Procedures & Arrays

CSE 351 Winter 2019

Instructors:

Max Willsey

Luis Ceze

Teaching Assistants:

Britt Henderson

Lukas Joswiak

Josie Lee

Wei Lin

Daniel Snitkovsky

Luis Vega

Kory Watson

Ivy Yu

http://xkcd.com/1270/

Administrivia

- Lab 2 due tonight! (Feb 8)
- HW 3 and mid-quarter survey due next week

- **Midterm** next Wednesday (Feb 13, in class)
  - Make a cheat sheet! – two-sided letter page, ***handwritten***
  - You get a reference sheet (see website)
  - Look out for announcements!

Register Conventions Summary

- Caller-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - Callee may change those register values
- Callee-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - Caller expects unchanged values in those registers
- Don't forget to restore/pop the values later!

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- **Illustration of Recursion**

Recursive Function

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

```
fact:
  testl    %edi, %edi
  jne      .L8
  movl     $1, %eax
  ret
.L8:
  pushq    %rbx
  movl     %edi, %ebx
  subl     $1, %edi
  call     fact
  imull    %ebx, %eax
  popq     %rbx
  ret
```

Compiler Explorer:
https://godbolt.org/z/pzs7N1
Try -O2!

Recursive Function: Base Case

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Register	Use(s)	Type
%rdi	n	Argument
%rax	Return value	Return value

```
fact:
  testl    %edi, %edi
  jne      .L8
  movl     $1, %eax
  ret
.L8:
  pushq    %rbx
  movl     %edi, %ebx
  subl     $1, %edi
  call     fact
  imull    %ebx, %eax
  popq     %rbx
  ret
```

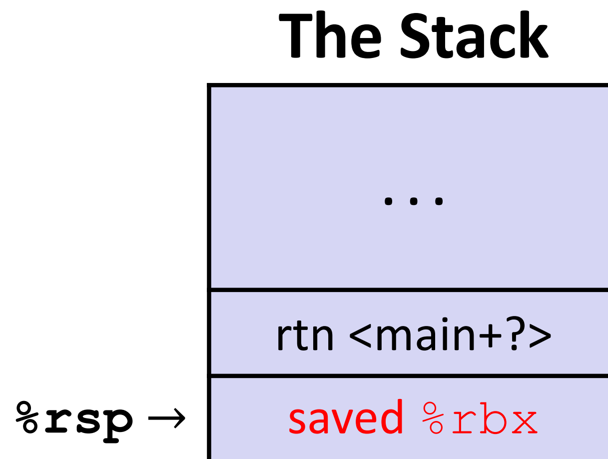
Recursive Function: Callee Register Save

Register	Use(s)	Type
%rdi	n	Argument

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Need original value of n *after* recursive call to fact.

"Save" by putting in %rbx (callee saved), but need to save old value of %rbx before you change it.

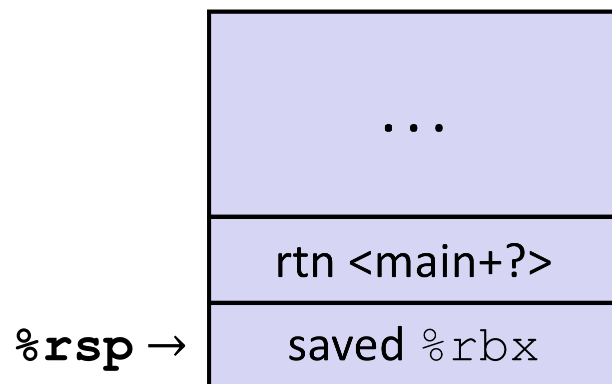



```
fact:
   testl    %edi, %edi
   jne      .L8
   movl     $1, %eax
   ret
.L8:
   pushq    %rbx
   movl     %edi, %ebx
   subl     $1, %edi
   call     fact
   imull    %ebx, %eax
   popq     %rbx
   ret
```

Recursive Function: Call Setup

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Register	Use(s)	Type
%rdi	n (new)	Argument
%rbx	n (old)	Callee saved



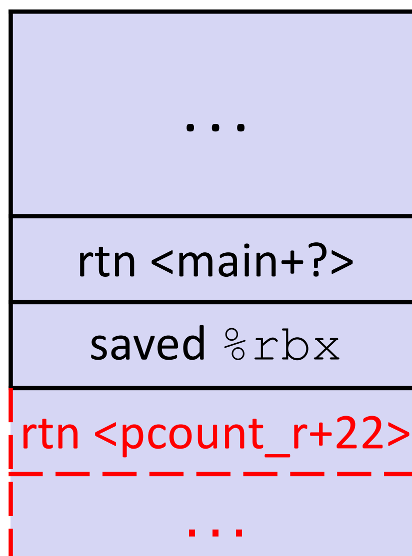

```
fact:
   testl    %edi, %edi
   jne      .L8
   movl     $1, %eax
   ret
.L8:
   pushq    %rbx
   movl     %edi, %ebx
   subl     $1, %edi
   call     fact
   imull    %ebx, %eax
   popq     %rbx
   ret
```

Recursive Function: Call

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Register	Use(s)	Type
`%rax`	Recursive call return value	Return value
`%rbx`	`n` (old)	Callee saved

