Lecture 09
Pointers and Arrays

CS213 – Intro to Computer Systems

Branden Ghena – Spring 2021

Slides adapted from:
St-Amour, Hardavellas, Bustamente (Northwestern), Bryant, O'Hallaron (CMU), Garcia, Weaver (UC Berkeley)

Administriva

- Homework 3 will be released tomorrow

Today’s Goals

- Understand C arrays
  - Single and multi-dimensional

- And how they translate into assembly code

Outline

- **Pointers**
- One-dimensional Arrays
- Multi-dimensional Arrays
- Multi-level Arrays
- Dynamic arrays

Basic Data Types

- Integers
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used

Intel	**ASM**	**Bytes**	**C**
byte	`b`	1	`[unsigned] char`
word	`w`	2	`[unsigned] short`
double word	`l`	4	`[unsigned] int`
quad word	`q`	8	`[unsigned] long int`

Floating point data

- Won’t be focusing on floating point
  - Has changed much more than integer types across updates
  - Not all x86-64 machines have the same capabilities here
- Registers %xmm0 - %xmm15
  - 128-bit registers
  - On newest machines refer to as %ZMM0-%ZMM31 (512-bit registers)
- Instructions
  - addss (add scalar single-precision)
  - addsd (add scalar double-precision)
  - addpd (add packed double-precision, two doubles at once)

More complex data types

- Pointers and Arrays (today's lecture)

```
int* a  = &v;

int list[2] = {15, 27};
```

- Structs and Unions (next lecture)

```
typedef struct {
   int a;
   char b;
   int* c;
} mystruct_t;
```

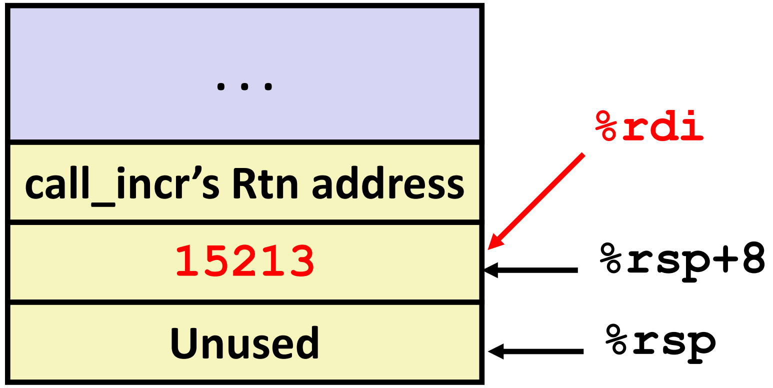
Example pointer code: calling `incr`

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movq    $3000, %rsi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

- Pointers are addresses
- `v1` must be stored on stack
  - Why? need to create pointer to it
- Compute pointer as `8(%rsp)`
  - Use `leaq` instruction

Memory (stack)



Register	Use(s)
`%rdi`	`&v1`
`%rsi`	3000

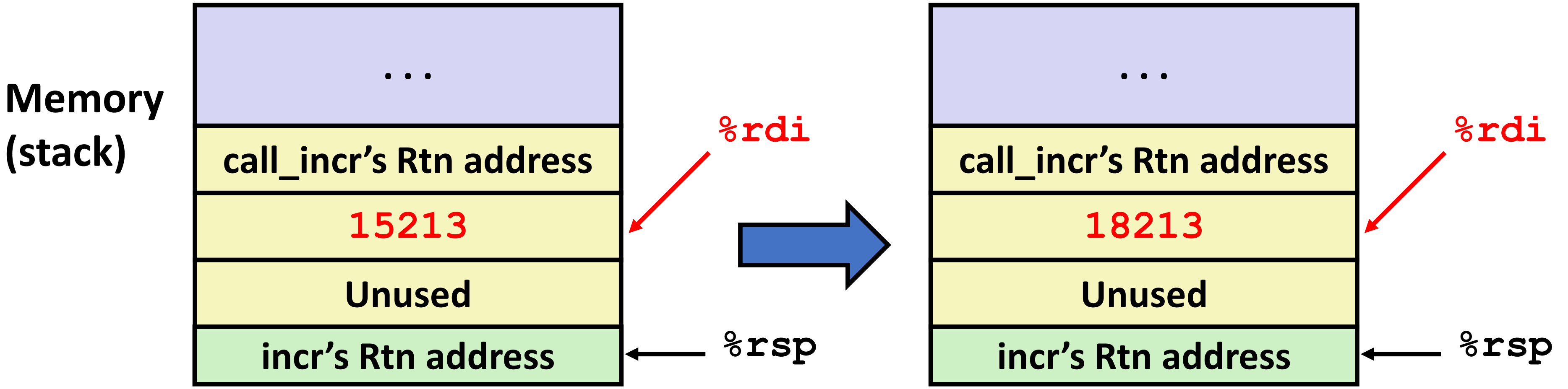
Example pointer code : executing `incr`

```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
incr:
  movq     (%rdi), %rax
  addq     %rax, %rsi
  movq     %rsi, (%rdi)
  ret
```

Register	Use(s)
`%rdi`	Argument `p`
`%rsi`	Argument `val` (3000)
`%rax`	. . .

Register	Use(s)
`%rdi`	Argument `p`
`%rsi`	18213
`%rax`	15213 (return value)



Pointers to global variables

```c
int global_var = 15;

int* myfunc(void) {
  global_var += 2;
  return &global_var;
}
```

```
  .text
  .globl myfunc
  .type myfunc, @function
myfunc:
  addl $2, 0x2f1f(%rip)
  mov $0x404028, %eax
  ret
  .globl global_var
  .data
  .align4
  .type global_var, @object
  .size global_var, 4
global_var:
  .long 15
```

Naming constants

These two are the same code.
One just uses a name for the constant.

```
   .text
   .globl myfunc
   .type myfunc, @function
myfunc:
   addl $2, 0x2f1f(%rip)
   mov $0x404028, %eax
   ret
   .globl global_var
   .data
   .align4
   .type global_var, @object
   .size global_var, 4
global_var:
   .long 15
```

```
  .text
   .globl myfunc
   .type myfunc, @function
myfunc:
   addl $2, global_var(%rip)
   mov $global_var, %eax
   ret
   .globl global_var
   .data
   .align4
   .type global_var, @object
   .size global_var, 4
global_var:
   .long 15
```

