

Executables & Arrays

CSE 351 Winter 2020

Instructor:

Ruth Anderson

Teaching Assistants:

Jonathan Chen

Justin Johnson

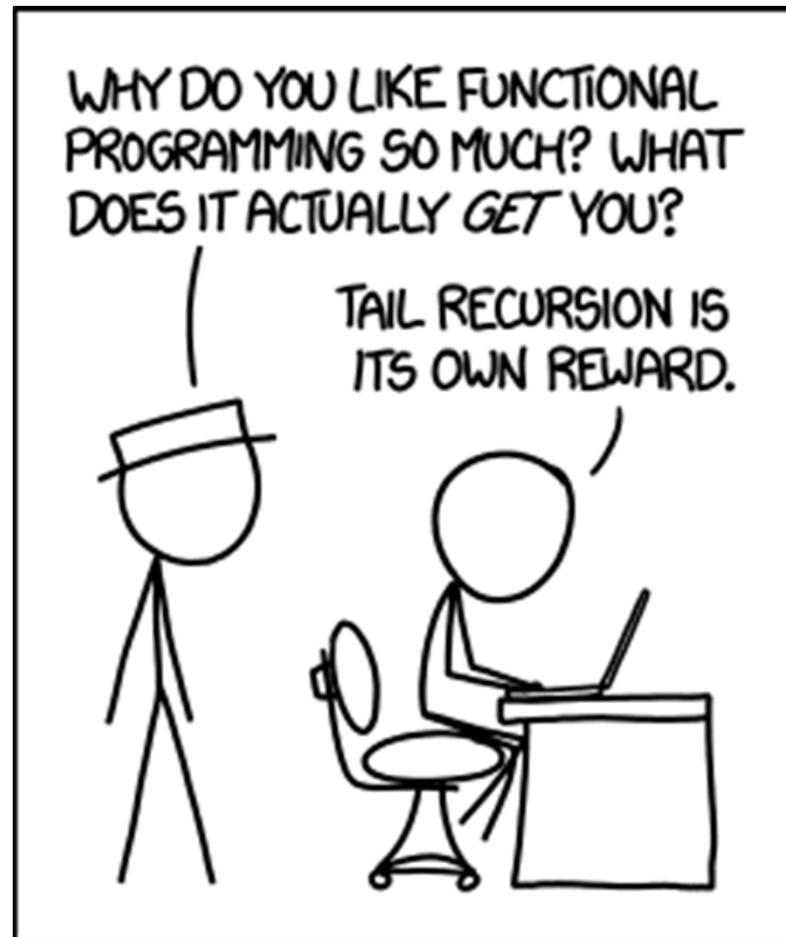
Porter Jones

Josie Lee

Jeffery Tian

Callum Walker

Eddy (Tianyi) Zhou



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

Administrivia

- ❖ Lab 2 & hw12 due Friday (2/07)
- ❖ **Midterm** Monday (2/10), during lecture
- ❖ Mid-quarter survey due Thursday (2/13)
- ❖ hw13 due *next* Friday (2/14)
 - Based on the next two lectures, longer than normal

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();
```

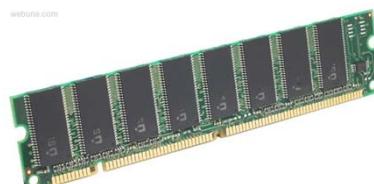
Assembly language:

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

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:



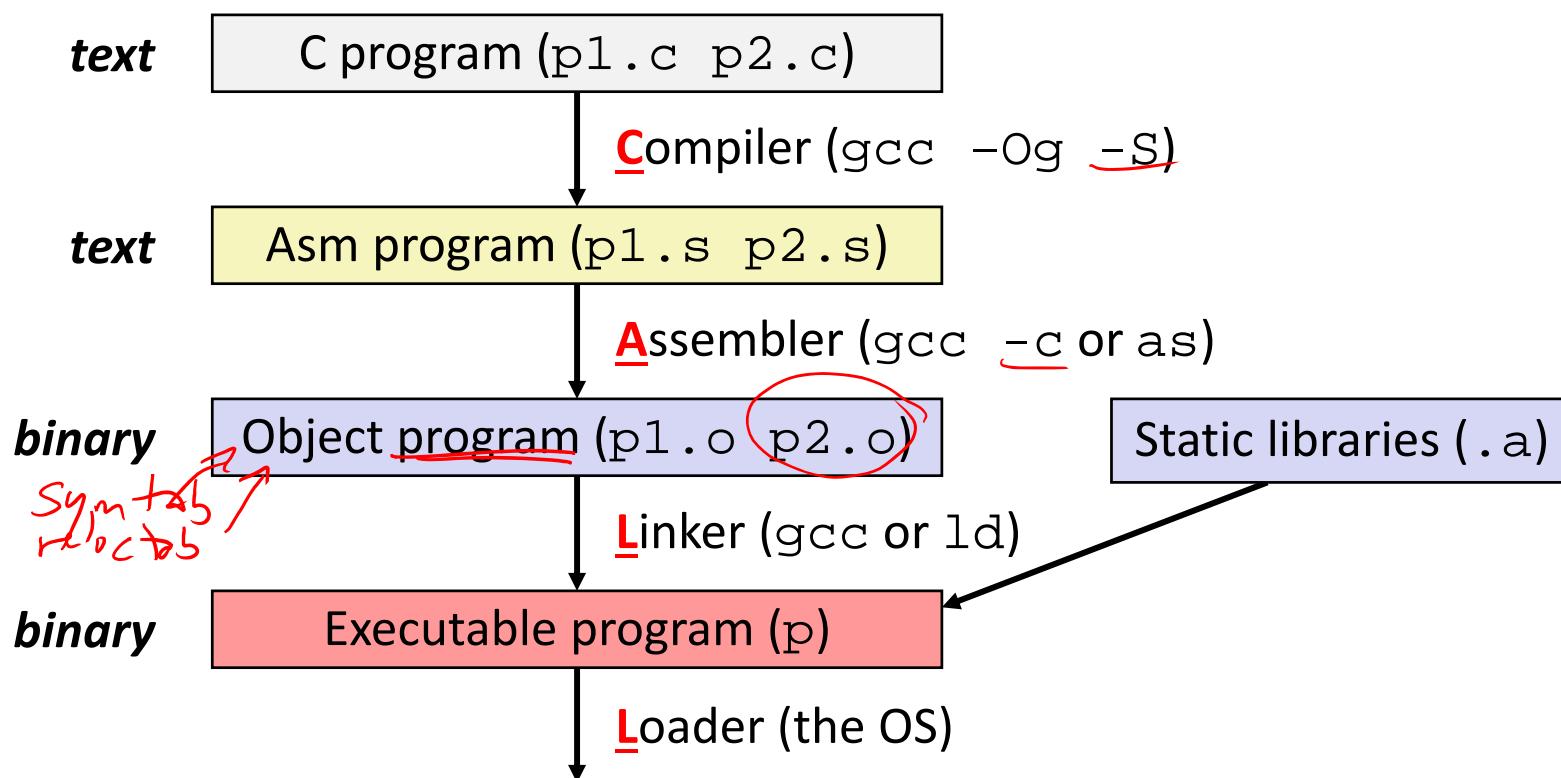
Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

OS:



Building an Executable from a C File

- ❖ Code in files `p1.c p2.c` can compile multiple source files into a single executable
- ❖ Compile with command: `gcc -Og p1.c p2.c -o p`
 - Put resulting machine code in file `p`
- ❖ Run with command: `./p`



Compiler

- ❖ **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://tigcc.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

Compiling Into Assembly

- ❖ 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
```