The Stack



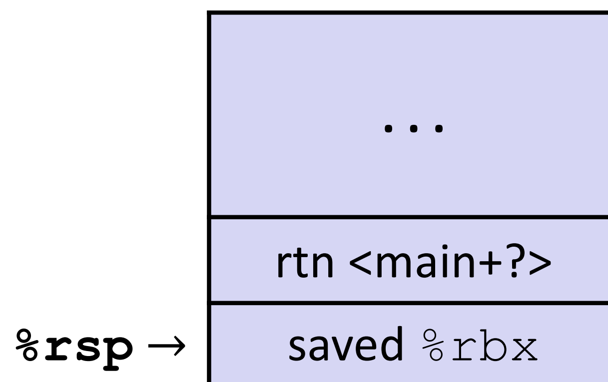

```
fact:
   testl    %edi, %edi
   jne      .L8
   movl     $1, %eax
   ret
.L8:
   pushq    %rbx
   movl     %edi, %ebx
   subl     $1, %edi
   call     fact
   imull    %ebx, %eax
   popq     %rbx
   ret
```

Recursive Function: Result

```
unsigned int fact(unsigned int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n-1);
}
```

Register	Use(s)	Type
%rax	Return value	Return value
%rbx	n	Callee saved

```
fact:
   testl    %edi, %edi
   jne      .L8
   movl     $1, %eax
   ret
.L8:
   pushq    %rbx
   movl     %edi, %ebx
   subl     $1, %edi
   call     fact
   imull    %ebx, %eax
   popq     %rbx
   ret
```



Observations About Recursion

- Works without any special consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the code explicitly does so (*e.g.* buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out (LIFO)
- Also works for mutual recursion (P calls Q; Q calls P)

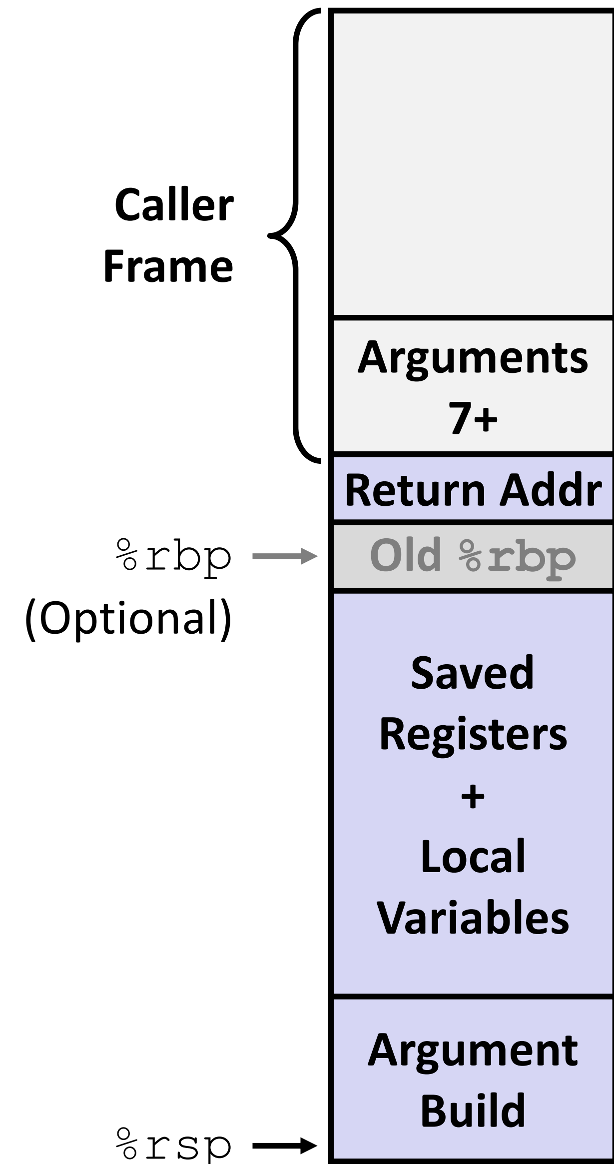
x86-64 Stack Frames

- Many x86-64 procedures have a minimal stack frame
  - Only return address is pushed onto the stack when procedure is called

- A procedure *needs* to grow its stack frame when it:
  - Has too many local variables to hold in **caller**-saved registers
  - Has local variables that are arrays or structs
  - Uses `&` to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Is using **caller**-saved registers and then calls a procedure
  - Modifies/uses **callee**-saved registers

x86-64 Procedure Summary

- Important Points
  - Procedures are a combination of *instructions* and *conventions*
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
  - Recursion handled by normal calling conventions
- Heavy use of registers
  - Faster than using memory
  - Use limited by data size and conventions
- Minimize use of the Stack