Outline

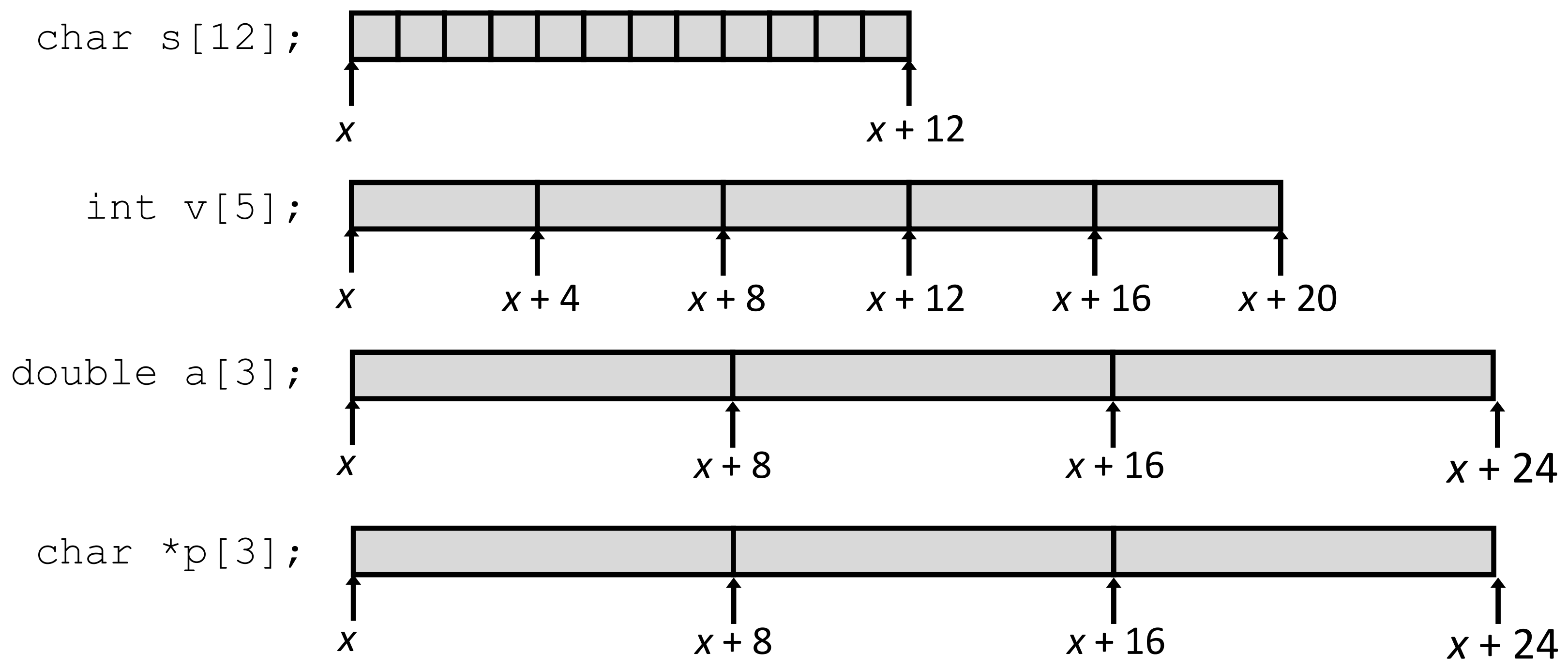
- Pointers
- **One-dimensional Arrays**
- Multi-dimensional Arrays
- Multi-level Arrays
- Dynamic arrays

One-Dimensional Array Allocation

- Basic Principle

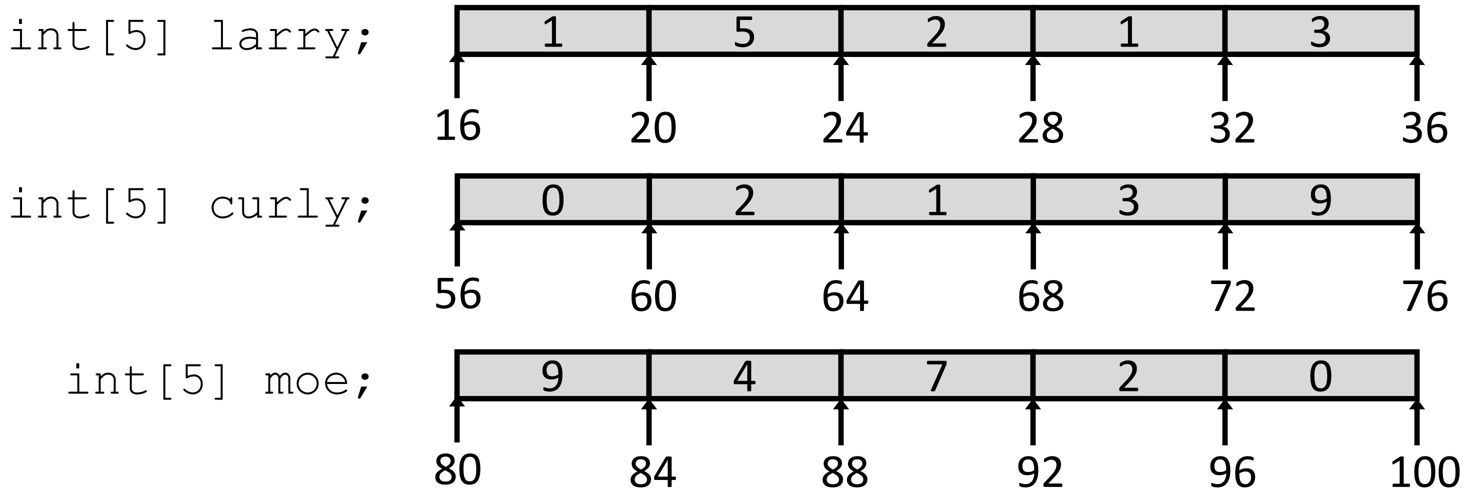
  T **`A[`***L***`];  // e.g., int A[4];`**

  - Array of data type *T* and length *L*
  - Contiguously allocated region in memory of *L* * **`sizeof`**(*T*) bytes



Placing arrays at addresses

```
int[5] larry = { 1, 5, 2, 1, 3 };
int[5] curly = { 0, 2, 1, 3, 9 };
int[5] moe   = { 9, 4, 7, 2, 0 };
```



- Each array is allocated in contiguous 20 byte blocks
  - But no guarantee that `curly[]` will be right after `larry[]`, etc.

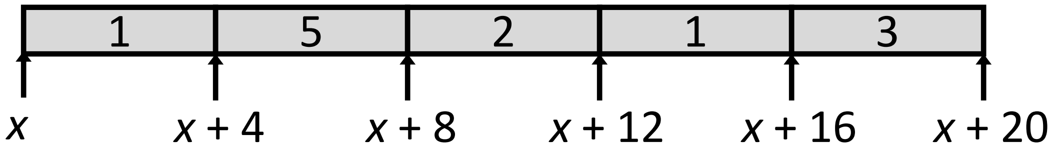
Array Access and Pointer Arithmetic

- Basic Principle

  T **`A[`***L***`];`**

  - Identifier **`A`** can be used as a pointer to array element 0: **`A`** is of type *T**
  - ***Warning***: in C, arrays count # of elements, in assembly count # bytes!