Sum.S

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

Assembler

- ❖ **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

addq %rax, %rbx

- ❖ **Simple cases:** arithmetic and logical operations, shifts, etc.
 - All necessary information is contained in the instruction itself
- ❖ What about the following?
 - Conditional jump
 - Accessing static data (e.g. global var or jump table)
 - call ~~to~~
- ❖ 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

- ❖ **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
- ❖ Each file has its own symbol and relocation tables

Object File Format

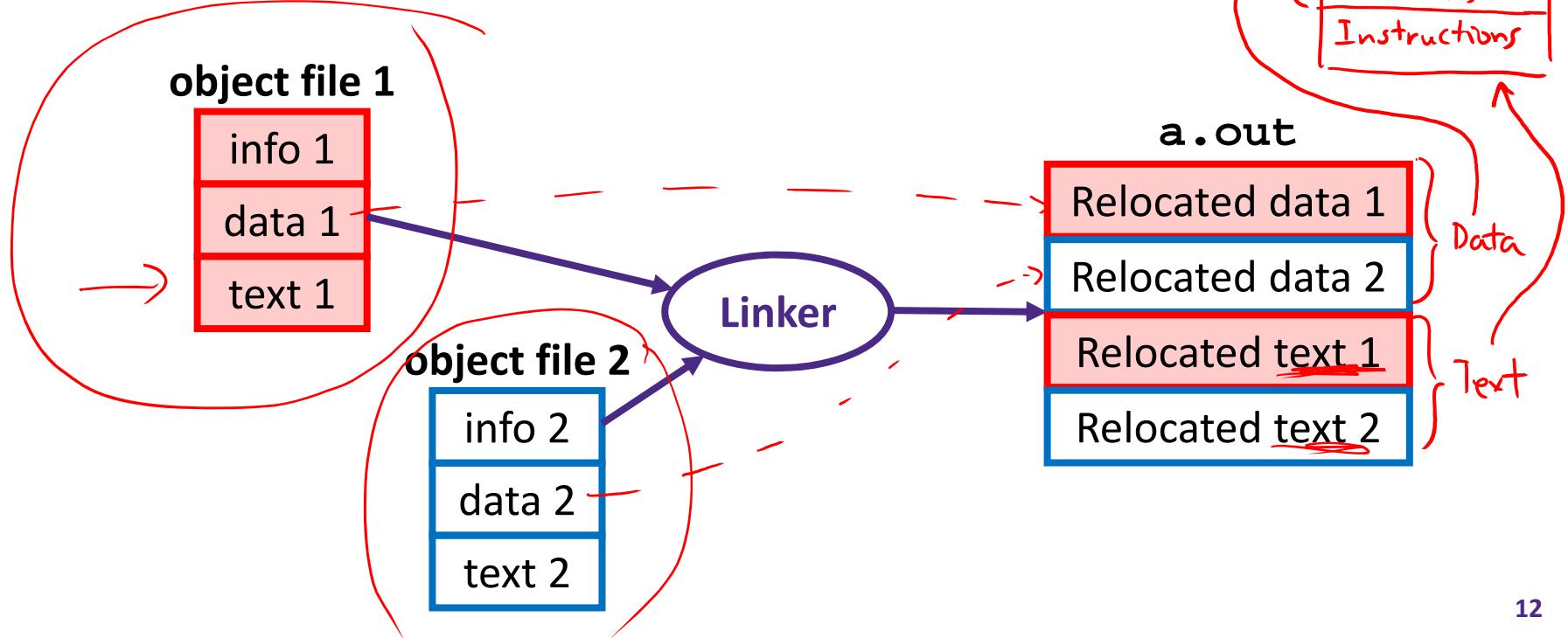
- 1) object file header: size and position of the other pieces of the object file "table of contents"
 - 2) text segment: the machine code (*Instructions*)
 - 3) data segment: data in the source file (binary) (*Static Data & Literals*)
 - 4) relocation table: identifies lines of code that need to be "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

