

Executables & Arrays

CSE 351 Autumn 2024

Instructor:

Ruth Anderson

Teaching Assistants:

Alexandra Michael

Connie Chen

Chloe Fong

Chendur Jayavelu

Joshua Tan

Nikolas McNamee

Nahush Shrivatsa

Naama Amiel

Neela Kausik

Renee Ruan

Rubee Zhao

Samantha Dreussi

Sean Siddens

Waleed Yagoub



<http://xkcd.com/1270/>

Relevant Course Information

- ❖ HW11 due TONIGHT, Wednesday (10/23) @ 11:59 pm
- ❖ Lab 2 (x86-64) due this Friday (10/25)
 - Since you are submitting a text file (`defuser.txt`), there won't be any Gradescope autograder output this time
- ❖ HW12 due Friday (10/25) @ 11:59 pm
- ❖ **Midterm Exam:** take home, on Gradescope, more details coming soon: <https://cs.uw.edu/cse351/exams/>
 - Released: Thursday 10/31 at 5pm
 - Due: Saturday 11/02 at 11:59pm
 - Study BEFORE the exam opens!
 - Review in section next week (10/31)

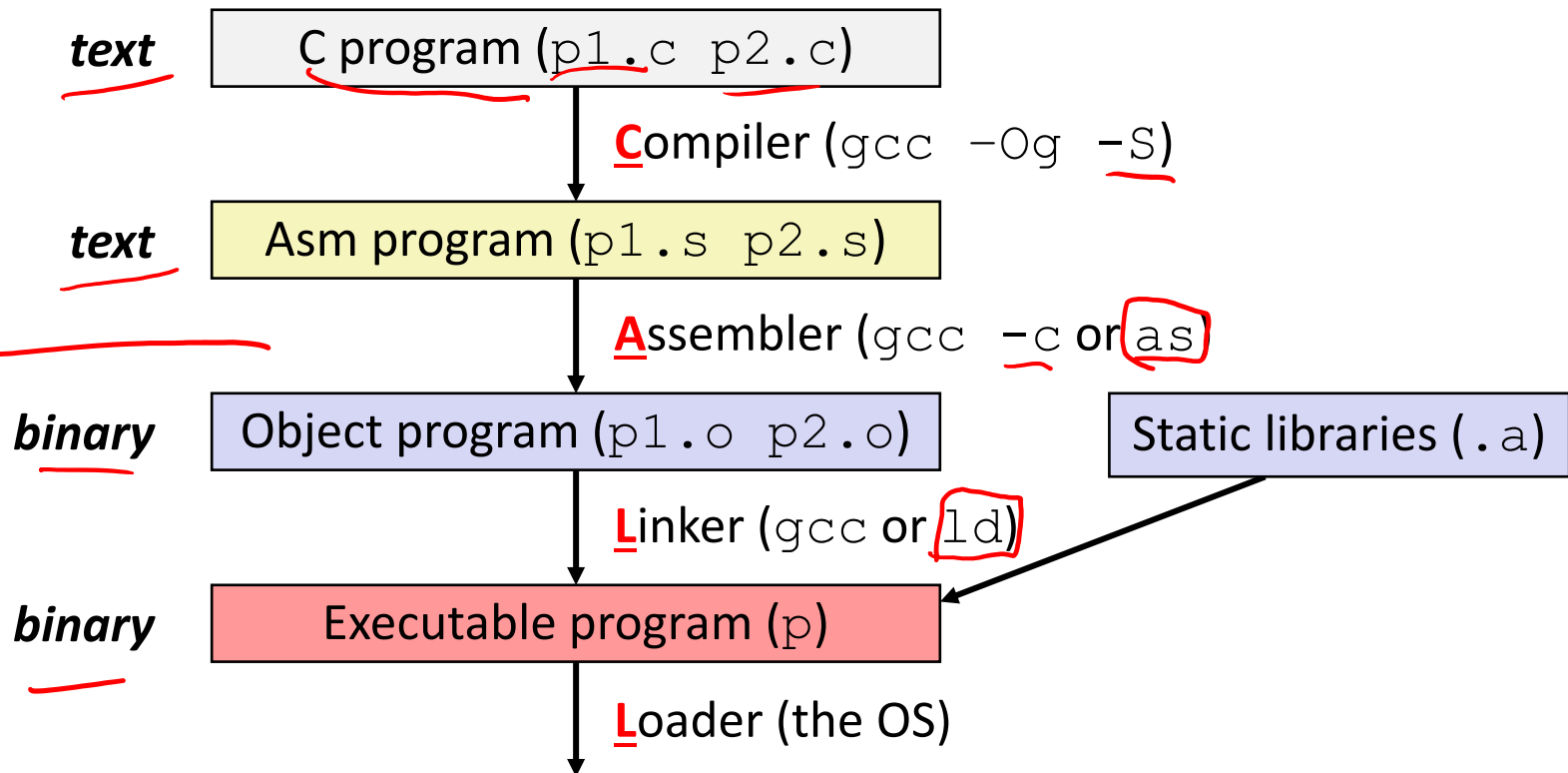
Reading Review

❖ Terminology:

- CALL: compiler, assembler, linker, loader
- Object file: symbol table, relocation table
- Disassembly
- Multidimensional arrays, row-major ordering
- Multilevel arrays

Building an Executable with C (Review)

- ❖ Code in files `p1.c p2.c`
 - ❖ Compile with command: `gcc -Og p1.c p2.c -o p`
 - Put resulting machine code in file `p`
 - ❖ Run with command: `./p`
- Handwritten notes:*
- Red arrow from `p1.c p2.c` to `gcc` with text: "can compile multiple source files into a single executable"
- Red circles around `p1.c p2.c` and `-o p`
- Red arrow from `-o p` to `./p`



Polling Question

❖ Use the following disassembly:

```

000000000000401126 <main>:
 401126:  48 83 ec 08          sub    $0x8,%rsp
 40112a:  bf 10 20 40 00      mov    $0x402010,%edi
 40112f:  e8 fc fe ff ff      callq 401030 <puts@plt>
 401134:  b8 00 00 00 00      mov    $0x0,%eax
 401139:  48 83 c4 08          add    $0x8,%rsp
 40113d:  c3                  retq
 40113e:  66 90              xchg   %ax,%ax

```

- What is the byte of data at address **0x40113b**? 0xc4
- The immediate \$0x402010 can be found in the machine code! **What is its address?** 0x40112b

Compiler (Review)

- ❖ **Input:** Higher-level language code (*e.g.* C, Java)
 - `foo.c`
 - ❖ **Output:** Assembly language code (*e.g.* x86, ARM, MIPS)
 - `foo.s`
-
- ❖ First there's a preprocessor step to handle #directives
 - Macro substitution, plus other specialty directives
 - If curious/interested: <http://tigrcc.ticalc.org/doc/cpp.html>
 - ❖ Super complex, whole courses devoted to these!
 - ❖ Compiler optimizations
 - "Level" of optimization specified by capital 'O' flag (*e.g.* `-Og`, `-O3`)
 - Options: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

gcc -E

#define
#include

Compiling Into Assembly (Review)