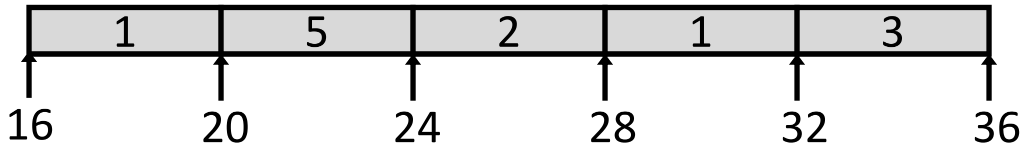
`int val[5];`



Reference	Type	Value	
`val[4]`	**`int`**	3	
`val`	**`int *`**	*x*	
`val+1`	**`int *`**	*x* + 4	
`&val[2]`	**`int *`**	*x* + 8	
`val[5]`	**`int`**	??	**No array bounds checking!!!**
`*(val+1)`	**`int`**	5	
`val + `*i*	**`int *`**	*x* + 4 *i*	

One-Dimensional Array Accessing Example

```
int[5] larry;
```




```
int get_digit(int[5] larry, size_t digit)
{
  return larry[digit];
}
```

```
get_digit:
  # %rdi = larry
  # %rsi = digit
  movl (%rdi,%rsi,4),%rax   # z[digit]
  retq
```

`%rdi` -> starting address of array

`%rsi` -> array index

- Desired digit at `%rdi + 4*%rsi`
- Use memory addressing!
  `(%rdi,%rsi,4)`

- Now we see why x86 memory operands are `D(Rb, Ri, s)`
  - Scale 1, 2, 4, or 8

One-Dimensional Array Loop Example

```
void zincr(int *z) {
  size_t i;
  for (i = 0; i < 4; i++)
    z[i]++;
}
```

```
zincr:
  # %rdi = z
  movl    $0, %eax           #   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
  jbe     .L4                #   if i<=4, goto loop
  retq
```

Quiz + Break

```
z -> %rdi

i -> %rax

addl $1, (%rdi,%rax,4) #   z[i]++ (int z[])
```

- What changes if z is instead an array of:
  - short
  - char
  - bool
  - char*
  - unsigned int

Quiz + Break

```
z -> %rdi

i -> %rax

addl $1, (%rdi,%rax,4) #   z[i]++ (int z[])
```

- What changes if z is instead an array of:
  - short `addl $1, (%rdi,%rax,2)`
  - char
  - bool
  - char*
  - unsigned int

Quiz + Break

```
z -> %rdi

i -> %rax

addl $1, (%rdi,%rax,4) #   z[i]++ (int z[])
```

- What changes if z is instead an array of:
  - short `addl $1, (%rdi,%rax,2)`
  - char `addl $1, (%rdi,%rax,1)`
  - bool
  - char*
  - unsigned int

Quiz + Break

```
z -> %rdi
i -> %rax
addl $1, (%rdi,%rax,4) #   z[i]++ (int z[])
```

- What changes if z is instead an array of:
  - short `addl $1, (%rdi,%rax,2)`
  - char `addl $1, (%rdi,%rax,1)`
  - bool `addl $1, (%rdi,%rax,1)`
  - char*
  - unsigned int

Quiz + Break

```
z -> %rdi
i -> %rax
addl $1, (%rdi,%rax,4) #   z[i]++ (int z[])
```

- What changes if z is instead an array of:
  - short `addl $1, (%rdi,%rax,2)`
  - char `addl $1, (%rdi,%rax,1)`
  - bool `addl $1, (%rdi,%rax,1)`
  - char* `addl $1, (%rdi,%rax,8)`
  - unsigned int

Quiz + Break

```
z -> %rdi

i -> %rax

addl $1, (%rdi,%rax,4) #   z[i]++ (int z[])
```

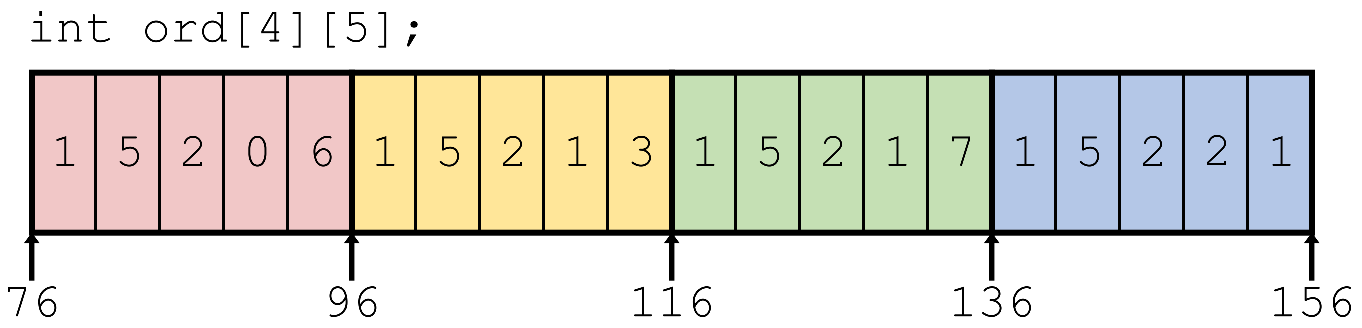
- What changes if z is instead an array of:
  - short `addl $1, (%rdi,%rax,2)`
  - char `addl $1, (%rdi,%rax,1)`
  - bool `addl $1, (%rdi,%rax,1)`
  - char* `addl $1, (%rdi,%rax,8)`
  - unsigned int Nothing. Still 4 bytes. add works the same on sign/unsigned

Outline

- Pointers
- One-dimensional Arrays
- **Multi-dimensional Arrays**
- Multi-level Arrays
- Dynamic arrays