- ❖ **Input:** Object files (e.g. ELF, COFF)
 - `foo.o`
- ❖ **Output:** executable binary program
 - `a.out`
- ❖ 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

- 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

- ❖ Disassembled:

```
0000000000400536 <sumstore>:  
400536: 48 01 fe          add    %rdi,%rsi  
        36 37 38           |       |  
400539: 48 89 32          mov    %rsi,(%rdx)  
        39 3a 3b           |       |  
40053c: c3                retq  
        3c               |  
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 `.out` (complete executable) or `.o` 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

- ❖ **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

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();
```

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs

Assembly language:

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

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



Data Structures in Assembly

❖ Arrays

- One-dimensional
- Multidimensional (nested)
- Multilevel

❖ Structs

- Alignment

❖ ~~Unions~~

Review: Array Allocation

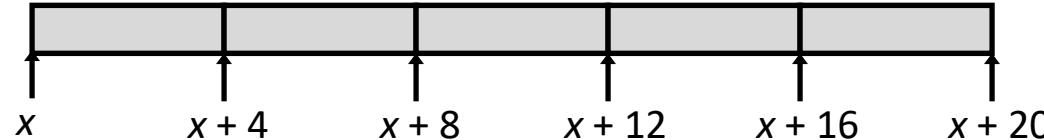
❖ Basic Principle

- ~~$T A[N]; \rightarrow$ array of data type T and length N~~
- ~~Contiguously allocated region of $N * \text{sizeof}(T)$ bytes~~
- Identifier A returns address of array (type T^*)

```
char msg[12];
```



```
int val[5];
```



```
double a[3];
```



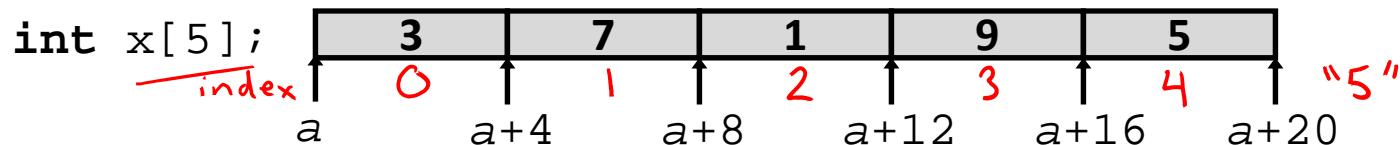
```
char* p[3];  
(or char *p[3]);
```



Review: Array Access

❖ Basic Principle

- $\mathbf{T} \ A[N]; \rightarrow \text{array of data type } \mathbf{T} \text{ and length } N$
- Identifier A returns address of array (type \mathbf{T}^*)



❖ Reference

Type	Value
------	-------

x[4]	int	5
--------	-----	---

x	int*	a
---	------	---

x+1 <i>← ptr arithmetic</i>	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 };
```

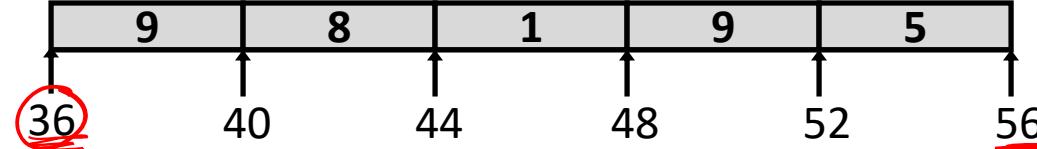
20 B each

```
int cmu[ 5 ];
```

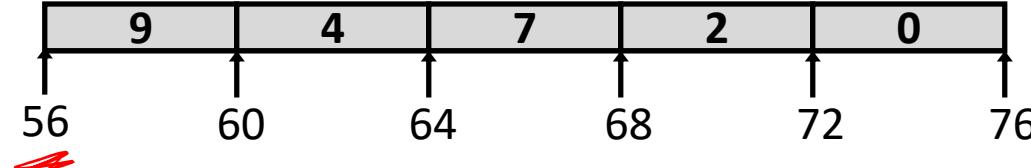


no gap in
this example

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



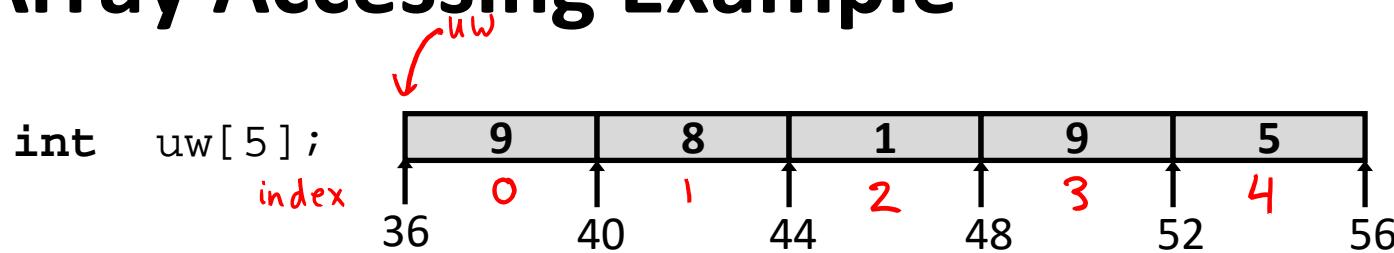
```
int ucb[ 5 ];
```



- ❖ 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

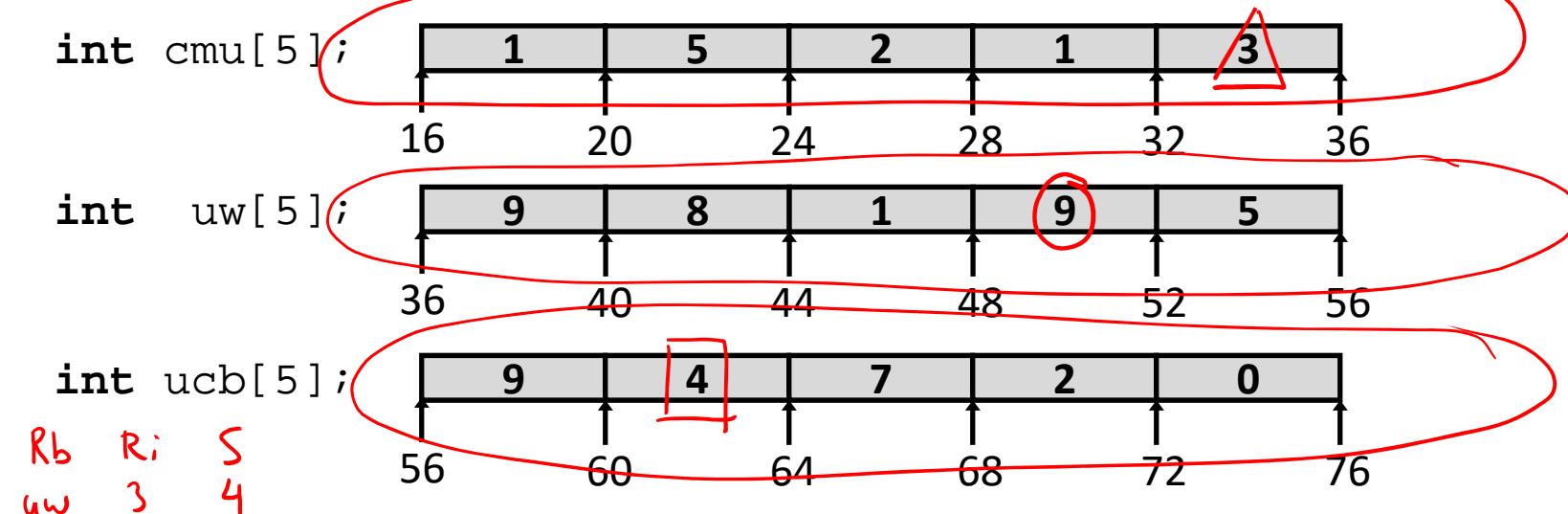