❖ C Code (sum.c)

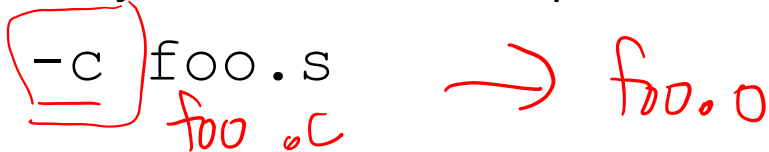
```
void sumstore(long x, long y, long *dest) {  
    long t = x + y;  
    *dest = t;  
}
```

❖ x86-64 assembly (gcc -Og -S sum.c)

```
sumstore(long, long, long*):  
    addq    %rdi, %rsi  
    movq    %rsi, (%rdx)  
    ret
```

Warning: You may get different results with other versions of gcc and different compiler settings

Assembler (Review)

- ❖ **Input:** Assembly language code (*e.g.* x86, ARM, MIPS)
 - `foo.s`
 - ❖ **Output:** Object files (*e.g.* ELF, COFF)
 - `foo.o`
 - Contains *object code* and *information tables*
-
- ❖ Reads and uses *assembly directives*
 - *e.g.* `.text`, `.data`, `.quad`
 - x86: https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html
 - ❖ Produces “machine language”
 - Does its best, but object file is ~~not~~ a completed binary
 - ❖ Example: `gcc -c foo.s` 

Producing Machine Language (Review)

- ❖ **Simple cases:** arithmetic and logical operations, shifts, etc.
 - All necessary information is contained in the instruction itself
- ❖ What about the following?
 - Conditional/unconditional jump *label*
 - Accessing static data (e.g. global var or jump table)
 - call *foo*
- ❖ **Addresses and labels are problematic because the final executable hasn't been constructed yet!**
 - So how do we deal with these in the meantime?

Object File Information Tables (Review)

- ❖ Each object file has its own symbol and relocation tables
- ❖ **Symbol Table** holds list of “items” that may be used by other files “what I have”
 - *Non-local labels* – function names for `call`
 - *Static Data* – variables & literals that might be accessed across files
- ❖ **Relocation Table** holds list of “items” that this file needs the address of later (currently undetermined) “What I need”
 - Any *label* or piece of *static data* referenced in an instruction in this file
 - Both internal and external

In .o
file

Object File Format

- 1) object file header: size and position of the other pieces of the object file "table of contents"
- 2) text segment: the machine code (Instructions)
- 3) data segment: data in the source file (binary) (Static Data & Literals)
- 4) relocation table: identifies lines of code that need to be "handled"
- 5) symbol table: list of this file's labels and data that can be referenced
- 6) debugging information (info for GDB)

❖ More info: ELF format

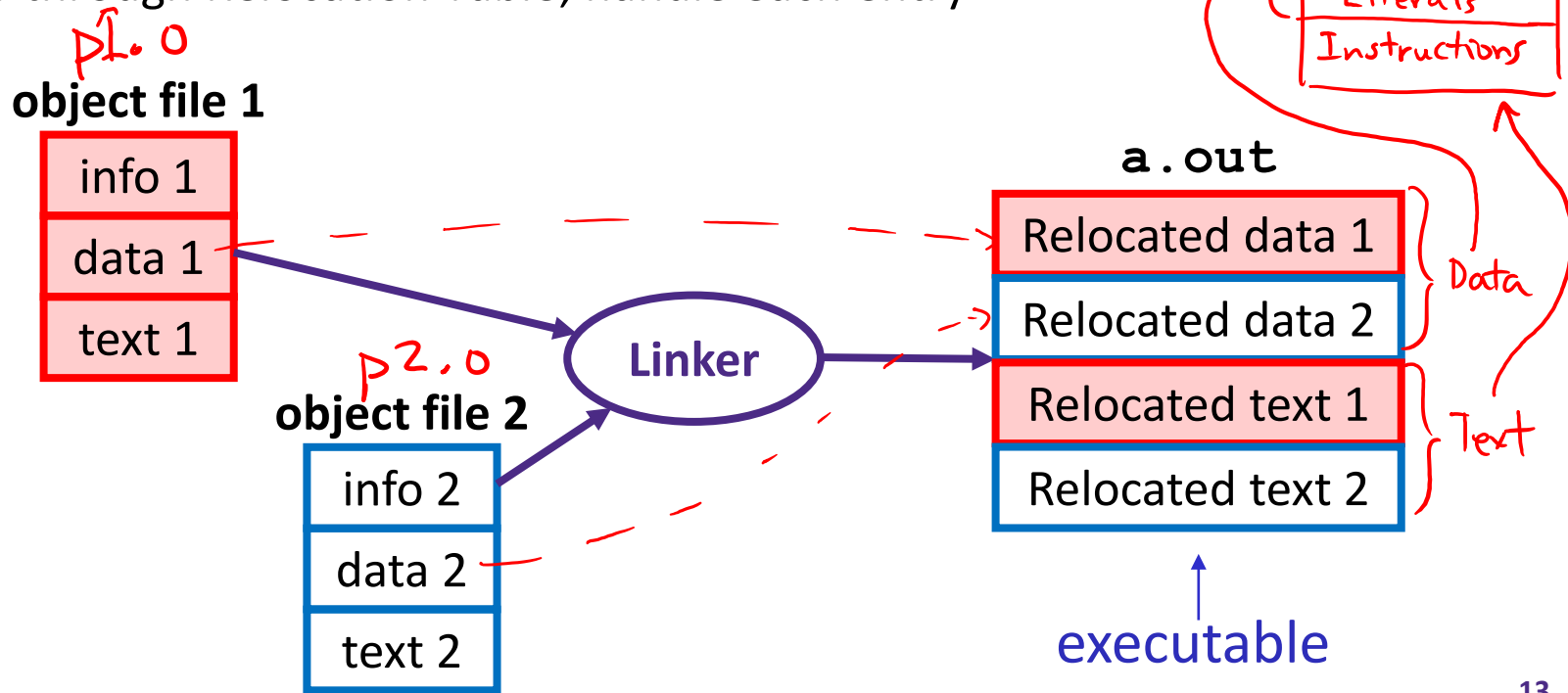
- http://www.skyfree.org/linux/references/ELF_Format.pdf

Linker (Review)

- ❖ **Input:** Object files (e.g. ELF, COFF)
 - `foo.o`
- ❖ **Output:** executable binary program
 - `a.out` ← default name for executable
- ❖ Combines several object files into a single executable (*linking*)
- ❖ Enables separate compilation/assembling of files
 - Changes to one file do not require recompiling of whole program