Multidimensional (Nested) Array Example

```
int ord[4][5] =
  /* 4 rows, 5 cols */
  {{1, 5, 2, 0, 6 },
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



- Let's decipher "`int ord[4][5]`"
  - `int` **`ord[4]`** `[5]`: **`ord`** is an array of **4** elements, allocated contiguously
  - **`int`** `ord[4]` **`[5]`**: Each element is an array of **5** **`int`**'s, allocated contiguously

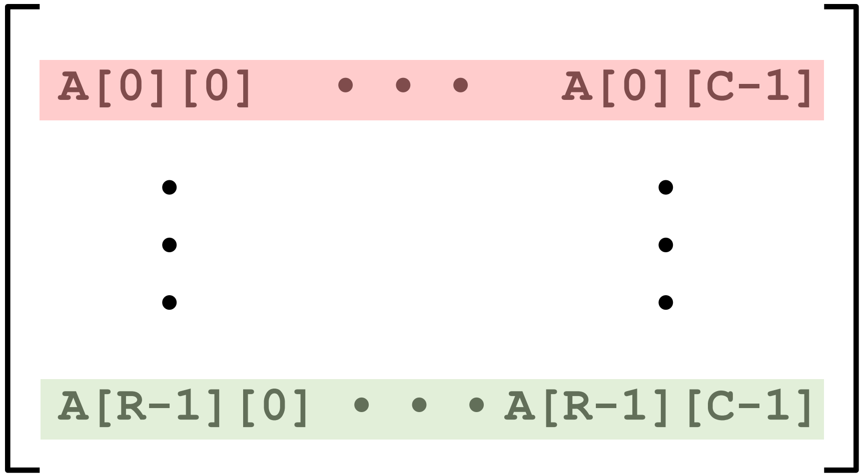
- "Row-Major" ordering of all elements guaranteed
  - Entire row (all columns in it) will be placed in memory before the next row starts

Multidimensional (Nested) Arrays

- Declaration

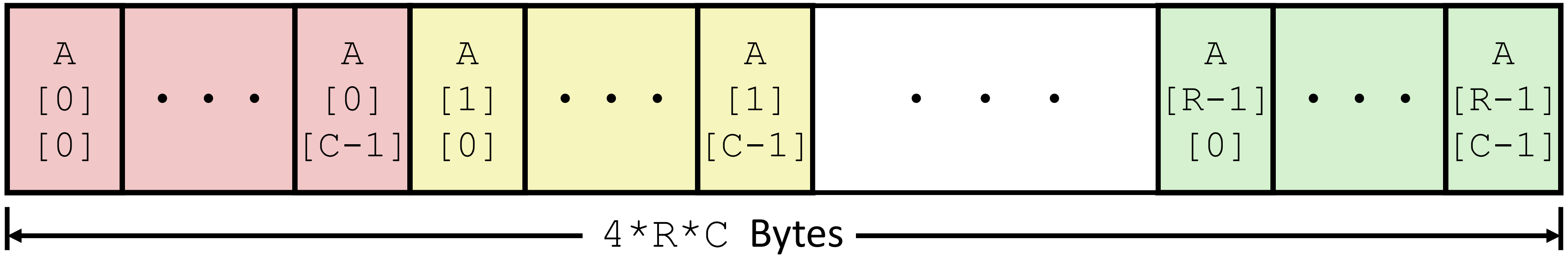
  T **A**[*R*][*C*];

  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes



- Types
  - *What is A?* *T [R] [C]*
  - *What is A[i]?* *T [C]*
  - *What is A[i][j]?* *T*

- Arrangement
  - Row-Major Ordering

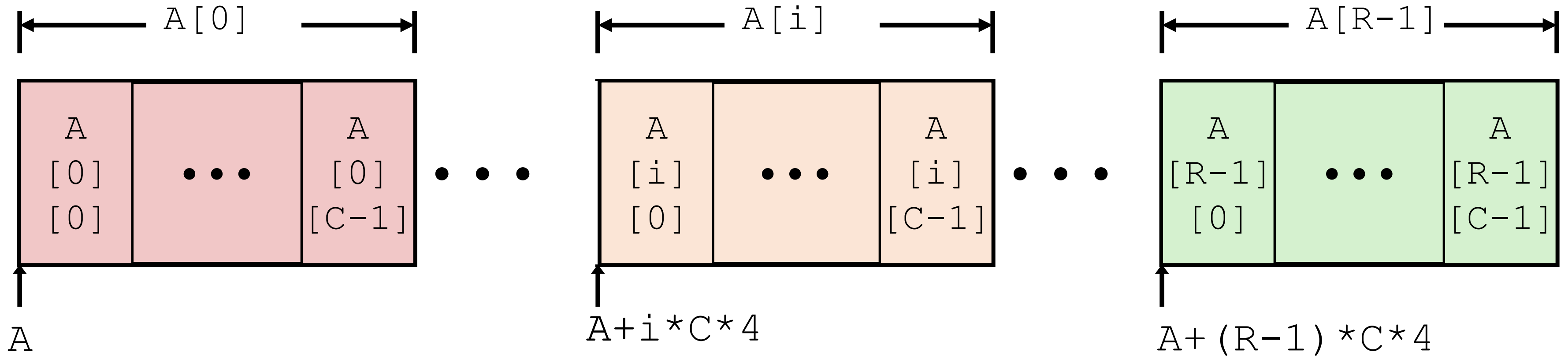



```
int A[R][C];
```

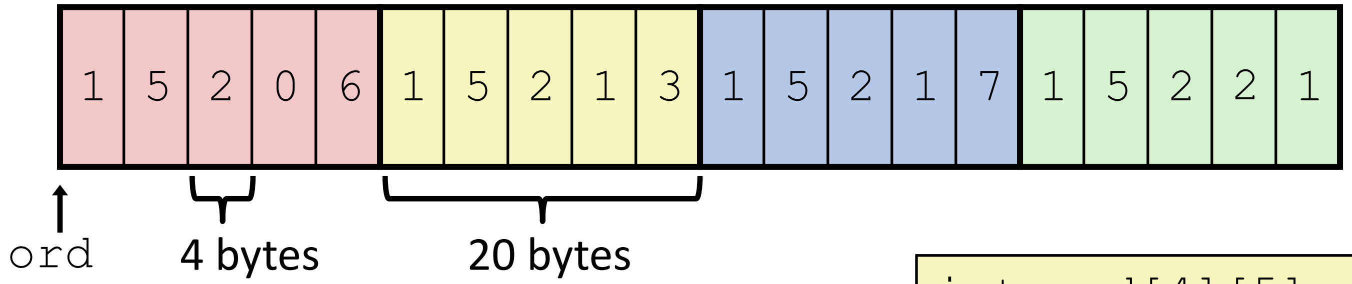
Nested Array Row Access

- To figure out how to get the element we want
  - Let's first figure out how to get the row we want (its starting address)
- Row Vectors
  - **`A[i]`** (row) 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

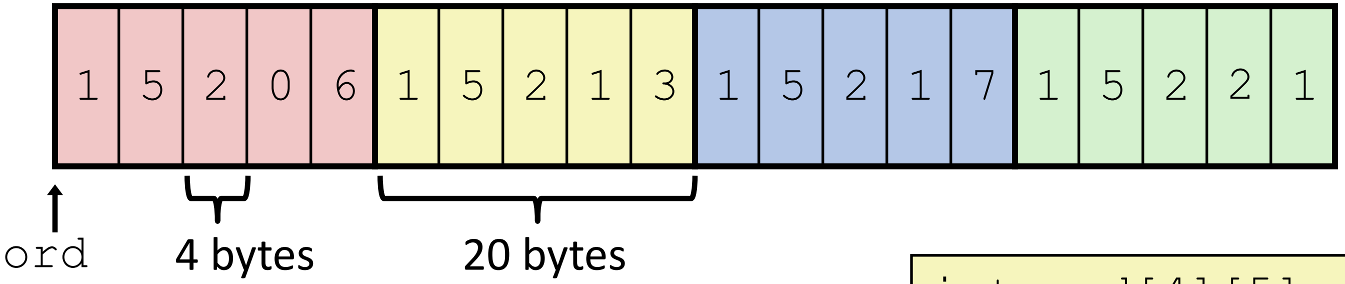