```
// return specified digit of ZIP code
int get_digit(int z[5], int digit) {
    return z[digit];
}
```

get_digit: *R_b* *R_i* *S*: scale factor (size of)
movl (%rdi,%rsi,4), %eax # *z[digit]*

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

Referencing Examples



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

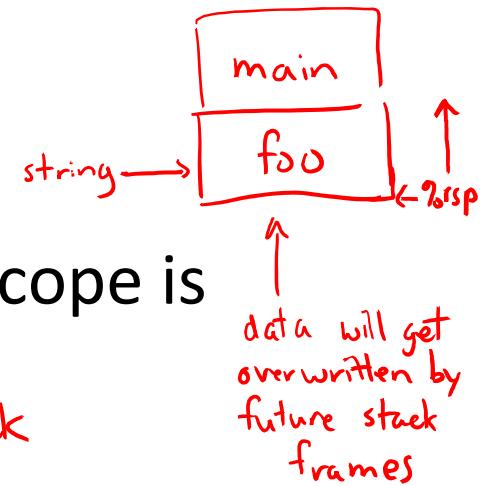
- ❖ Declared arrays only allocated while the scope is valid:

```
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

❖ ~~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 }};
```

} 2D array

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

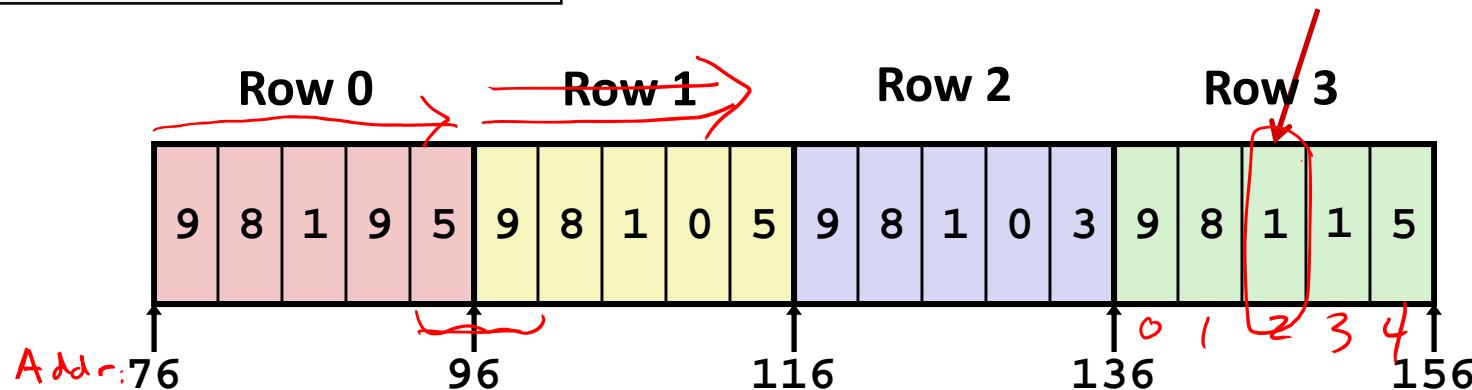
- ❖ What is the layout in memory?

Nested Array Example

