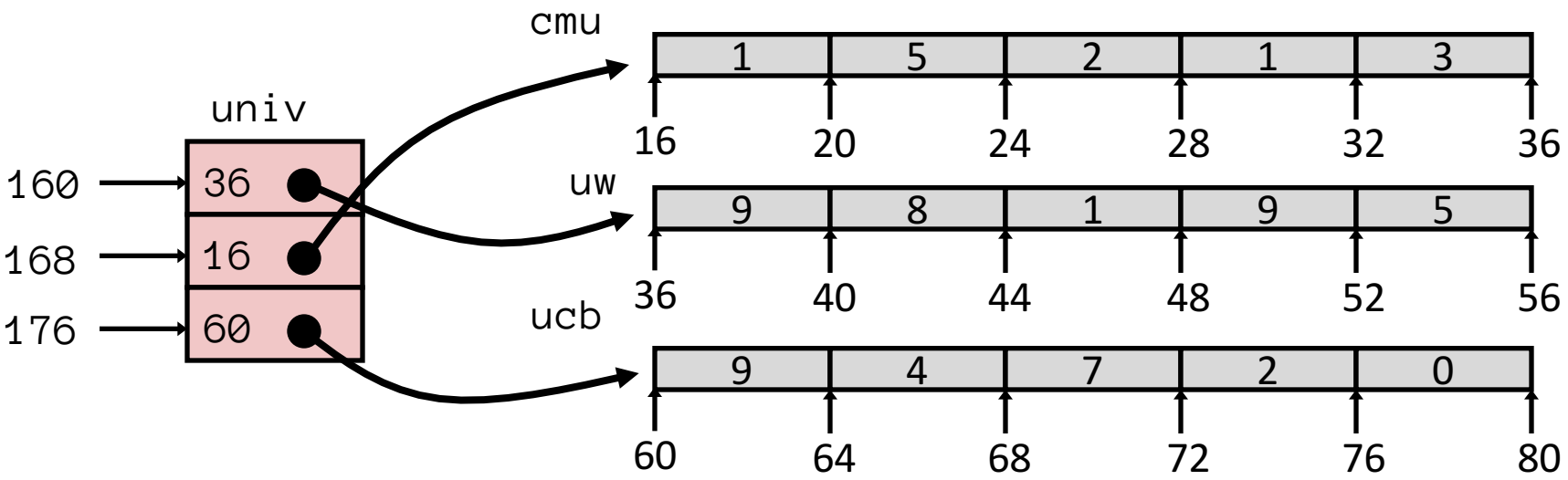


Access *looks* the same, but it isn't:

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

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

Strange Referencing Examples



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

  `univ[2][3]`

  `univ[1][5]`

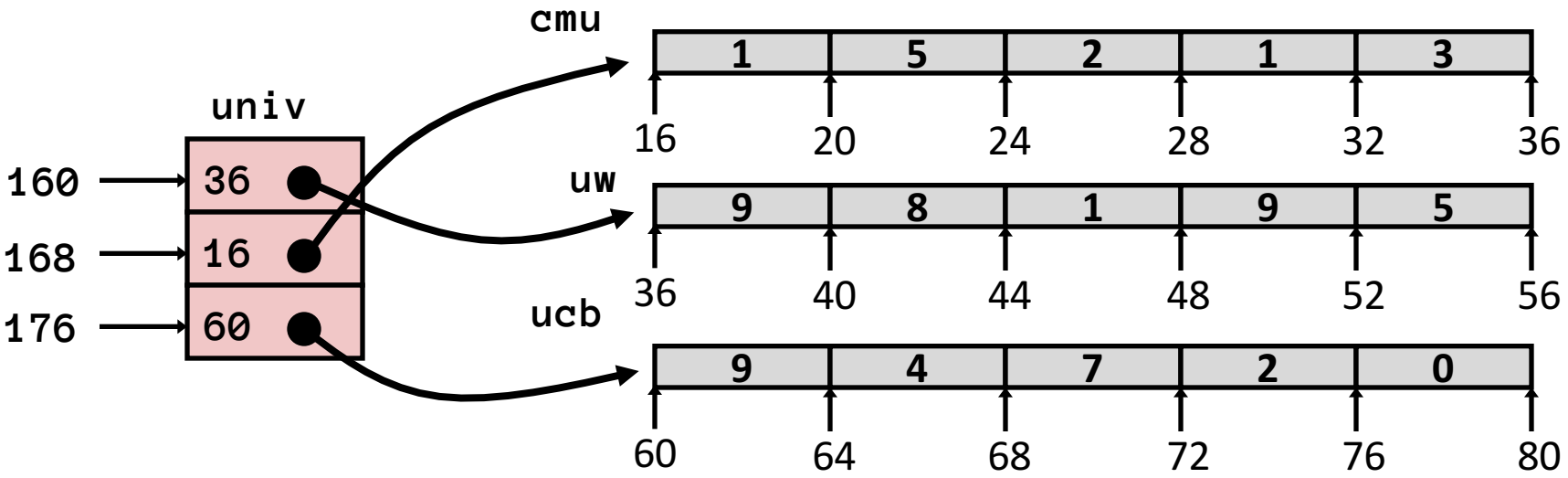
  `univ[2][-2]`

  `univ[3][-1]`

  `univ[1][12]`

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

Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`	`60+4*3 = 72`	2	Yes
`univ[1][5]`	`16+4*5 = 36`	9	No
`univ[2][-2]`	`60+4*-2 = 52`	5	No
`univ[3][-1]`	`#@%!^??`	??	No
`univ[1][12]`	`16+4*12 = 64`	4	No

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

Summary: Arrays in C

- **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