Linking (Review)

- 1) Take text segment from each `.o` file and put them together
- 2) Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments
- 3) Resolve References
 - Go through Relocation Table; handle each entry



Disassembling Object Code (Review)

❖ Disassembled:

```

000000000000400536 <sumstore>:
  400536:  48 01 fe      add    %rdi,%rsi
  400539:  48 89 32      mov    %rsi, (%rdx)
  40053c:  c3           retq

```

address of instruction

object code bytes (hex)

interpreted assembly instructions

❖ Disassembler (objdump -d sum)

- Useful tool for examining object code (man 1 objdump)
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can run on either a complete executable or object file

What Can be Disassembled?

```
% objdump -d WINWORD.EXE
```

```
WINWORD.EXE: file format pei-i386
```

```
No symbols in "WINWORD.EXE".
```

```
Disassembly of section .text:
```

```
30001000 <.text>:
```

```
30001000:
```

```
30001001:
```

```
30001003:
```

```
30001005:
```

```
3000100a:
```

**Reverse engineering forbidden by
Microsoft End User License Agreement**

- ❖ Anything that can be interpreted as executable code
- ❖ Disassembler examines bytes and attempts to reconstruct assembly source

Loader (Review)

- ❖ **Input:** executable binary program, command-line arguments
 - `./a.out` arg1 arg2
- ❖ **Output:** <program is run>
- ❖ Loader duties primarily handled by OS/kernel
 - More about this when we learn about processes
- ❖ Memory sections (Instructions, Static Data, Stack) are set up
- ❖ Registers are initialized

Building an Executable Summary

- ❖ Multistep process:
 - compiling, assembling, linking
- ❖ Object code finished by linker
 - using symbol and relocation tables to produce machine code (with finalized addresses)
- ❖ Loader sets up initial memory from executable

Data Structures in C

❖ Arrays

- One-dimensional

- 
- Multidimensional (nested)
 - Multilevel

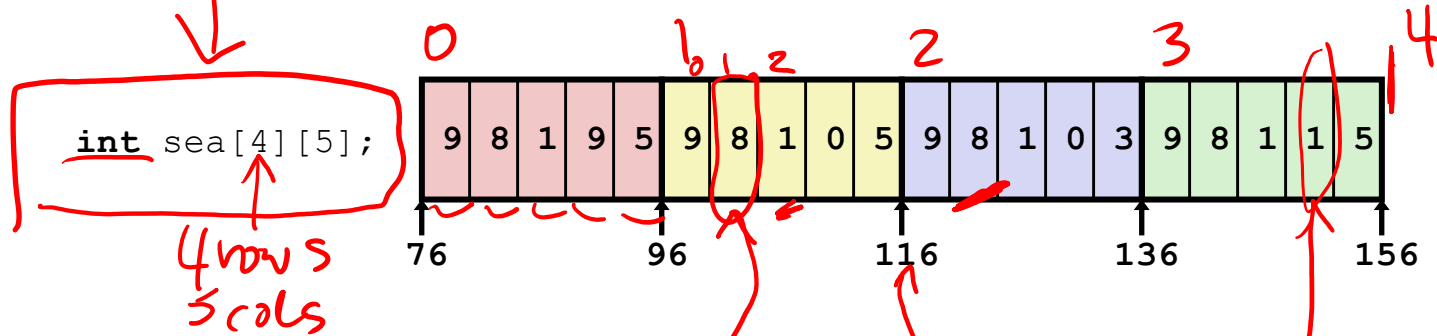
❖ Structs

- Alignment

Polling Question

Multi-dimensional (Nested)
row-major layout

❖ Which of the following statements is FALSE?



A. sea[4][-2] is a *valid* array reference

False B. sea[1][1] makes *two* memory accesses

C. sea[2][1] will *always* be a higher address than sea[1][2]

D. sea[2] is calculated using *only* 1 ea

Review: Array Allocation (Review)

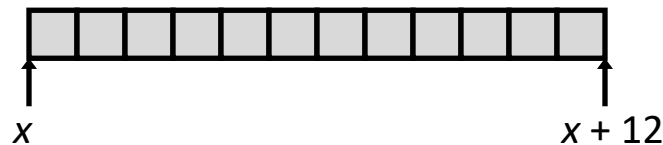
❖ Basic Principle

- $\mathbf{T} \ A[N]; \rightarrow$ array of data type $\mathbf{T}$ and length N

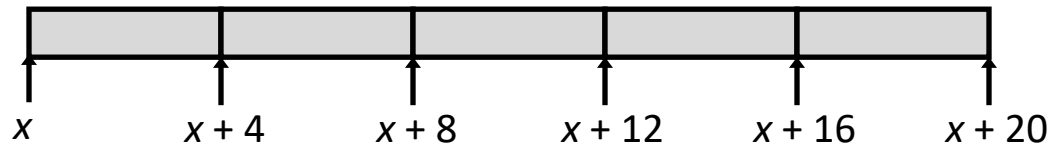
★ *Contiguously* allocated region of $N * \text{sizeof}(\mathbf{T})$ bytes

- Identifier A returns address of array (type $\mathbf{T}^*$)

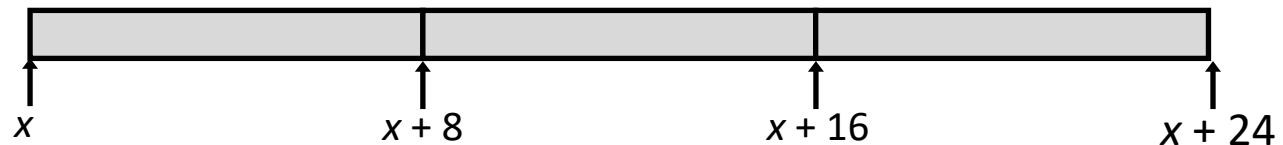
`char msg[12];`



`int val[5];`



`double a[3];`



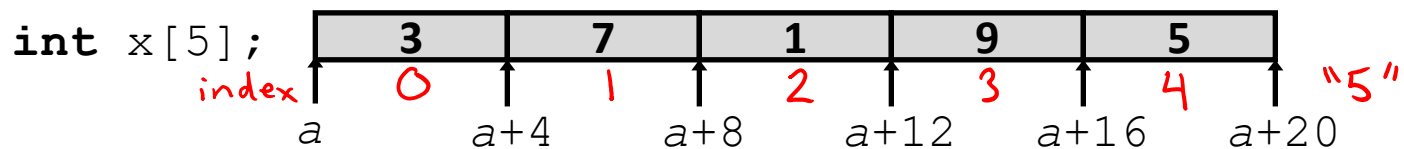
`char* p[3];`
(or `char *p[3];`)