```
int sea[4][5] =  
{ { 9, 8, 1, 9, 5 }, , ← red  
  { 9, 8, 1, 0, 5 } , ← yellow  
  { 9, 8, 1, 0, 3 } , ← blue  
  { 9, 8, 1, 1, 5 } } ; ← green
```

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

$\mathbf{sea[3][2];}$

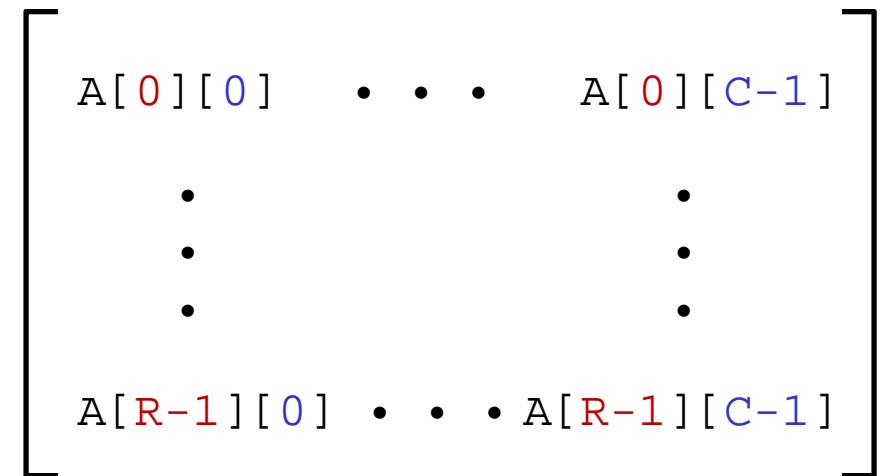


- ❖ “Row-major” ordering of all elements
- ❖ Elements in the same row are contiguous
- ❖ Guaranteed (in C)

Two-Dimensional (Nested) Arrays

❖ Declaration: $T A[R][C];$

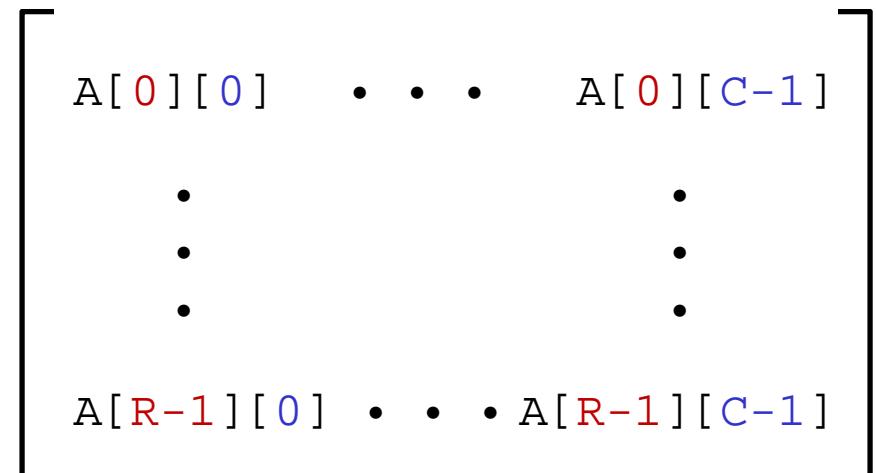
- 2D array of data type T
- R rows, C columns
- Each element requires `sizeof(T)` bytes



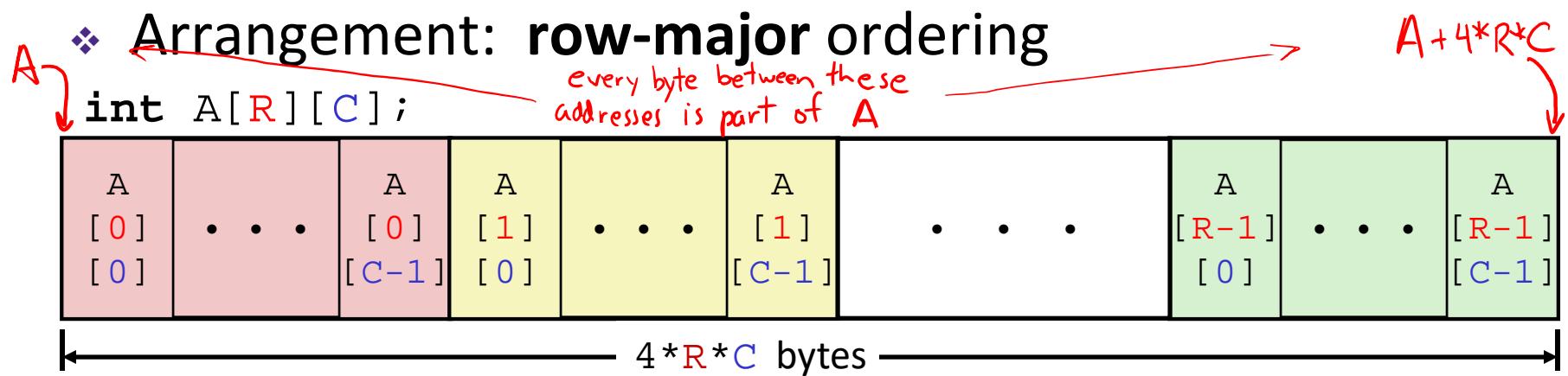
❖ Array size?

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

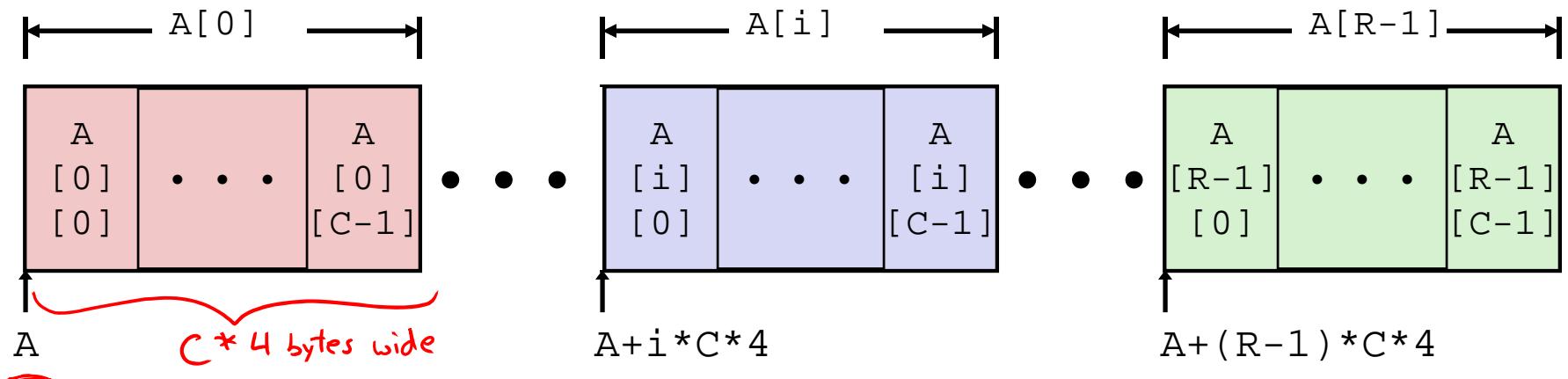


Nested Array Row Access

❖ Row vectors

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

