Lecture 09
Pointers, Arrays, and Structs

CS213 – Intro to Computer Systems

Branden Ghena – Winter 2024

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

Northwestern

Administrivia

- Bomb Lab
  - Due next week Tuesday (Feb. 13) by end-of-day
  - Start now if you haven't yet!!
  - Remember, it's tricky, but not trying to trick you
    - You can trust function names to roughly do what they say
  - Piazza has a pinned post linking to GDB tutorials

- Homework 3 should be out later today
- Attack Lab starts this week Thursday

Drop deadline

- Drop deadline is this Friday by end-of-day

- You already have all the grades you'll get by then
  - Homework 2 will likely not be graded in time

- If you're concerned and want to chat, please make a private post on Piazza and I'll find time to meet with you.

Today's Goals

- Wrap up x86-64 assembly!
  - Although assembly details will remain important

- Understand C arrays
  - Single and multi-dimensional
  - And how they translate into assembly code

- Discuss how structures are accessed
  - Memory layout details including alignment

- Bonus material on Dynamic Arrays and Unions

Outline

- **Pointers**

- One-dimensional Arrays
- Multi-dimensional Arrays
- Multi-level Arrays

- Struct Layout
- Struct Padding and Alignment

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`

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

```
int* a  = &v;

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

- Structs

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

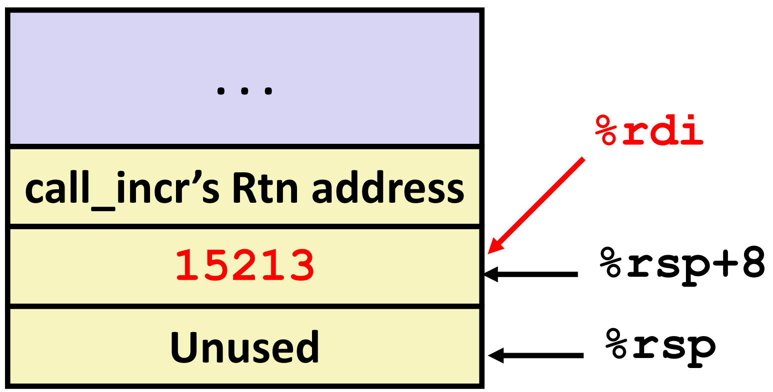
Example pointer code: calling `incr`

```
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`

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

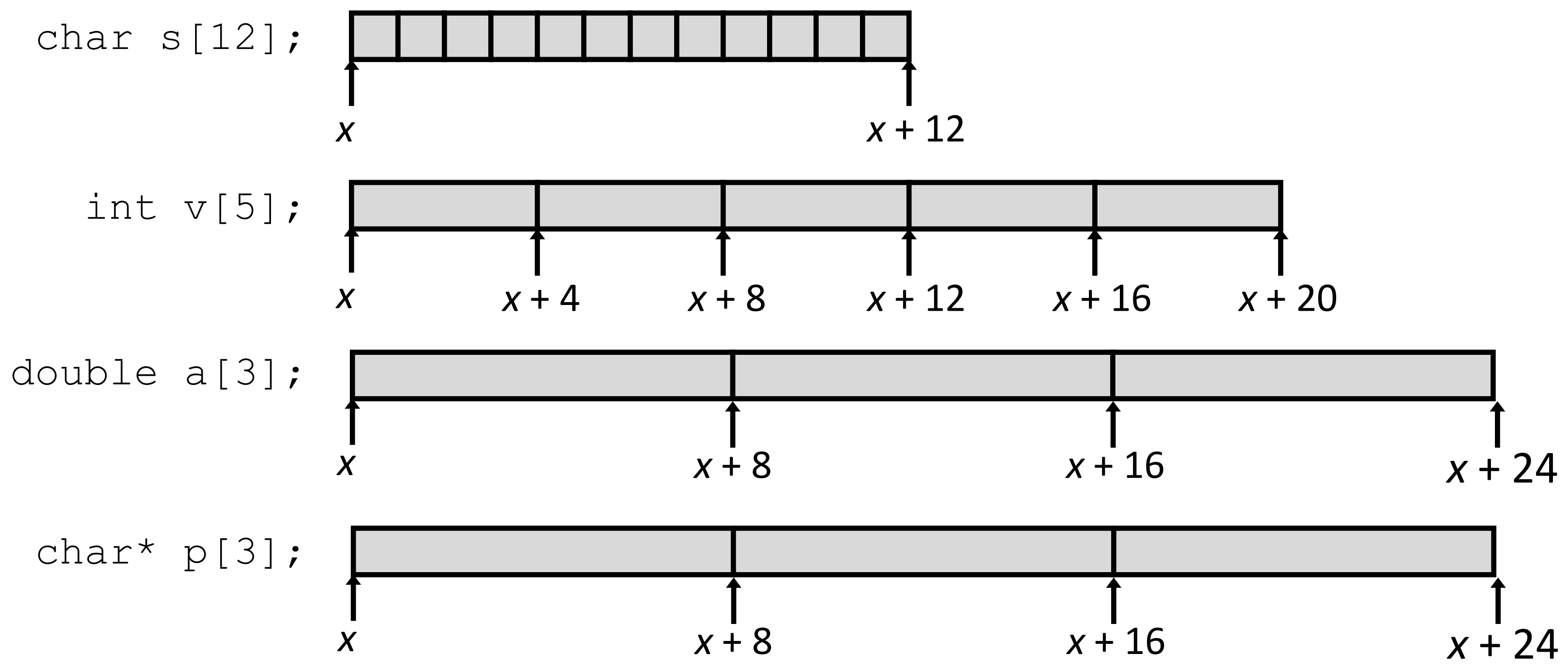
- Struct Layout
- Struct Padding and Alignment