Review: Array Access

❖ Basic Principle

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



❖ Reference Type Value

<code>x[4]</code>	<code>int</code>	5
<code>x</code>	<code>int*</code>	<code>a</code>
<code>x+1</code> ← ptr arithmetic	<code>int*</code>	<code>a + 4</code>
<code>&x[2]</code>	<code>int*</code>	<code>a + 8</code>
<code>x[5]</code>	<code>int</code>	?? (whatever's in memory at addr <code>x+20</code>)
<code>*(x+1)</code>	<code>int</code>	7
<code>x+i</code>	<code>int*</code>	<code>a + 4*i</code>

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 };
```

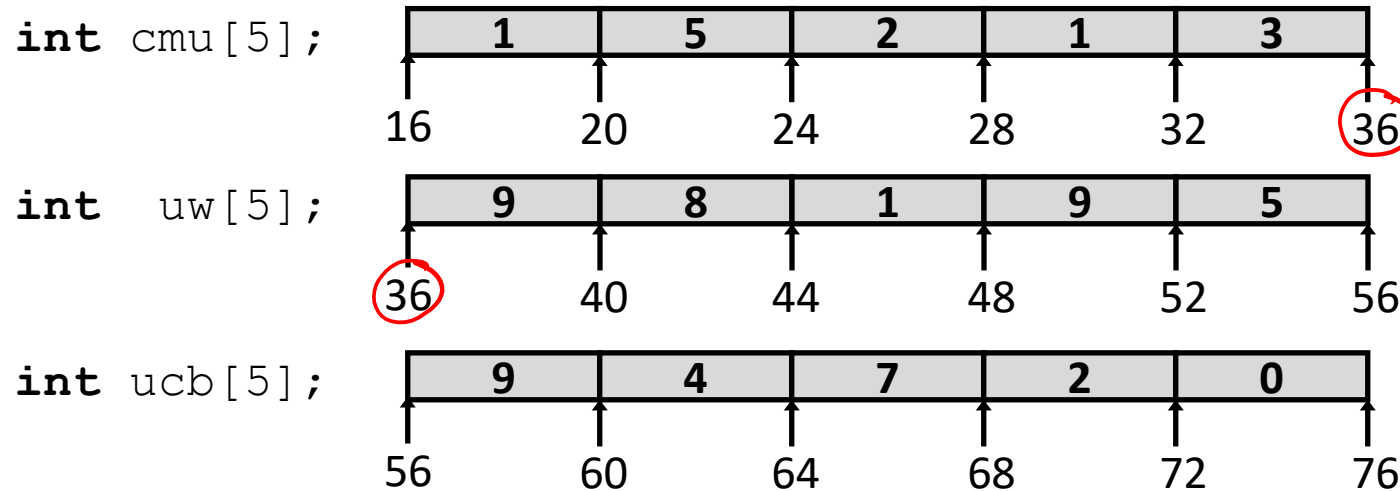
brace-enclosed
list initialization



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 };
```

20 B each

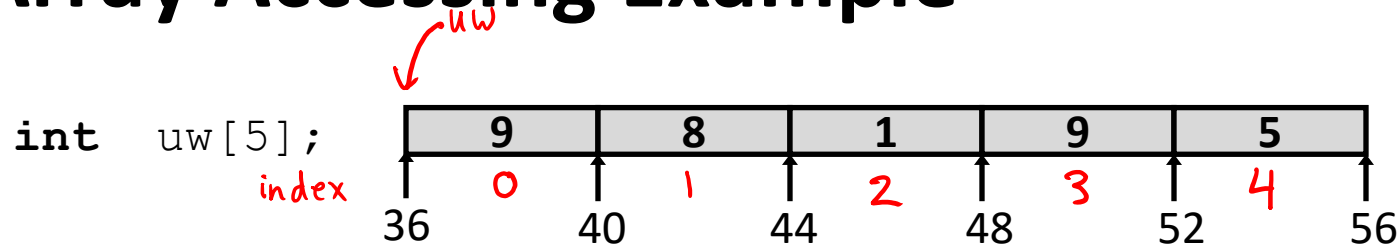


no gap in this example

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

(could have allocated variables in-between)

Array Accessing Example



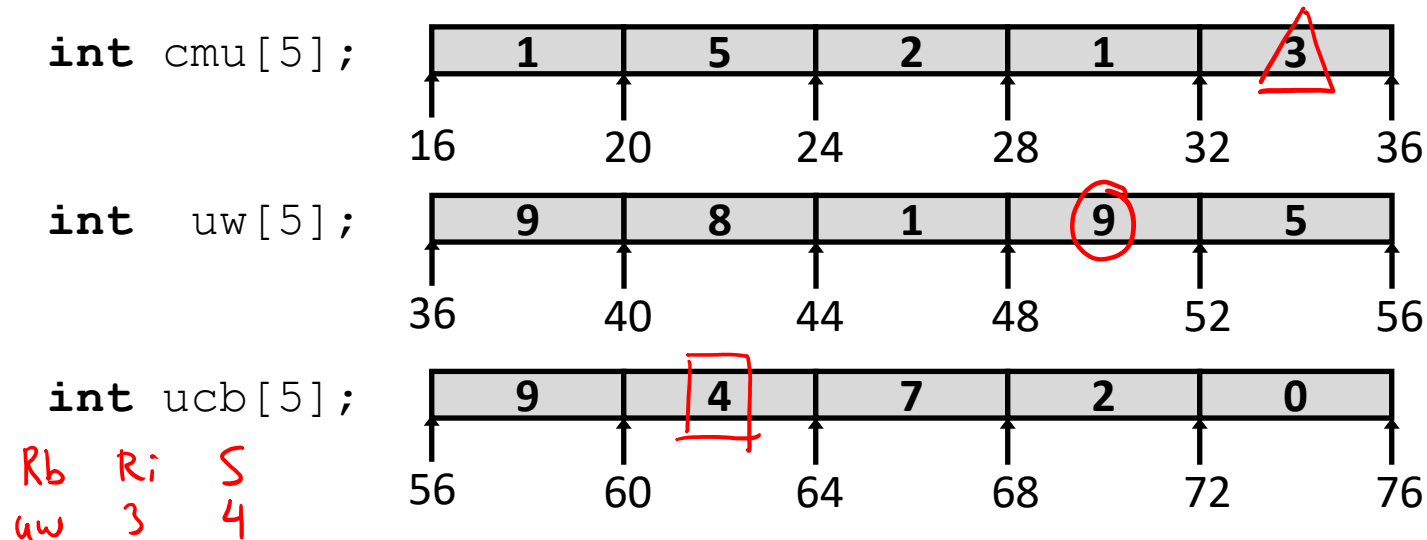
```
// return specified digit of ZIP code
int get_digit(int z[5], int digit) {
    return z[digit];
}
```