```
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 }};
```

address of array →

```
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
    ...
    ...
```

ends up in memory!

Nested Array Row Access Code

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

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 }};
```

- What data type is `sea[index]`? *address*
- What is its value? $A + C * \text{sizeof}(T) * i \rightarrow \text{sea} + 5 * 4 * \text{index}$

```
# %rdi = index
leaq (%rdi,%rdi,4),%rax
leaq sea(,%rax,4),%rax
```

Translation?

using a label as 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 }};
```

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

just calculating an address, so no memory access

❖ Row Vector

- sea[index] is array of 5 ints
- Starting address = sea+20*index

❖ Assembly Code

- Computes and returns address
- Compute as: sea+4*(index+4*index) = sea+20*index

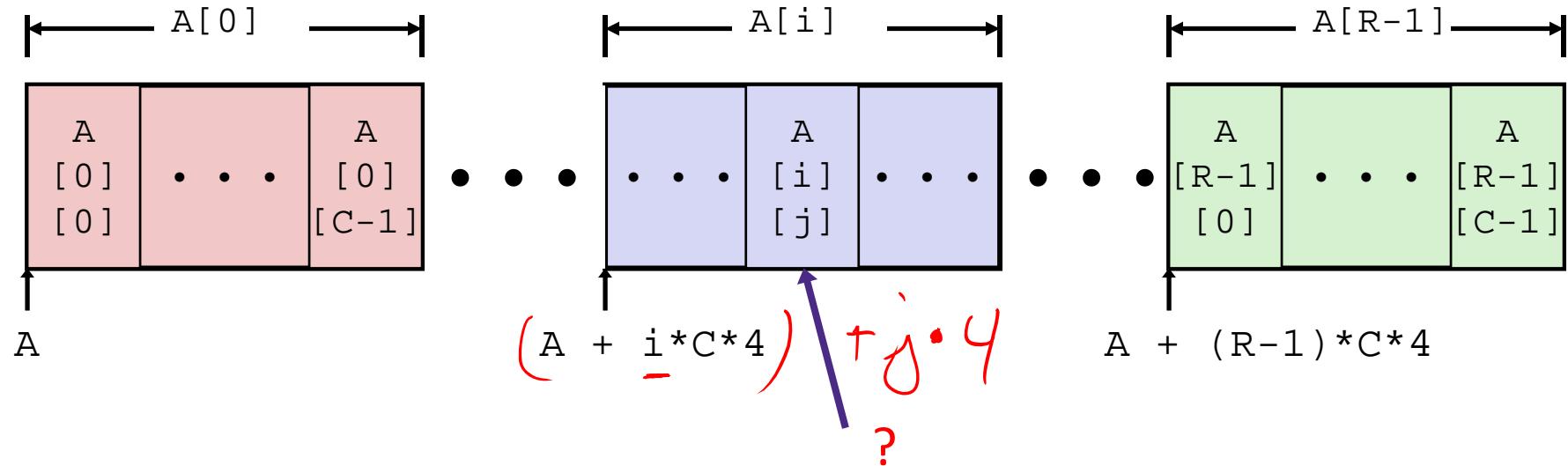
Nested Array Element Access

reminder: $\text{ar}[j] = *(\text{ar} + j)$

❖ 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

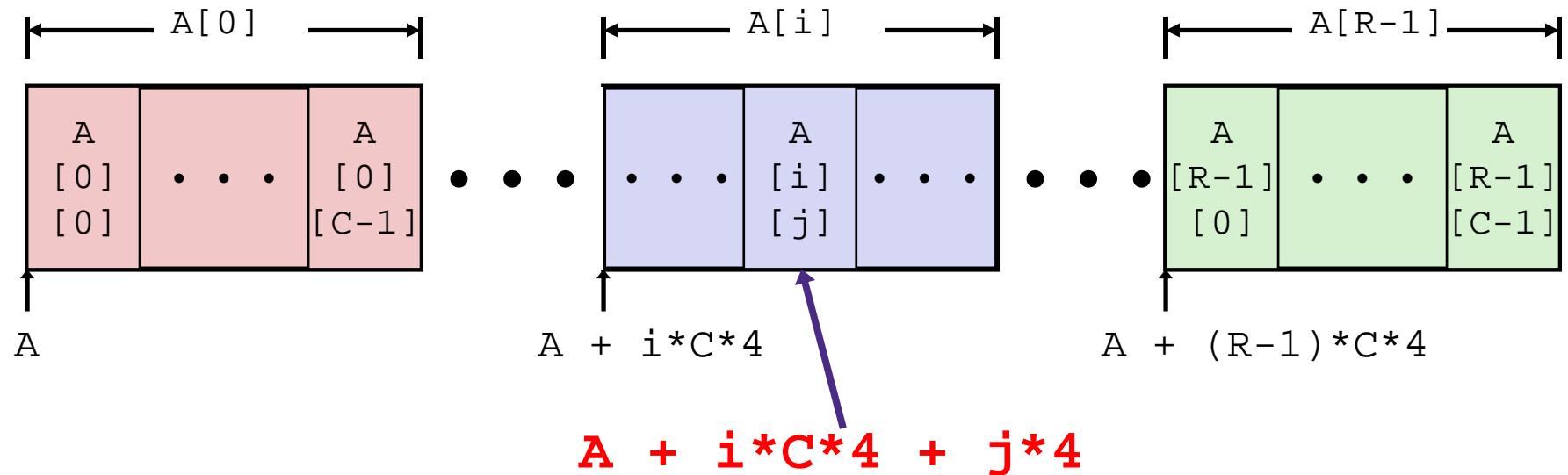
❖ Array Elements

- $A[i][j]$ is element of type \mathbf{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))
```