One-Dimensional Array Allocation

- Basic Principle

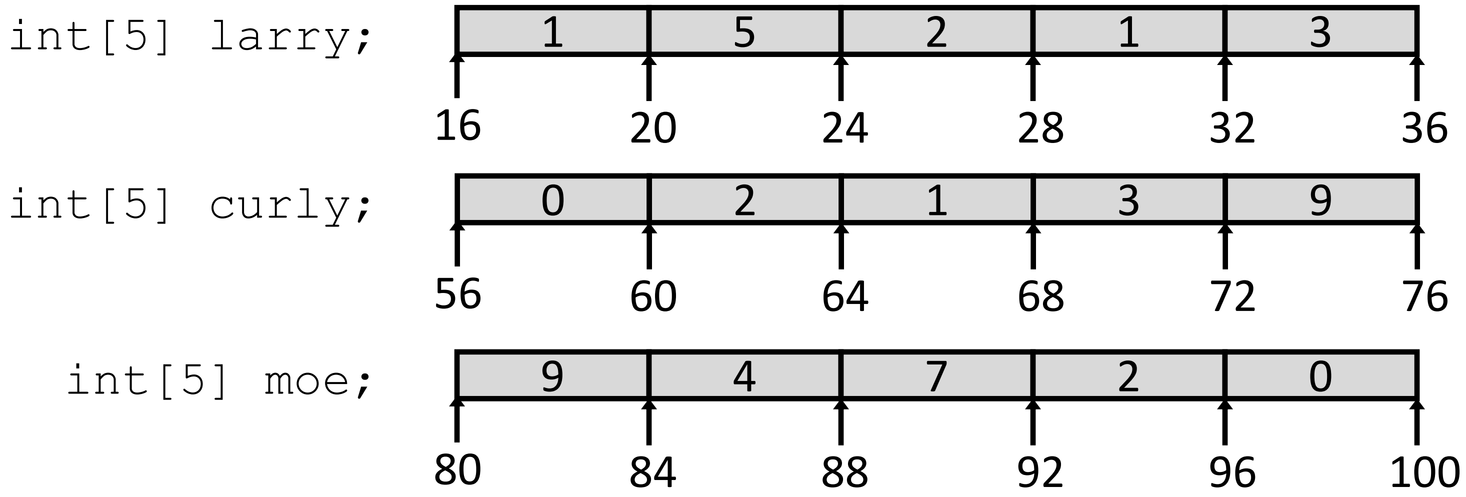
  T **`A[`***L***`];  // e.g., int A[4];`**

  - 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[]`!

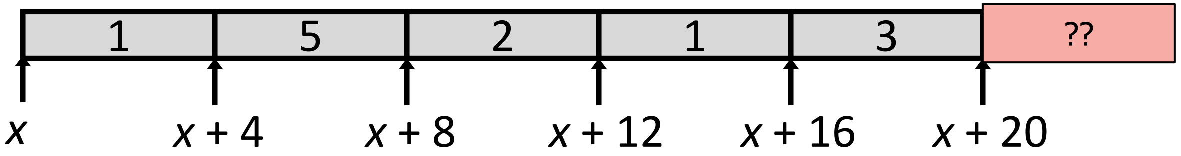
Array Access and Pointer Arithmetic

- 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 number of elements, but in assembly we count number of bytes!

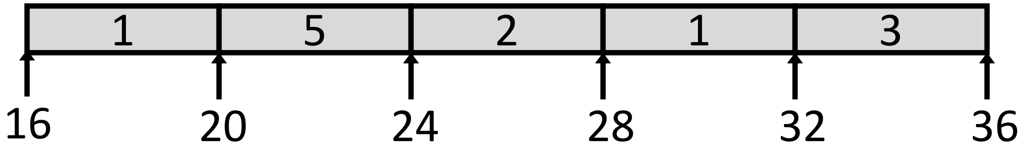
`int val[5];`



• 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)`

- This is why memory accesses have a scale! `D(Rb, Ri, s)`
  - Scale 1, 2, 4, or 8 -> type sizes