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();
```

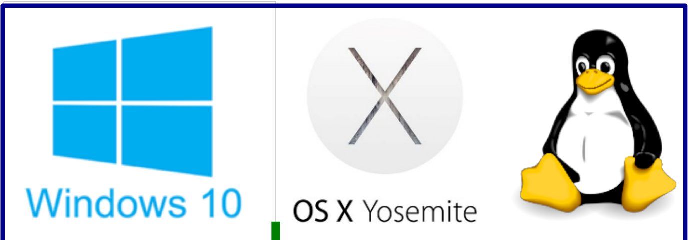
Assembly language:

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

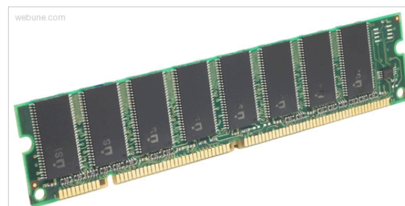
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



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

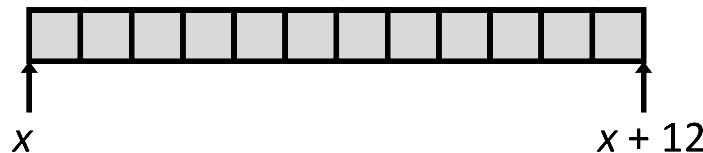
Data Structures in Assembly

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

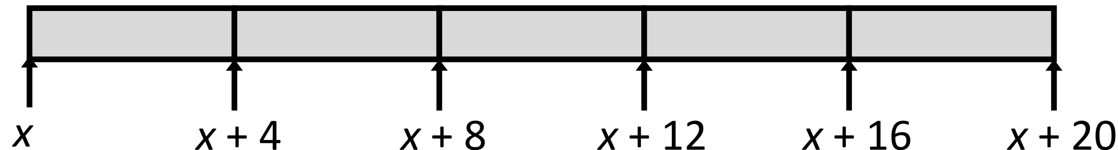
Array Allocation

- Basic Principle
  - `T A[N];` → array of data type **`T`** and length `N`
  - *Contiguously* allocated region of `N*sizeof(T)` bytes
  - Identifier `A` returns address of array (type **`T*`**)

`char` `msg[12];`

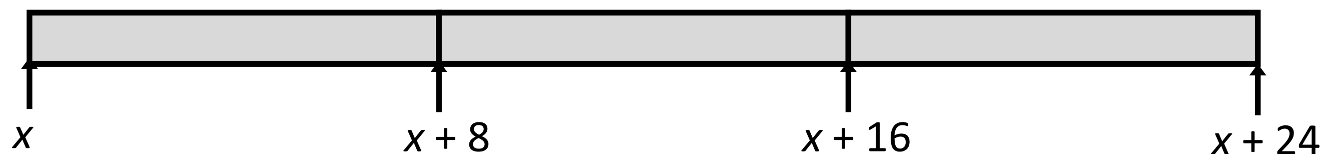


`int` `val[5];`



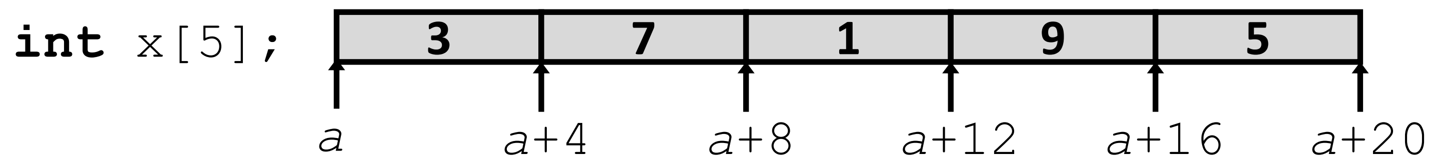
`double` `a[3];`

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



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
x[4]	**int**	5
x	**int***	a
x+1	**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

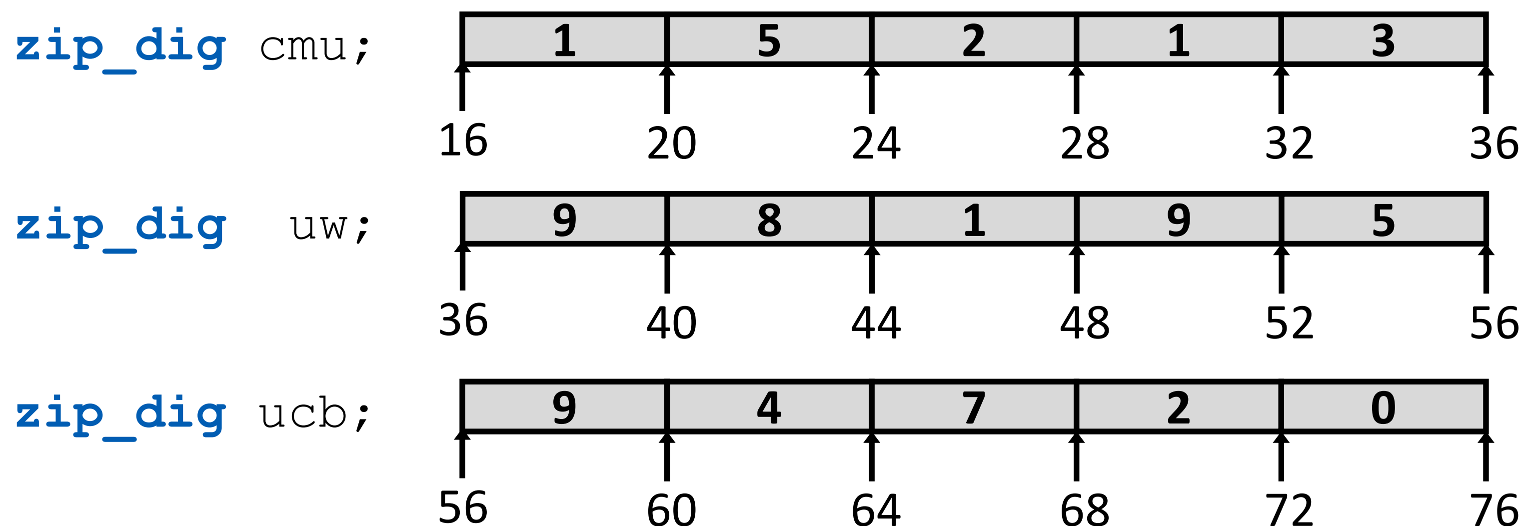
```
typedef int zip_dig[5];