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

Annotations:
 - `%rdi` is the base register (R_b)
 - `%rsi` is the index register (R_i)
 - `4` is the scale factor (S), which is `sizeof(int)`

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

Bonus: Referencing Examples



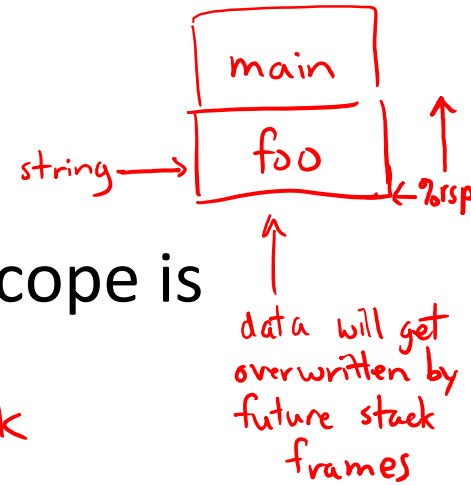
<u>Reference</u>	<u>Address</u>	<u>Value</u>	<u>Guaranteed?</u>
uw[3]	$36 + 3 * 4 = 48$	9	Yes
uw[6]	$36 + 6 * 4 = 60$	4	No
uw[-1]	$36 + (-1) * 4 = 32$	3	No
cmu[15]	$16 + 15 * 4 = 76$	?	No

- ❖ 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: `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 (0th) 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;  
}
```

array is allocated on stack

BAD!

returns stack addr that is < %rsp

- ❖ 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` (%rdi can only fit 8 bytes)

Must explicitly
pass the size!

Data Structures in Assembly

❖ Arrays

- One-dimensional
- **Multidimensional (nested)**
- Multilevel

❖ Structs

- Alignment

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 }};
```

2D array

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

❖ What is the layout in memory?

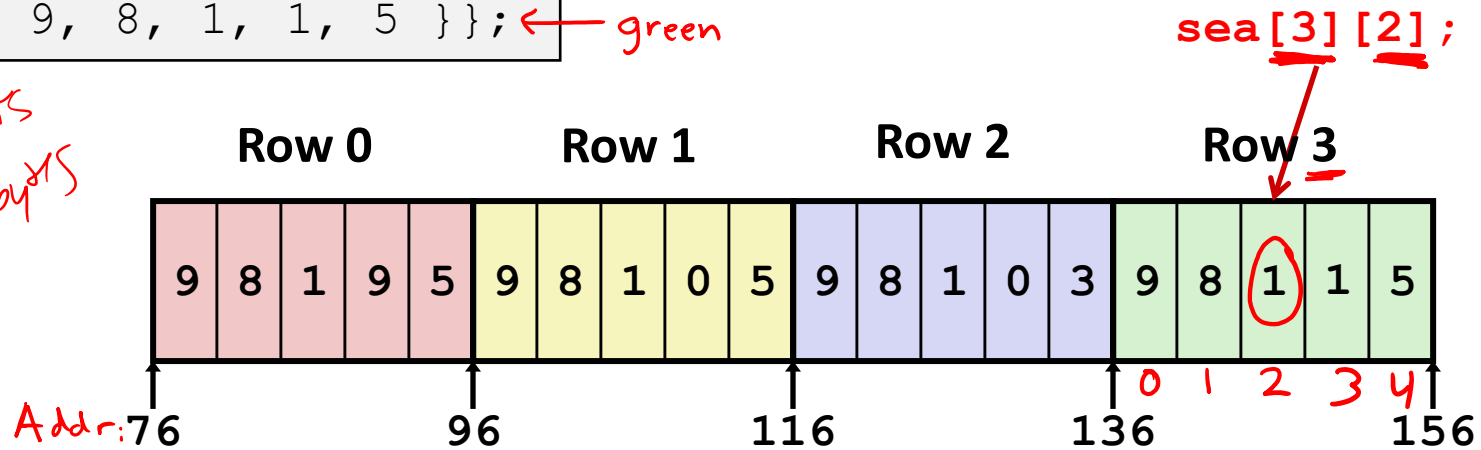
Nested Array Example

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

← red
← yellow
← blue
← green

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

20 ints
80 bytes



- ❖ “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

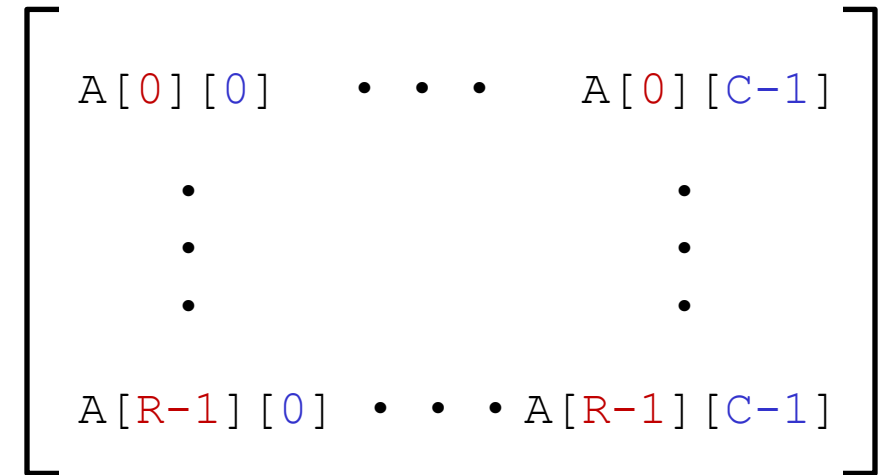
❖ Array size?

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

Two-Dimensional (Nested) Arrays

❖ Declaration: **T** $A[R][C]$;

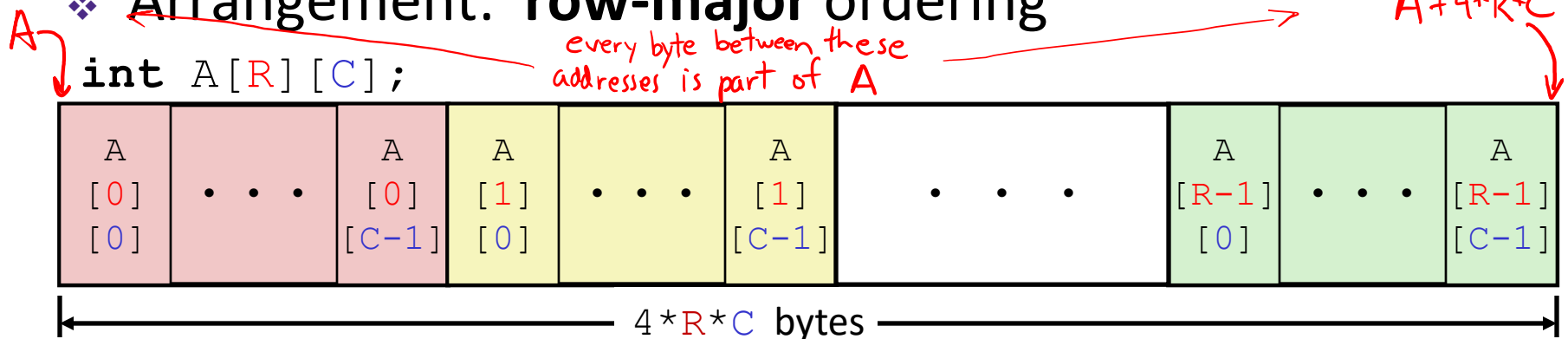
- 2D array of data type **T**
- R rows, C columns
- Each element requires **sizeof(T)** bytes