mov gets data

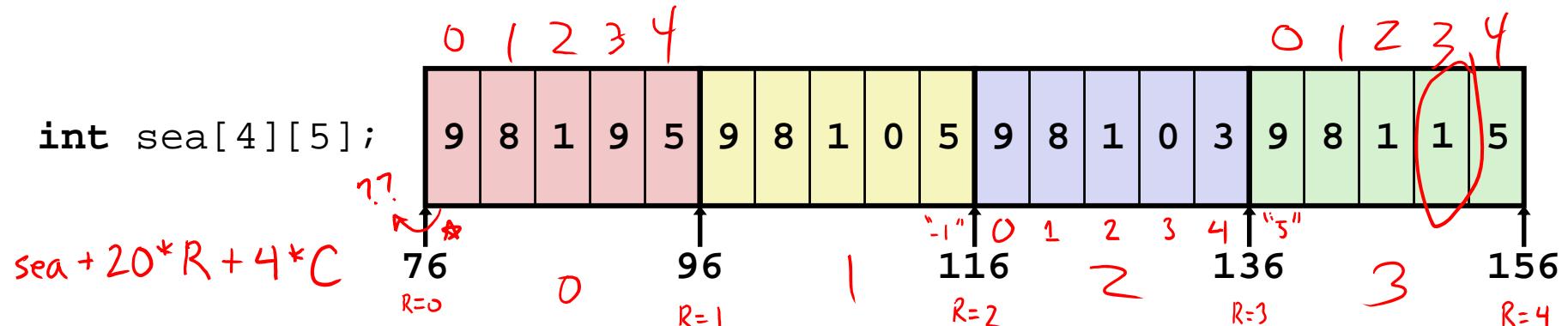
❖ Array Elements

- `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

Multidimensional Referencing Examples



Reference Address

<u>sea[3][3]</u>	$76 + 20 \cdot 3 + 4 \cdot 3 = 148$
<u>sea[2][5]</u>	$76 + 20 \cdot 2 + 4 \cdot 5 = 136$
<u>sea[2][-1]</u>	$76 + 20 \cdot 2 + 4 \cdot (-1) = 112$
<u>sea[4][-1]</u>	$76 + 20 \cdot 4 + 4 \cdot (-1) = 152$
<u>sea[0][19]</u>	$76 + 20 \cdot 0 + 4 \cdot (19) = 152$
<u>sea[0][-1]</u>	$76 + 20 \cdot 0 + 4 \cdot (-1) = 72$

Value Guaranteed?

1	Yes
9	Yes
5	Yes
5	Yes
5	Yes
??	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 }; 60B
```

~~int*~~ univ[3] = { uw, cmu, ucb } ; 24B

} could be apart

univ[0]

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
```

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

NO

univ2D[2][2] == 7

} guaranteed contiguous

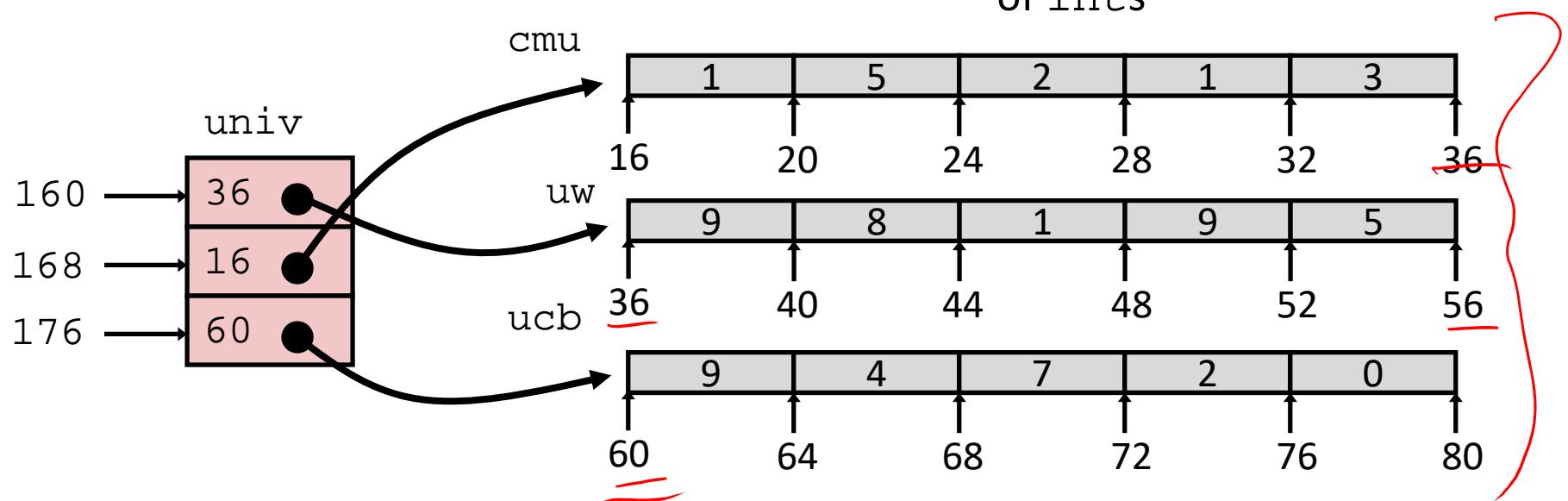
One array declaration = one contiguous block of memory