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

- `typedef`: Declaration "`zip_dig uw`" equivalent to "`int uw[5]`"

Array Example

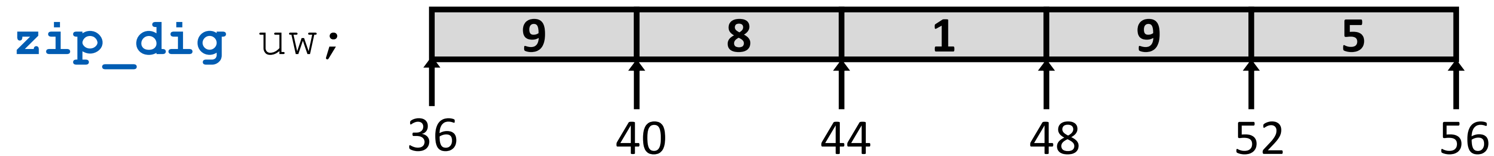
```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw  = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



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

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

Array Accessing Example

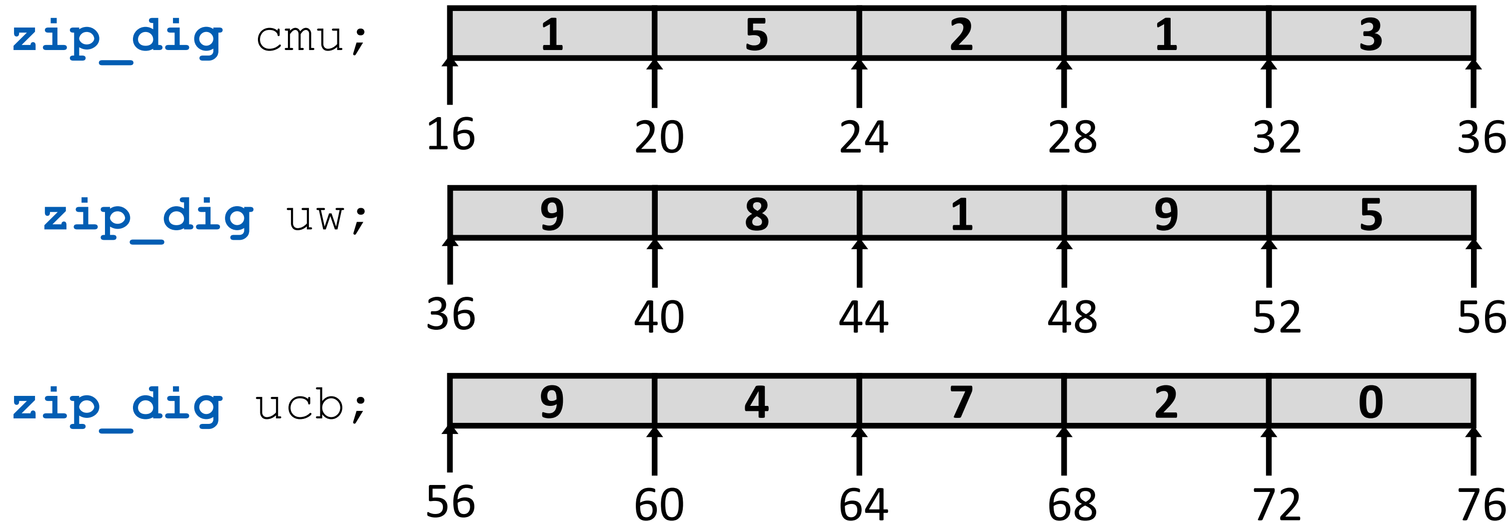



```
int get_digit(zip_dig z, int digit)
{
  return z[digit];
}
```

```
get_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`, so use memory reference `(%rdi,%rsi,4)`

Referencing Examples



Reference	Address	Value	Guaranteed?
`uw[3]`			
`uw[6]`			
`uw[-1]`			
`cmu[15]`			

- 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 looks like a pointer to the first (0th) element
  - `ar[0]` same as `*ar`; `ar[2]` same as `*(ar+2)`
- An array name is read-only (no assignment)
  - 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;
}
```

BAD!

- An array is passed to a function as a pointer:
  - Array size gets lost!

Really `int *ar`

```
int foo(int ar[], unsigned int size) {
    ... ar[size-1] ...
}
```

Must explicitly pass the size!

Data Structures in Assembly

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

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

Nested Array Example