❖ Array size:

- $R * C * \text{sizeof}(\mathbf{T})$ bytes

❖ Arrangement: **row-major** ordering



Nested Array Row Access

❖ Row vectors

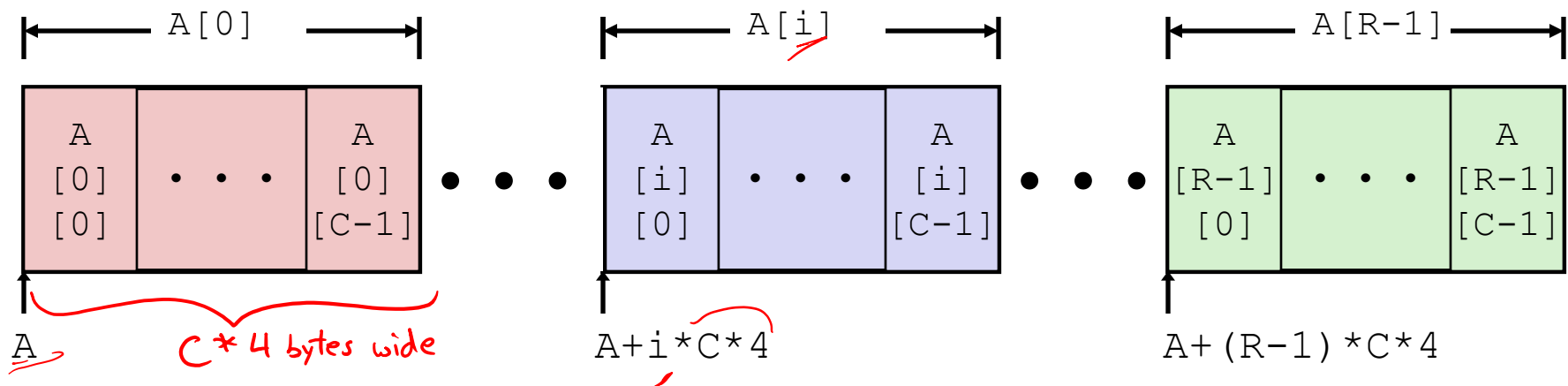
■ Given **T** $A[R][C]$,

- $A[i]$ is an array of C elements ("row i ") → just an address!
- A is address of array
- Starting address of row $i = A + i * (C * \text{sizeof}(\mathbf{T}))$

size of row

0, 1

`int A[R][C];`



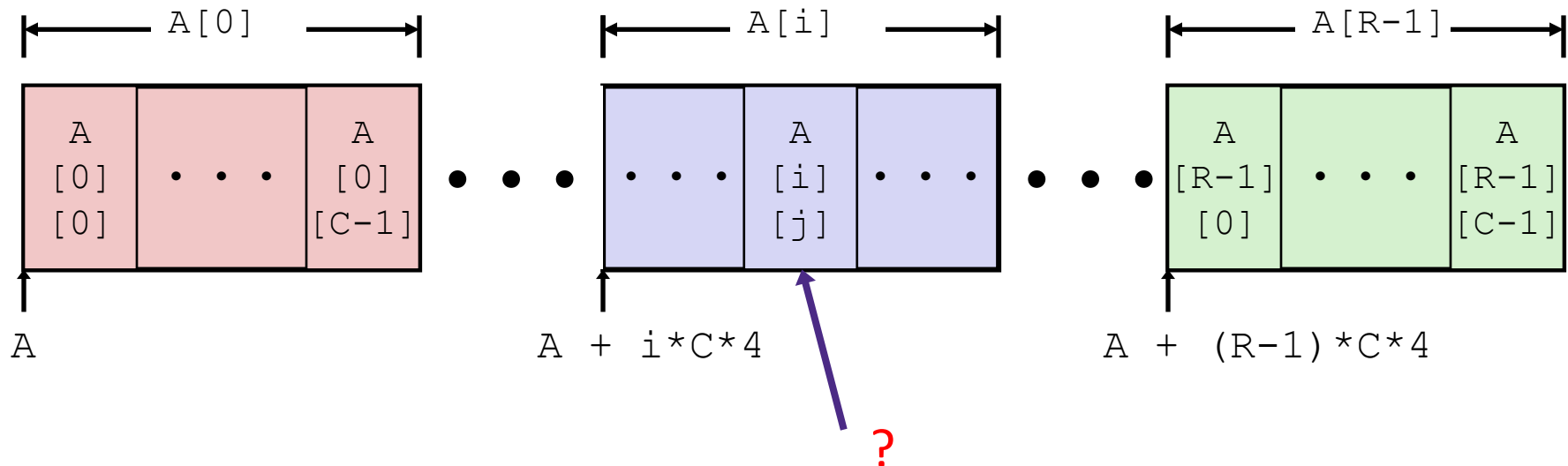
Nested Array Element Access

reminder: $ar[j] = *(ar + j)$

❖ Array Elements

- $A[i][j]$ is element of type T , which requires K bytes
- Address of $(A[i])[j]$ is $(A + i * C * \text{sizeof}(T)) + j * \text{sizeof}(T)$
address