```
int *get_ord_row(size_t index)
{
  return ord[index];
}
```

```
int ord[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

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

Nested Array Row Access Code



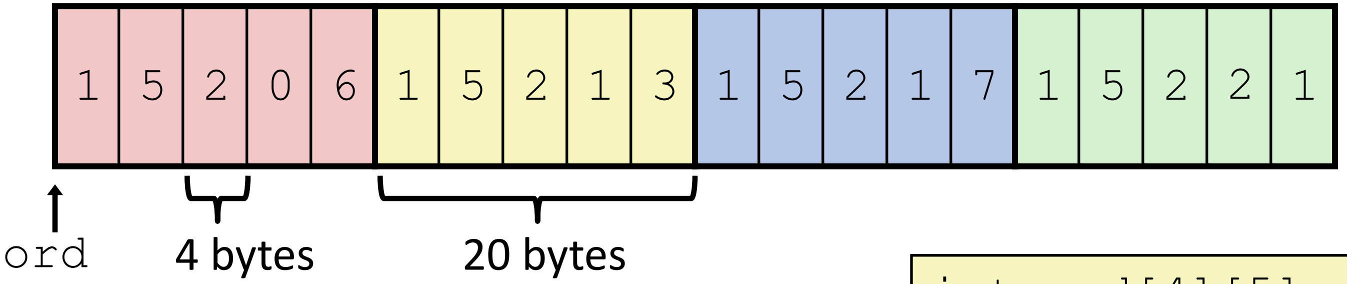

```
int *get_ord_row(size_t index)
{
  return ord[index];
}
```

```
int ord[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

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

- What's that displacement?
  - Constant address
  - `ord` is a global. Always in a location known at compile-time. So constant address!

Nested Array Row Access Code




```
int *get_ord_row(size_t index)
{
  return ord[index];
}
```

```
int ord[4][5] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

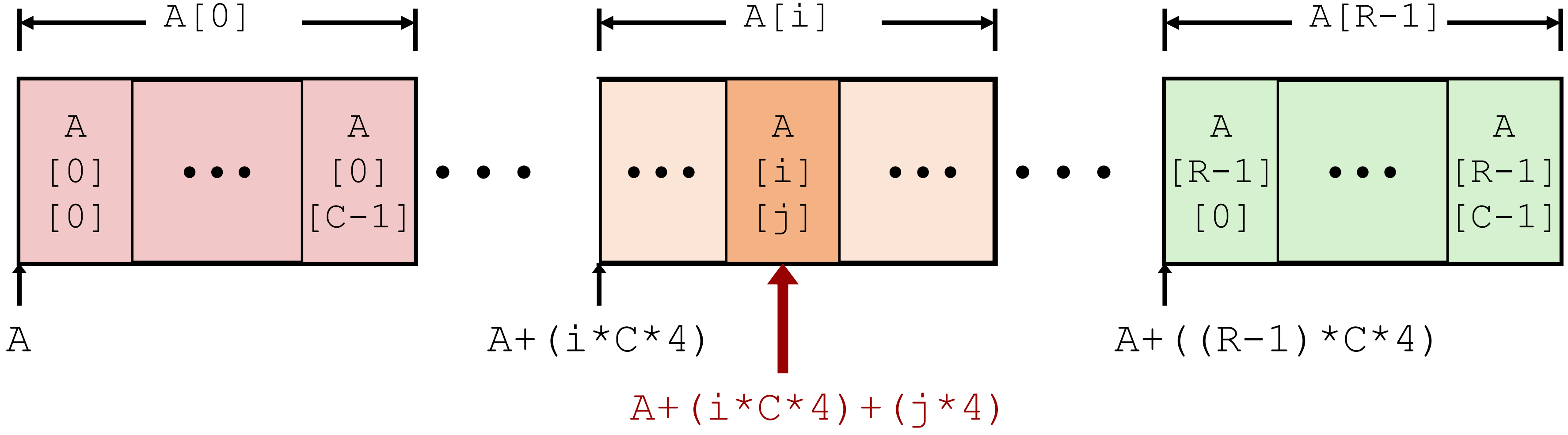
```
  # %rdi = index
  leaq (%rdi,%rdi,4),%rax    # %rax = 5 * index
  leaq ord(,%rax,4),%rax     # %rax = ord + 4*(5*index)
```

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

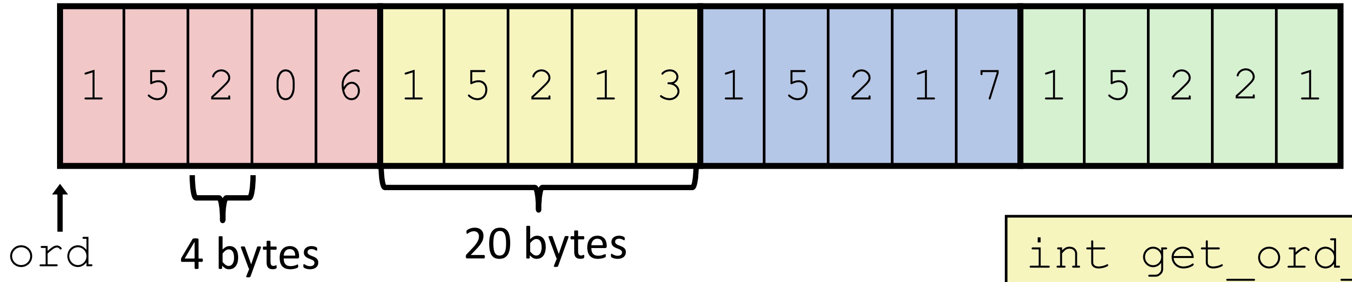
Nested Array Element Access

- Now, let's find the *element* that we want
- 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