```
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];
```

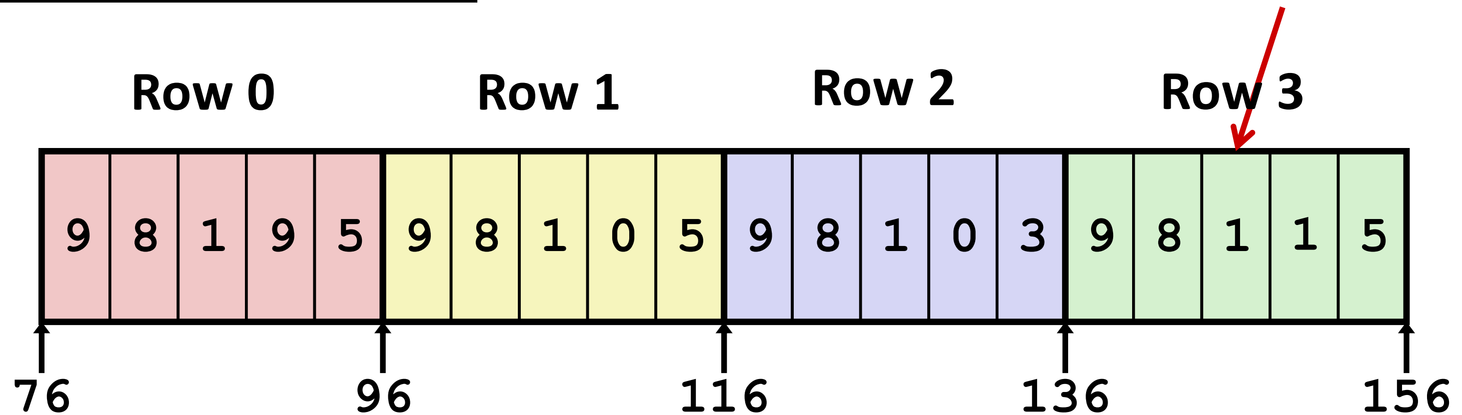
What is the layout in memory?

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

Nested Array Example

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

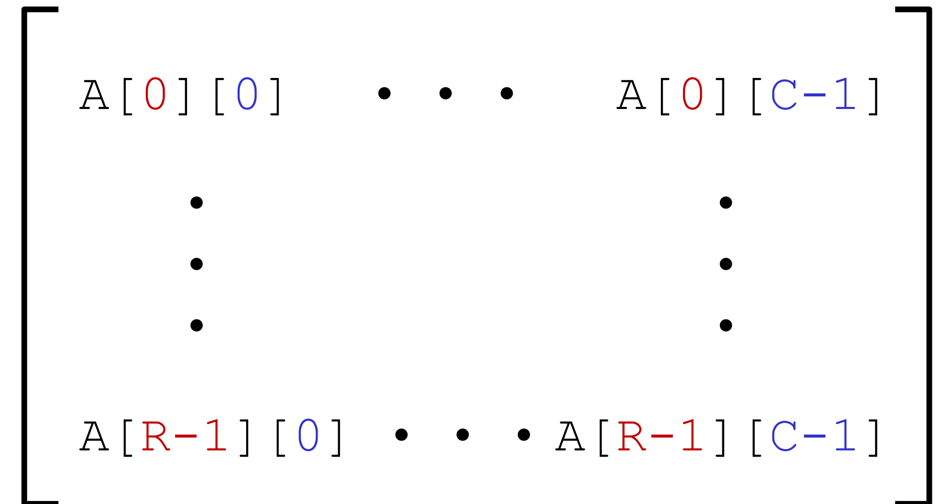
Remember, `T A[N]` is an array with elements of type `T`, with length `N`



- "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?



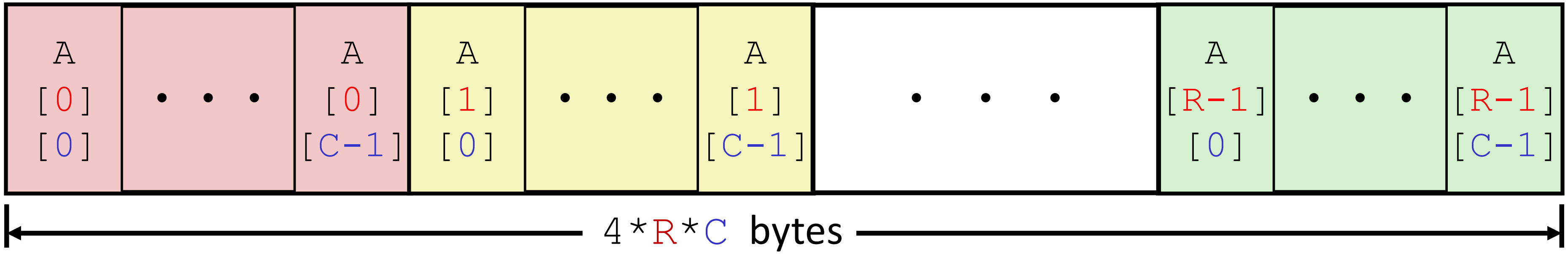
Two-Dimensional (Nested) Arrays

- 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

A[0][0] • • • A[0][C-1]

A[R-1][0] • • • A[R-1][C-1]

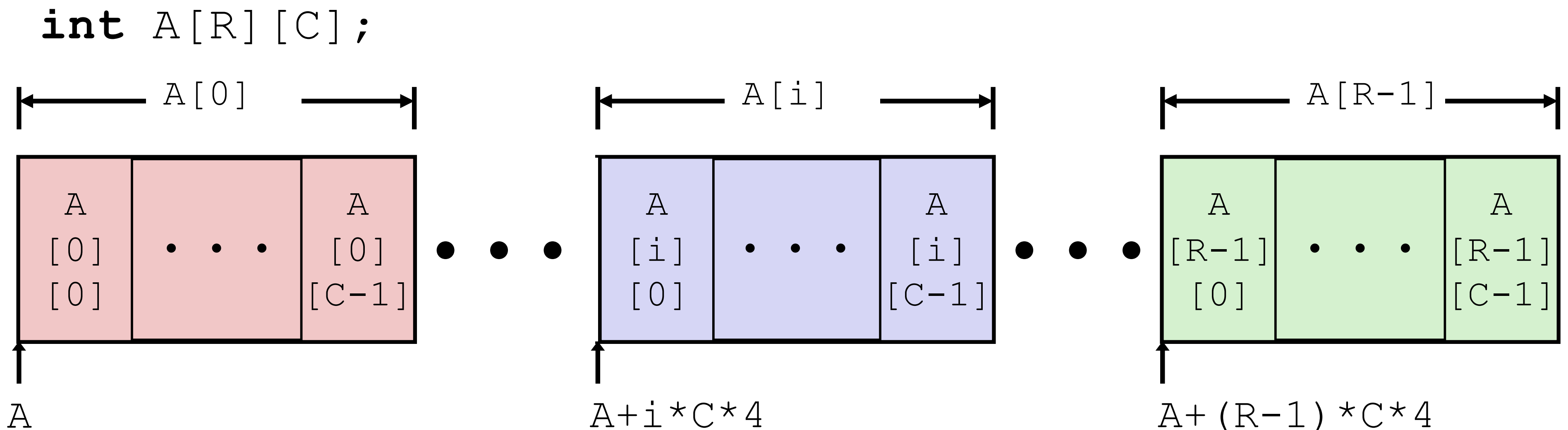
int A[R][C];



Nested Array Row Access

- Row vectors
  - Given **T** A[R][C],
    - A[i] is an array of C elements ("row i")
    - A is address of array
    - Starting address of row i = A + i*(C * sizeof(**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 }};
```

```
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
...
```