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



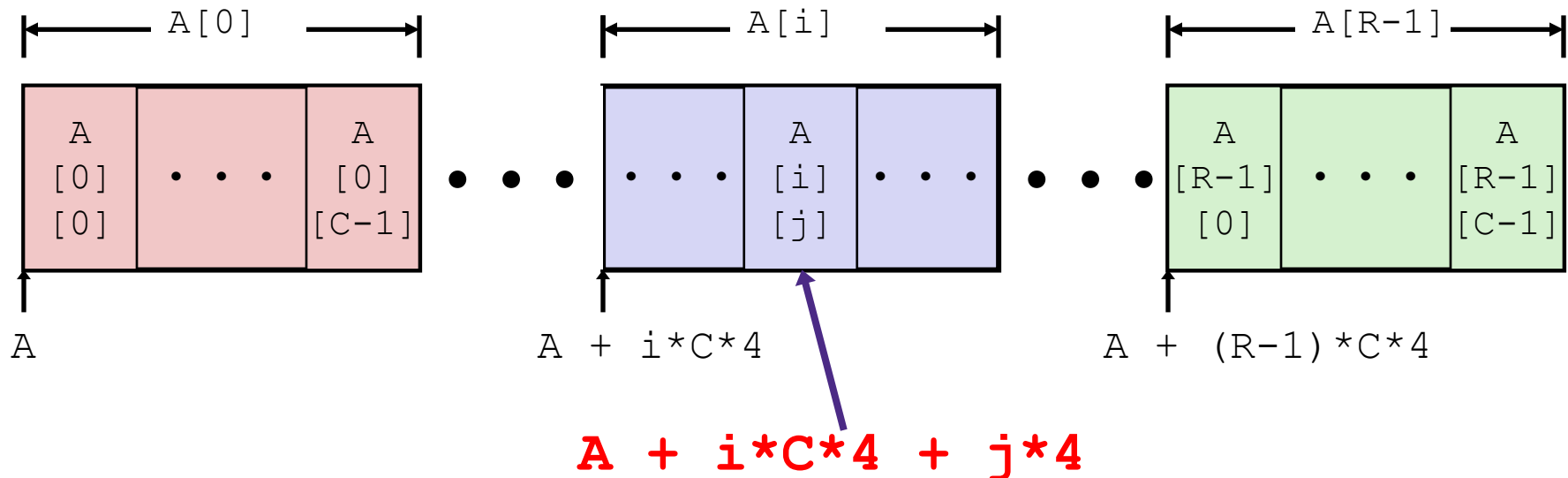
Nested Array Element Access

❖ Array Elements

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

$$A + \underbrace{i * (C * K)}_{\text{Size of row}} + \underline{j * K} == A + (i * C + j) * K$$

```
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))
```

mov gets data

❖ Array Elements

- sea[index][digit] is an **int** (**sizeof(int)**=4)

- Address = sea + 5*4*index + 4*digit

start of array

start of row

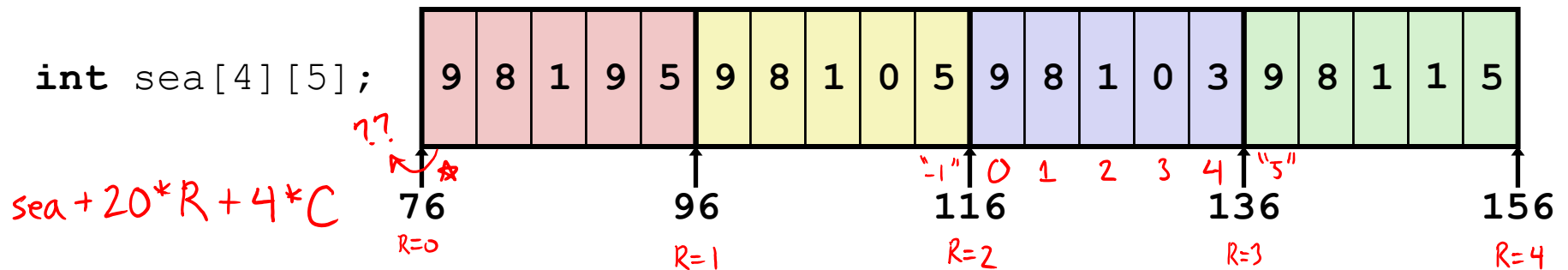
column of interest

❖ Assembly Code

- Computes address as: sea + ((index+4*index) + digit)*4

- movl performs memory reference

Bonus: Multidimensional Referencing Examples



<u>Reference</u>	<u>Address</u>	<u>Value</u>	<u>Guaranteed?</u>
<code>sea[3][3]</code>	$76 + 20 \cdot 3 + 4 \cdot 3 = 148$	1	Yes
<code>sea[2][5]</code>	$76 + 20 \cdot 2 + 4 \cdot 5 = 136$	9	Yes
<code>sea[2][-1]</code>	$76 + 20 \cdot 2 + 4 \cdot (-1) = 112$	5	Yes
<code>sea[4][-1]</code>	$76 + 20 \cdot 4 + 4 \cdot (-1) = 152$	5	Yes
<code>sea[0][19]</code>	$76 + 20 \cdot 0 + 4 \cdot (19) = 152$	5	Yes
<code>sea[0][-1]</code>	$76 + 20 \cdot 0 + 4 \cdot (-1) = 72$	??	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

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 };
```

} could be
apart

4 arrays

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

60B

24B

$univ[2][2] == 7$

2D Array Declaration:

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

60B

} guaranteed
contiguous

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

NO

$univ2D[2][2] == 7$

1 array

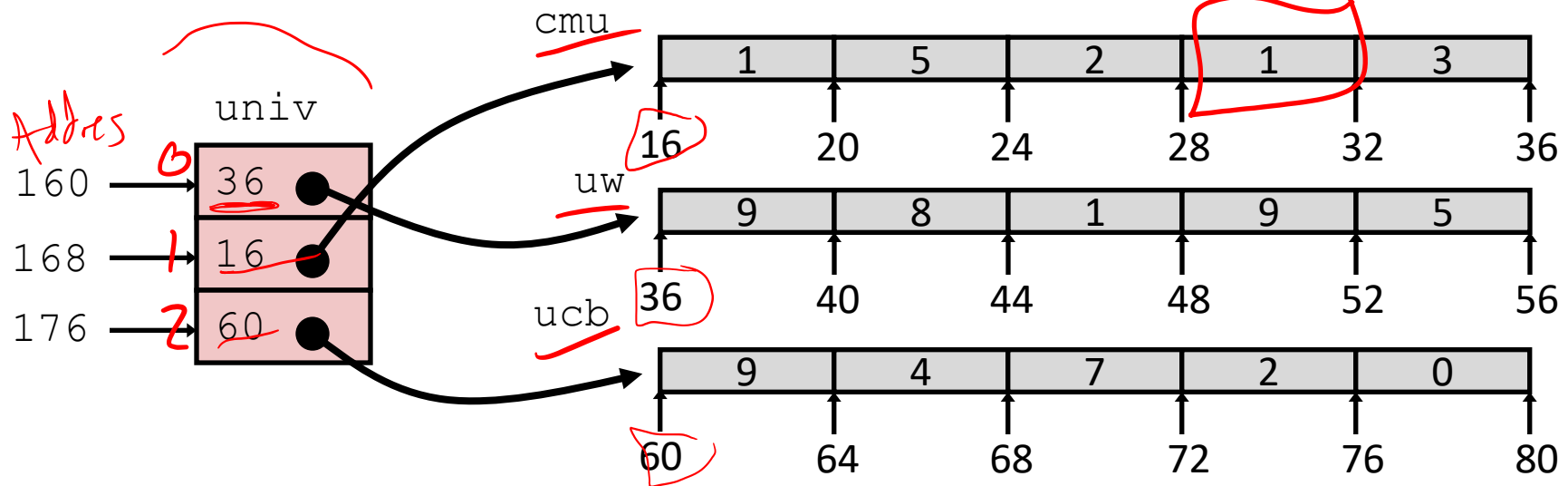
One array declaration = one contiguous block of memory

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};
```