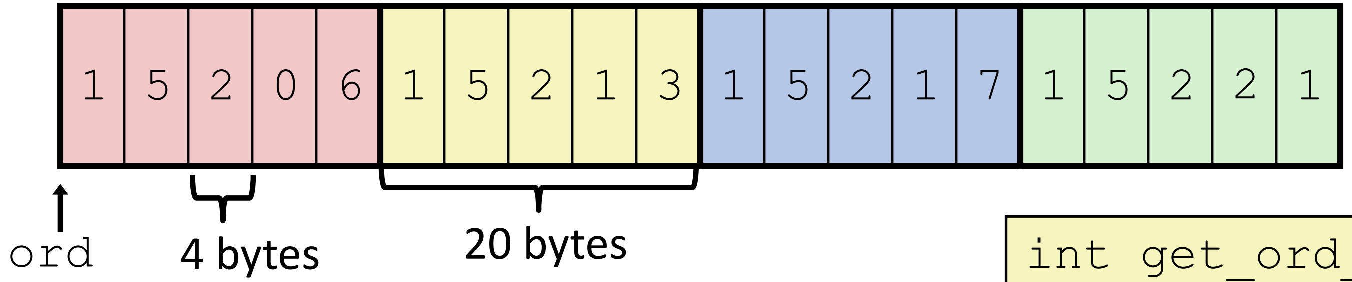



```
int get_ord_digit(size_t index, size_t digit)
{
  return ord[index][digit];
}
```

```
 # %rdi = index
leaq   (%rdi,%rdi,4), %rax    # 5*index
addq   %rax, %rsi             # 5*index + digit
movl   ord(,%rsi,4), %eax     # M[ord + 4*(5*index+digit)]
```

- Array Elements
  - **`ord[index][digit]`** is type **`int`**
  - Address: **`ord + 20*index + 4*digit`** = **`ord + 4*(5*index + digit)`**

Nested Array Element Access Code




```
int get_ord_digit(size_t index, size_t digit)
{
  return ord[index][digit];
}
```

```
 # %rdi = index
leaq   (%rdi,%rdi,4), %rax    # 5*index
addq   %rax, %rsi             # 5*index + digit
movl   ord(,%rsi,4), %eax     # M[ord + 4*(5*index+digit)]
```

- Array Elements
  - **`ord[index][digit]`** is type **`int`**
  - Address: **`ord + 20*index + 4*digit`** = **`ord + 4*(5*index + digit)`**
- QUIZ: what is the address of `ord[2][4]`? `ord+56`

Break + Practice

- Find the addresses (assume array starts at address 0)
  - *A + (i * C * K) + (j* K)*

- int A[16][16]; A[1][3]

- char B[16][16]; B[10][7]

- char* B[10][10]; B[0][2]

Break + Practice

- Find the addresses (assume array starts at address 0)
  - *A + (i * C * K) + (j* K)*

- int A[16][16];   A[1][3]
  - *A + (i*C*K) + (j*K) =   0 + (1 * 16 * 4) + (3 * 4) = 64 + 12 = 76*

- char B[16][16];   B[10][7]

- char* B[10][10];   B[0][2]

Break + Practice

- Find the addresses (assume array starts at address 0)
  - *A + (i * C * K) + (j* K)*

- int A[16][16]; A[1][3]
  - *A + (i*C*K) + (j*K) = 0 + (1 * 16 * 4) + (3 * 4) = 64 + 12 = 76*

- char B[16][16]; B[10][7]
  - *A + (i*C*K) + (j*K) = 0 + (10 * 16 * 1) + (7 * 1) = 160 + 7 = 167*

- char* B[10][10]; B[0][2]

Break + Practice

- Find the addresses (assume array starts at address 0)
  - *A + (i * C * K) + (j* K)*

- int A[16][16]; A[1][3]
  - *A + (i*C*K) + (j*K) = 0 + (1 * 16 * 4) + (3 * 4) = 64 + 12 = 76*

- char B[16][16]; B[10][7]
  - *A + (i*C*K) + (j*K) = 0 + (10 * 16 * 1) + (7 * 1) = 160 + 7 = 167*

- char* B[10][10]; B[0][2]
  - *A + (i*C*K) + (j*K) = 0 + (0 * 10 * 8) + (2 * 8) = 16*

Outline

- Pointers
- One-dimensional Arrays
- Multi-dimensional Arrays
- **Multi-level Arrays**
- Dynamic arrays

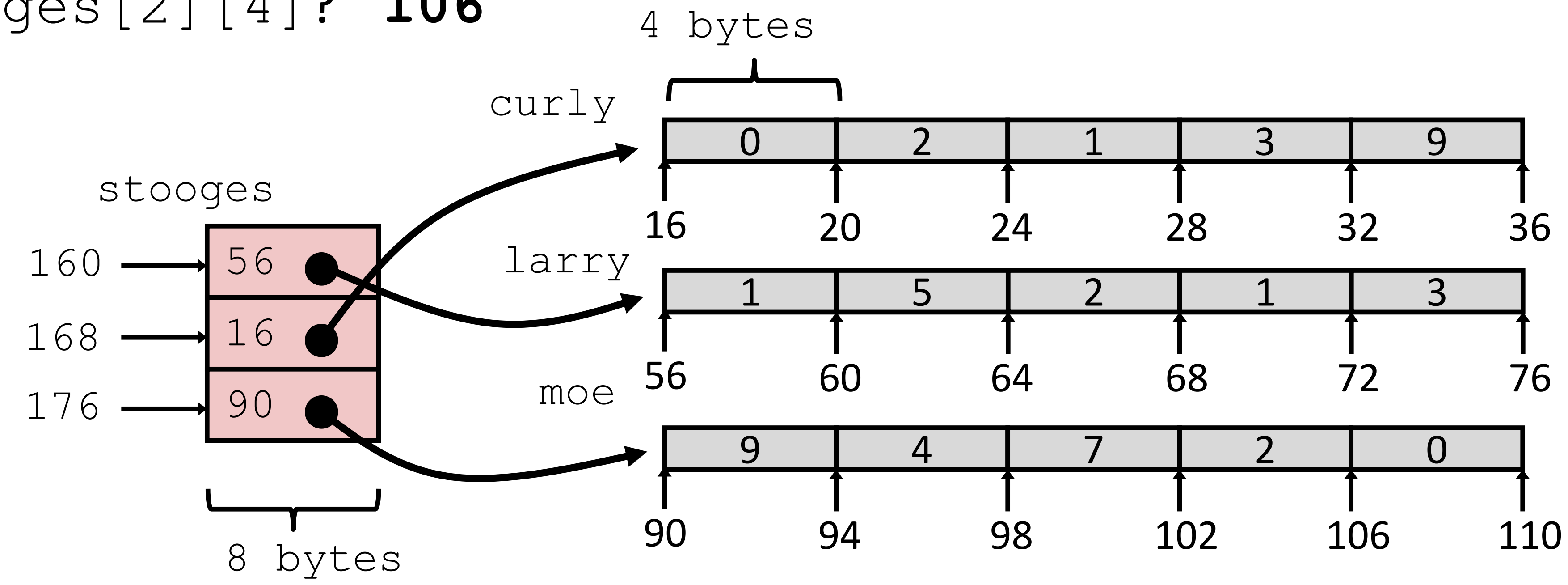
Multi-Level Array Example

```
int larry [5] = { 1, 5, 2, 1, 3 };
int curly [5] = { 0, 2, 1, 3, 9 };
int moe [5]   = { 9, 4, 7, 2, 0 };
```

```
int *stooges[3] =
   {larry, curly, moe};
```