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 value?

```
  # %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];
}
```

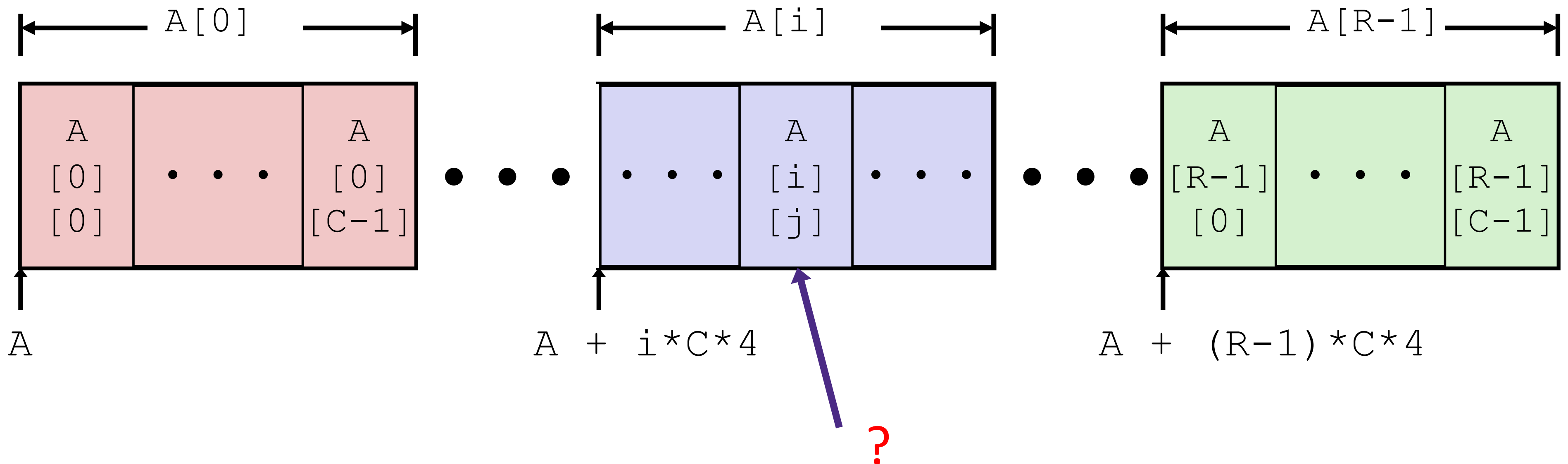
```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```
  # %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 `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

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

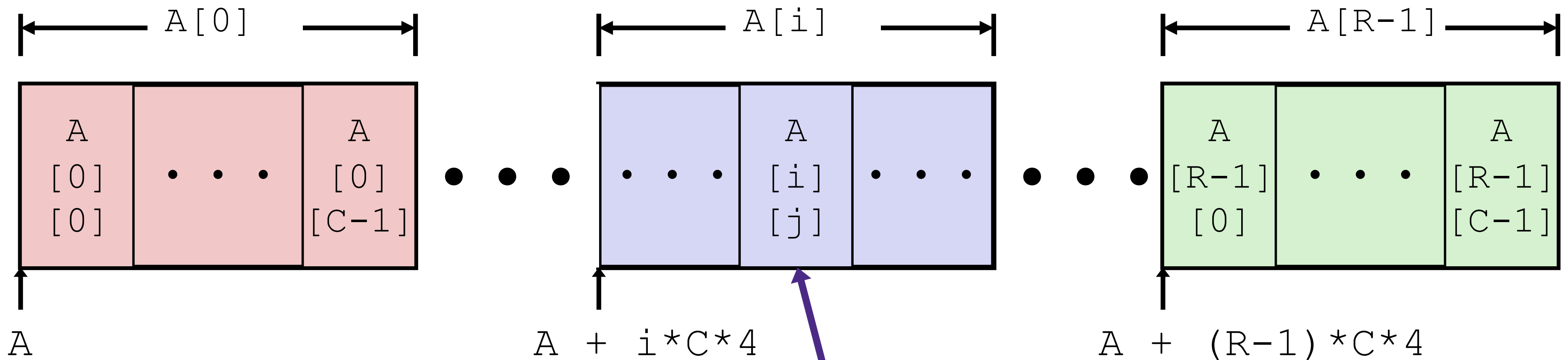


Nested Array Element Access

- 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

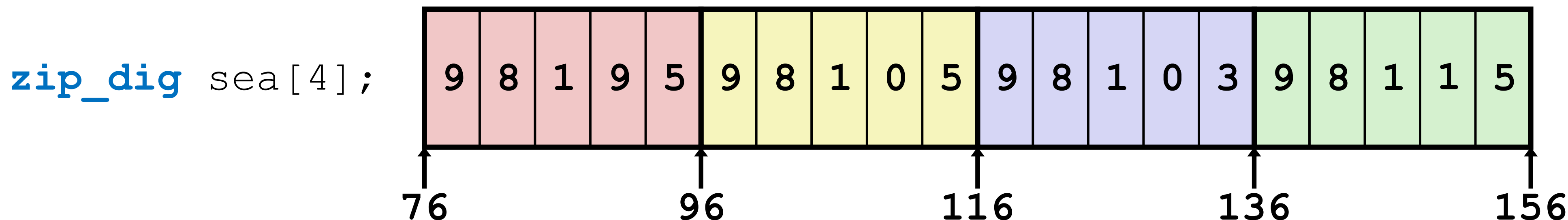
```
int get_sea_digit
   (int index, int digit)
{
  return sea[index][digit];
}
```