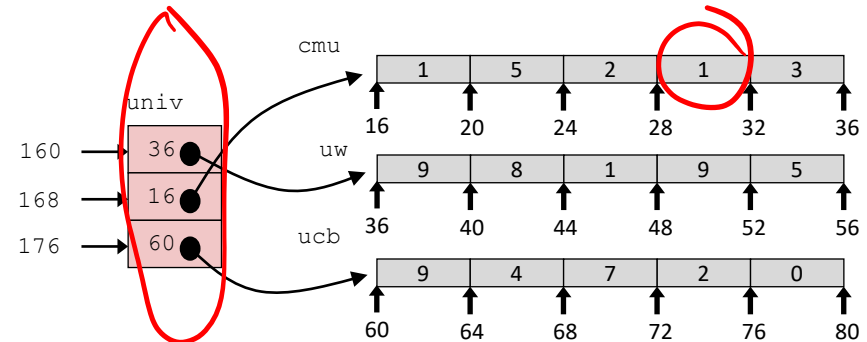
- ❖ Variable `univ` denotes array of 3 elements
- ❖ Each element is a pointer
 - 8 bytes each
- ❖ Each pointer points to array of `ints`



Note: this is how Java represents multidimensional arrays

Element Access in Multilevel 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
```

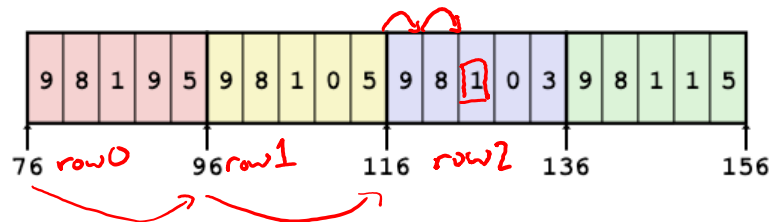
❖ Computation

- 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

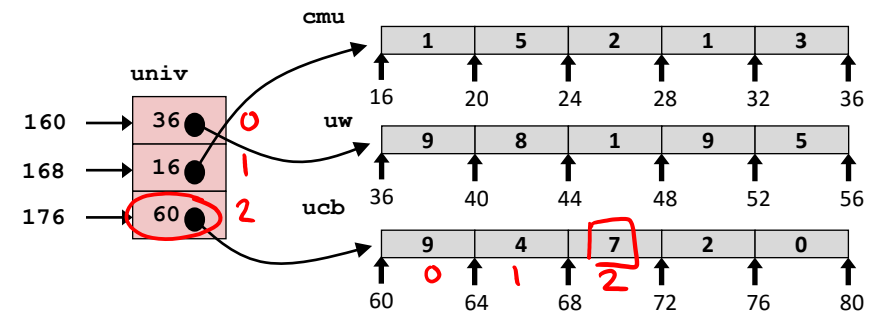
Multidimensional array

```
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];
}
```



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

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

more efficient:

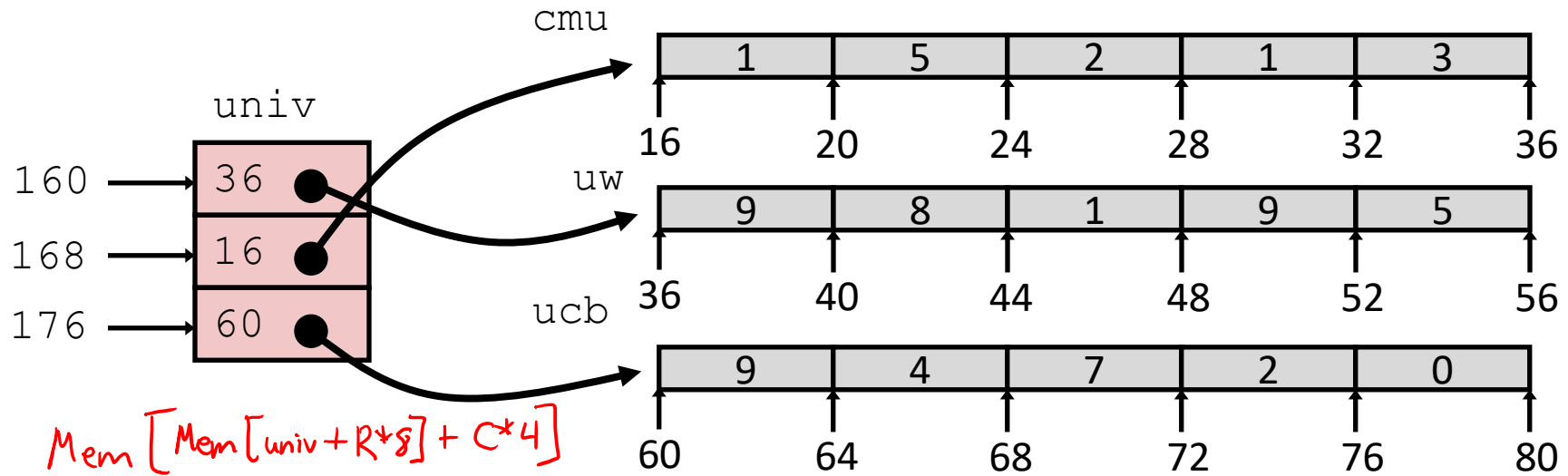
- less overall memory
- faster to access

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

more flexible:

- easier to "fit" smaller arrays in memory
- can swap out rows (and resize)
- can have rows of different lengths

Bonus: Multilevel Referencing Examples



Reference	Address	Value	Guaranteed?
<code>univ[2][3]</code>	$\text{Mem}[176] + 3 * 4 = 60 + 12 = 72$	2	Yes
<code>univ[1][5]</code>	$\text{Mem}[168] + 5 * 4 = 16 + 20 = 36$	9	No
<code>univ[2][-2]</code>	$\text{Mem}[176] + (-2) * 4 = 60 - 8 = 52$	5	No
<code>univ[3][-1]</code>	$\text{Mem}[184] + (-1) * 4 = ?? - 4 = ??$	???	No
<code>univ[1][12]</code>	$\text{Mem}[168] + 12 * 4 = 16 + 48 = 64$	4	No

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

Arrays Summary

- ❖ Contiguous allocations of memory
- ❖ **No bounds checking** (and no default initialization)
- ❖ Can usually be treated like a pointer to first element
- ❖ Multidimensional
 - `int a[4][5];` → array of arrays
 - all levels in one contiguous block of memory
- ❖ Multilevel
 - `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