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};
```

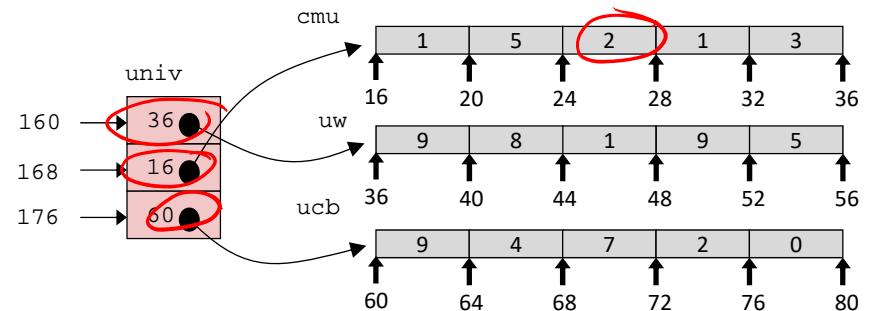
- ❖ Variable `univ` denotes array of 3 elements
- ❖ Each element is a pointer
 - 8 bytes each
- ❖ Each pointer points to array of `ints`



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
```

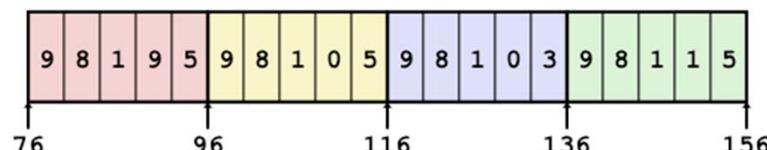
❖ Computation

- 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

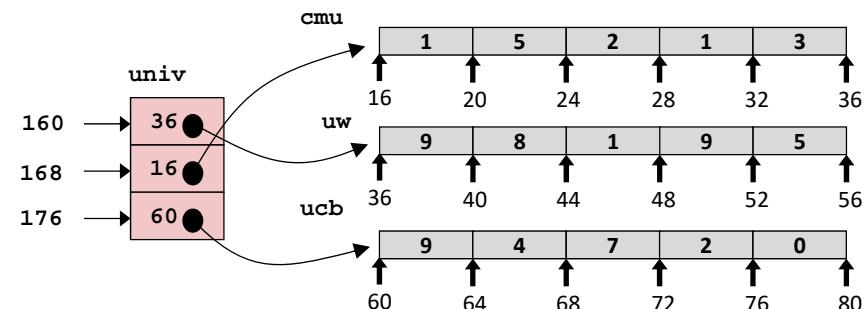
Multidimensional array

```
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];
}
```

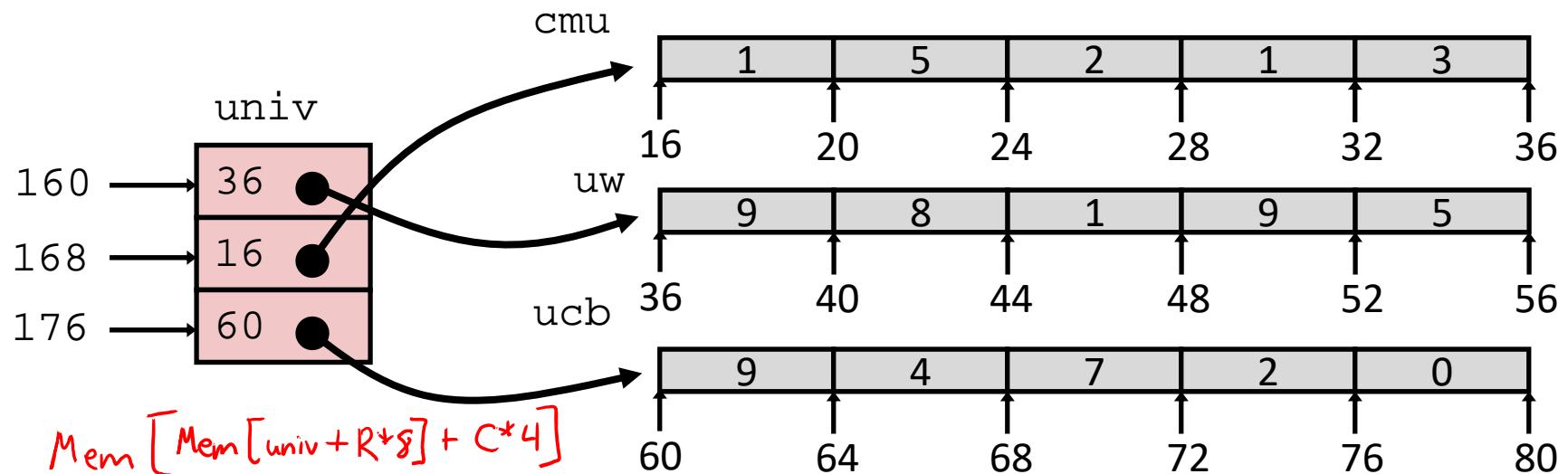


Access looks the same, but it isn't:

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

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

Multilevel Referencing Examples



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