```
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 * index + digit) * 4`
- Assembly Code
  - Computes address as: `sea + ((index+4*index) + digit)*4`
  - `movl` performs memory reference

Multi-Dimensional Referencing Examples



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

Data Structures in Assembly

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

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};
```

Is a multi-level array the same thing as a 2D array?

NO

2D Array Declaration:

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

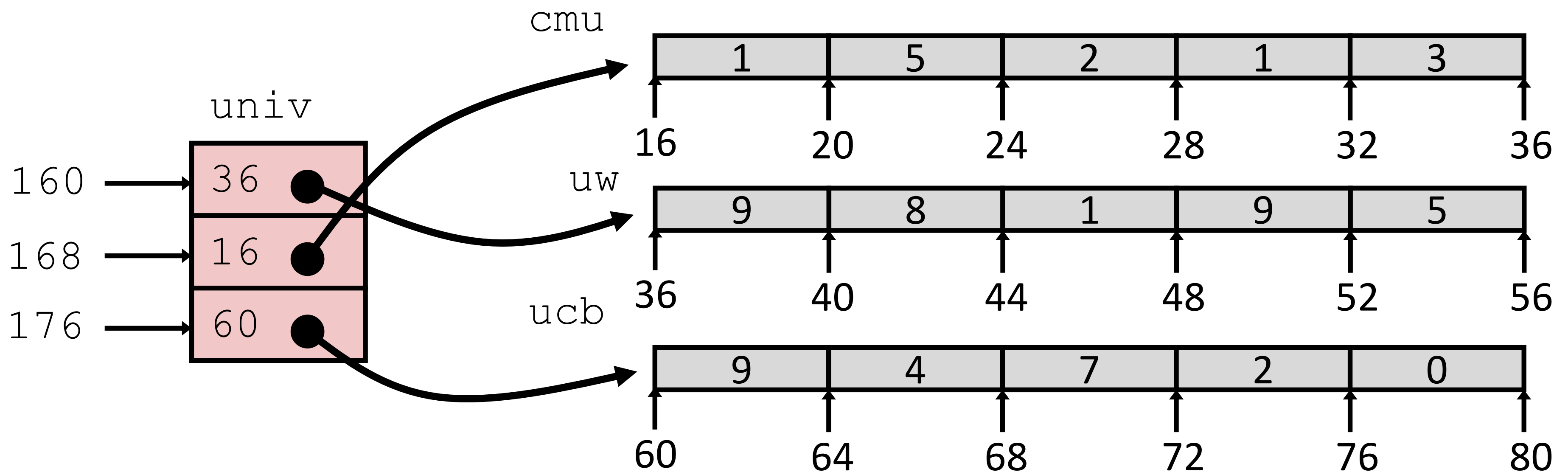
One array declaration = one contiguous block of memory

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};
```