- Variable `stooges` denotes array of 3 elements
- Each element is a pointer (8 bytes)
- Each pointer points to array of `int`s
- `stooges` is of type `int* []`
- `stooges` is of type `int**`

QUIZ: What is the address of `stooges[2][4]`? **106**



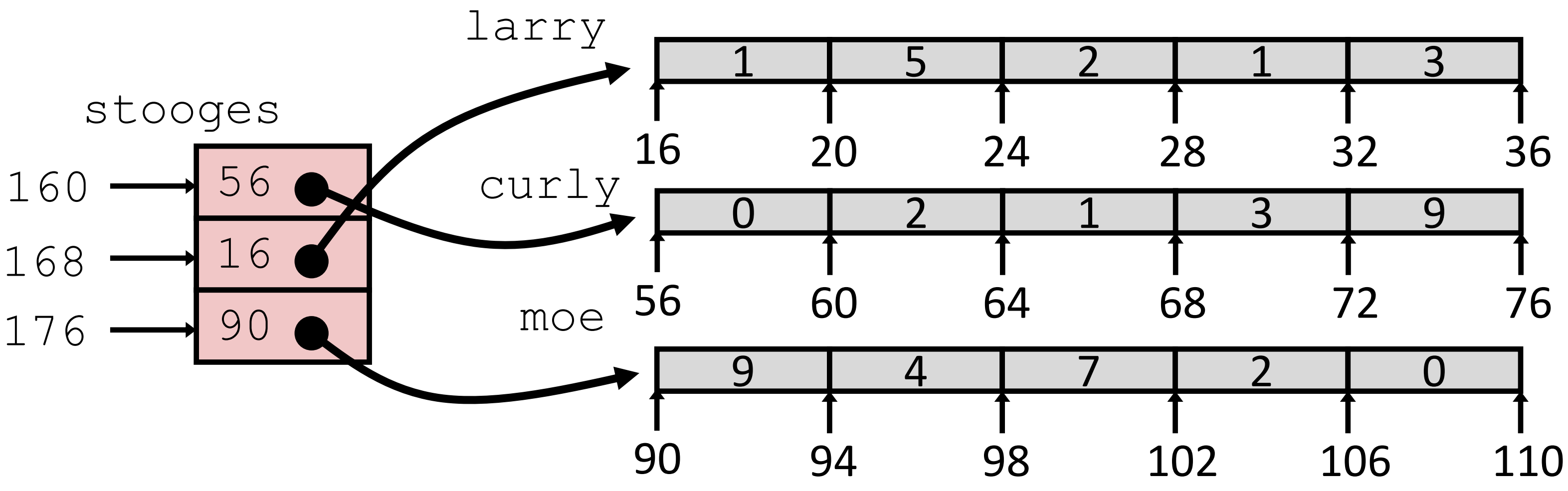
Multi-Level Array Element Access

```
int get_stooge_digit
  (size_t index, size_t digit){
  return stooges[index][digit];
}
```

- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

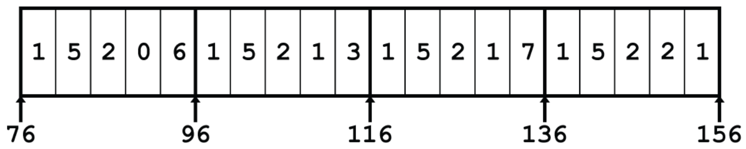
```
salq    $2, %rsi                 # 4*digit
addq    stooges(,%rdi,8), %rsi   # p = stooges[8*index] + 4*digit
movl    (%rsi), %eax             # return *p
ret
```

Element access `Mem[Mem[stooges+8*index] + 4*digit]`

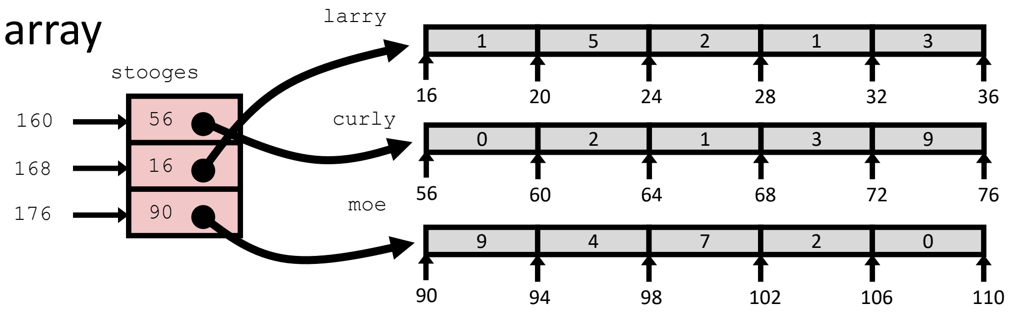


Nested vs. Multi-Level Array Element Accesses

Nested array



Multi-level array




```
int ord [4][5];


int get_ord_digit
  (size_t index, size_t digit){
  return ord[index][digit];
}
```

```
int larry[5], curly[5], moe[5];
int *stooges[3] = {larry, curly, moe};

int get_stooge_digit
  (size_t index, size_t digit){
  return stooges[index][digit];
}
```

Accesses look similar in C, but address computations are very different:

`ord` is of type `int*`

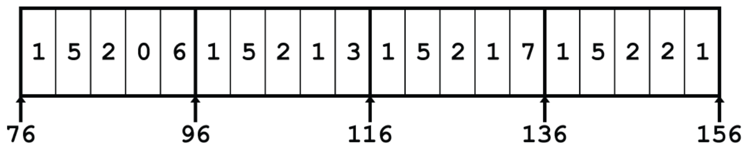
`stooges` is of type `int**`

`Mem[ord+(20*index)+(4*digit)]`

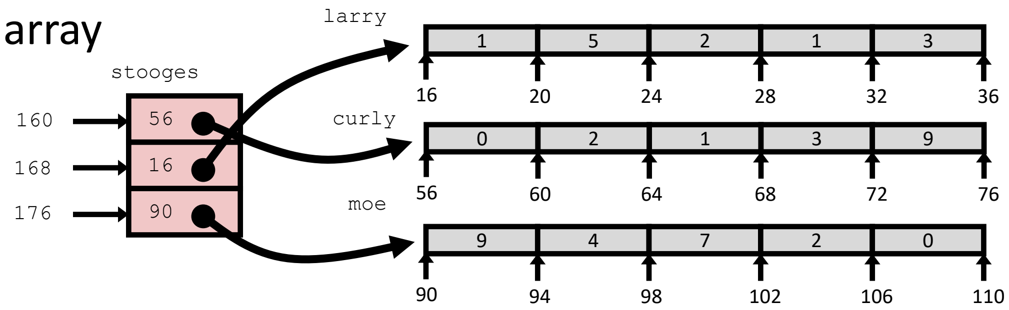
`Mem[Mem[stooges+(8*index)]+(4*digit)]`

Nested versus Multi-Level Arrays

Nested array



Multi-level array



`Mem[ord+(20*index)+(4*digit)]`

`Mem[Mem[stooges+(8*index)]+(4*digit)]`

- Strengths
  - Fast element access
    - Single memory access
  - Efficient memory usage
    - Stored in contiguous memory
- Limitations
  - Requires fixed size rows
  - Large memory usage
    - All rows need to be allocated

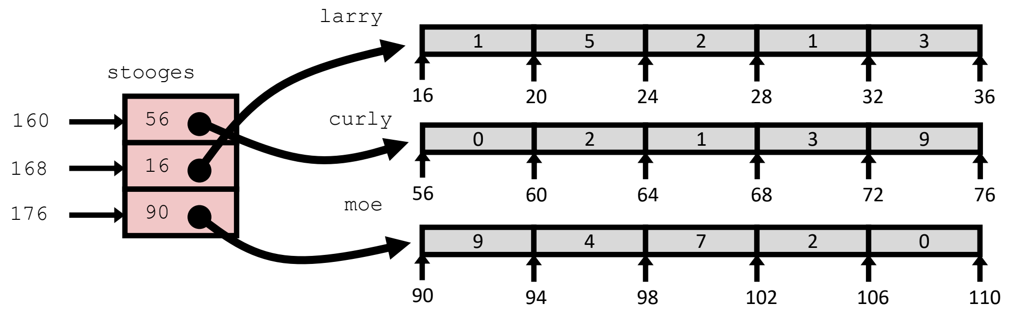
- Strengths
  - Rows may be of different size
  - Rows could even be different types
    - First array would store `void*`
- Limitations
  - Slow element access
    - Two memory references
  - Memory fragmentation
    - Many small chunks allocated

Outline

- Pointers
- One-dimensional Arrays
- Multi-dimensional Arrays
- Multi-level Arrays
- **Dynamic arrays**

Dynamic Multi-dimensional arrays – multi-level

- Multi-level is one way to make them



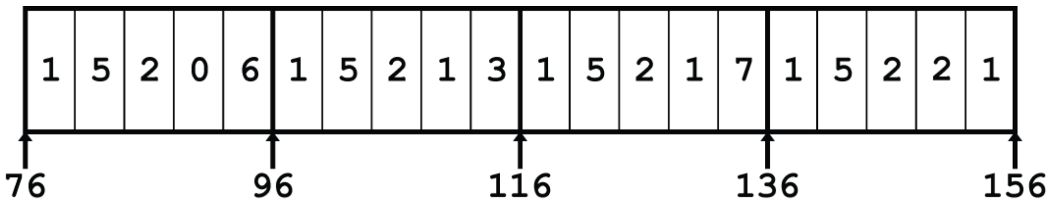

```
int** array_2d = (int**)malloc(rows * sizeof(int*));

for (int i=0; i<rows; i++) {
  array_2d[i] = (int*)malloc(cols * sizeof(int));
}

array_2d[2][4] = 0;
```

Dynamic multi-dimensional arrays - nested

- Nested works as well
  - Handle nested manually
  - Make sure you get it right!




```
int* array_2d = (int*)malloc(rows * cols * sizeof(int));

array_2d[2*cols + 4] = 0; // array_2d[2][4]
```

Nested arrays – static versus dynamic

```
void testarray(void) {
  volatile int A[16][16];
  A[2][4] = 0;

  volatile int* B =
    (int*)malloc(16*16*sizeof(int));
  B[2*16 + 4] = 0;
}
```

```
testarray():
 sub     $0x408,%rsp
 movl    $0x0,0x90(%rsp)
 mov     $0x400,%edi
 call    400480 <malloc@plt>
 movl    $0x0,0x90(%rax)
 add     $0x408,%rsp
 ret
```

Nested arrays – static versus dynamic

```
void testarray(void) {
  volatile int A[16][16];
  A[2][4] = 0;

  volatile int* B =
    (int*)malloc(16*16*sizeof(int));
  B[2*16 + 4] = 0;
}
```

```
testarray():
 sub     $0x408,%rsp
 movl    $0x0,0x90(%rsp)
 mov     $0x400,%edi
 call    400480 <malloc@plt>
 movl    $0x0,0x90(%rax)
 add     $0x408,%rsp
 ret
```

Nested arrays – static versus dynamic

```
void testarray(void) {
  volatile int A[16][16];
  A[2][4] = 0;

  volatile int* B =
    (int*)malloc(16*16*sizeof(int));
  B[2*16 + 4] = 0;
}
```

```
testarray():
 sub     $0x408,%rsp
 movl    $0x0,0x90(%rsp)
 mov     $0x400,%edi
 call    400480 <malloc@plt>
 movl    $0x0,0x90(%rax)
 add     $0x408,%rsp
 ret
```

Nested arrays – static versus dynamic

```
void testarray(void) {
  volatile int A[16][16];
  A[2][4] = 0;

  volatile int* B =
    (int*)malloc(16*16*sizeof(int));
  B[2*16 + 4] = 0;
}
```

```
testarray():
 sub     $0x408,%rsp
 movl    $0x0,0x90(%rsp)
 mov     $0x400,%edi
 call    400480 <malloc@plt>
 movl    $0x0,0x90(%rax)
 add     $0x408,%rsp
 ret
```

Outline

- Pointers
- One-dimensional Arrays
- Multi-dimensional Arrays
- Multi-level Arrays
- Dynamic arrays