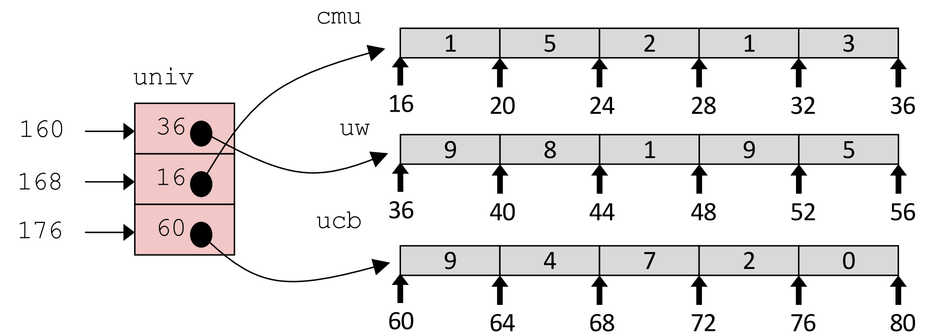
- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes each
- Each pointer points to array of `int`s



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];
}
```



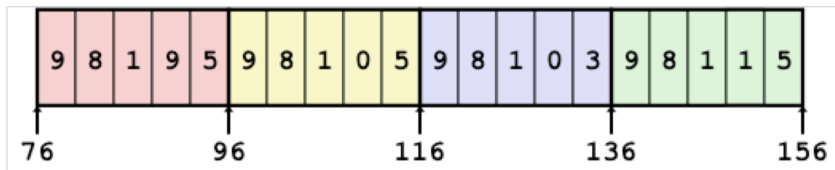

```
salq    $2, %rsi              # rsi = 4*digit
addq    univ(,%rdi,8), %rsi   # p = univ[index] + 4*digit
movl    (%rsi), %eax          # return *p
ret
```

- 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

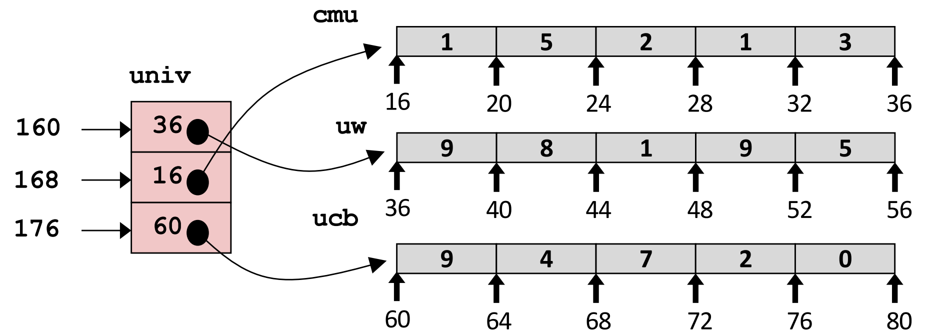
Nested array

```
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];
}
```

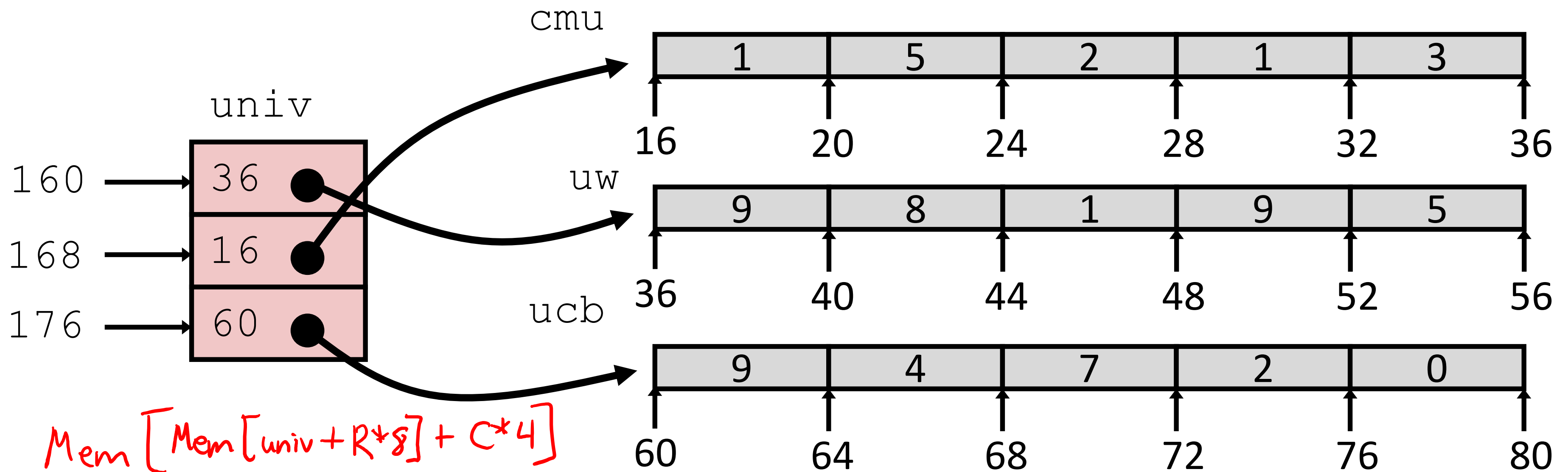


Access *looks* the same, but it isn't:

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

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

Multi-Level Referencing Examples



Reference	Address	Value	Guaranteed?
univ[2][3]	Mem[176]+3*4=60+12=72	2	Yes
univ[1][5]	Mem[168]+5*4=16+20=36	9	No
univ[2][-2]	Mem[176]+(-2)*4=60-8=52	5	No
univ[3][-1]	Mem[184]+(-1)*4= ??-4=??	???	No
univ[1][12]	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

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