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) #Source: 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) #Source: z[i]++ (int z[])
```

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

Quiz + Break

```
z -> %rdi

i -> %rax

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

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

Quiz + Break

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

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

Quiz + Break

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

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

Quiz + Break

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

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

Outline

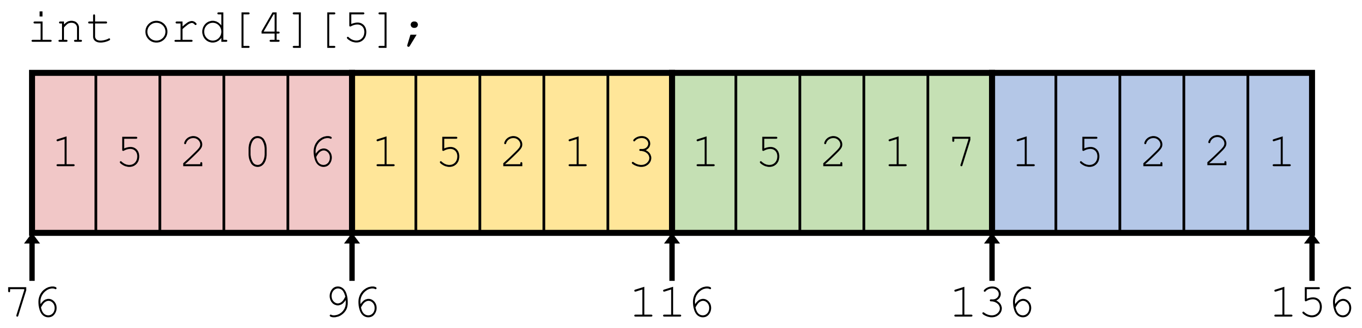
- Pointers

- One-dimensional Arrays
- **Multi-dimensional Arrays**
- Multi-level Arrays

- Struct Layout
- Struct Padding and Alignment

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

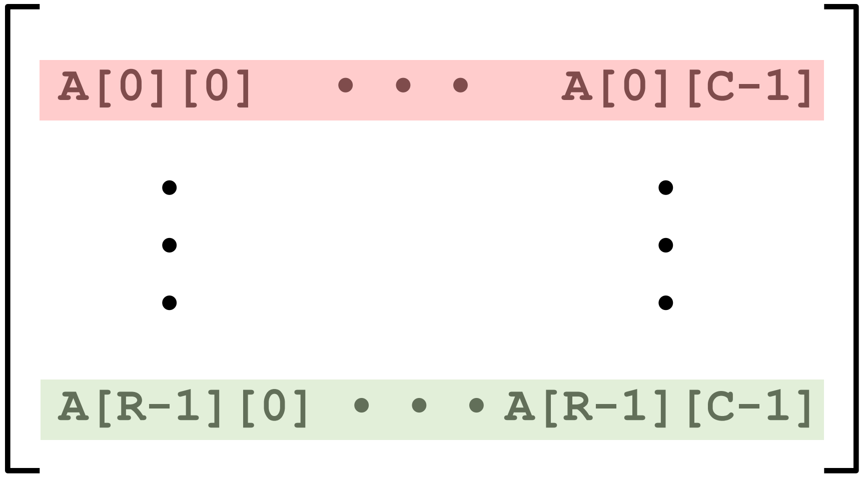
- "Row-Major" ordering of all elements is guaranteed
  - Entire row (all columns in it) will be placed in memory before the next row starts

Multidimensional (Nested) Arrays

- Declaration

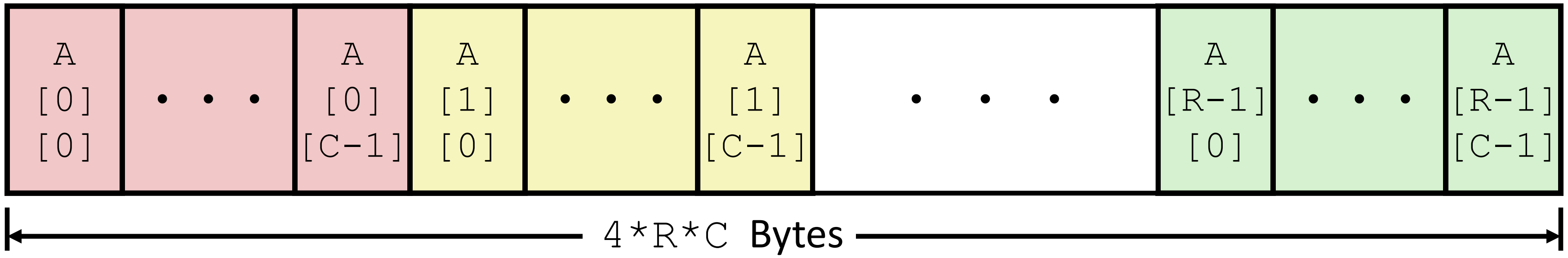
    T **A**[*R*][*C*];

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



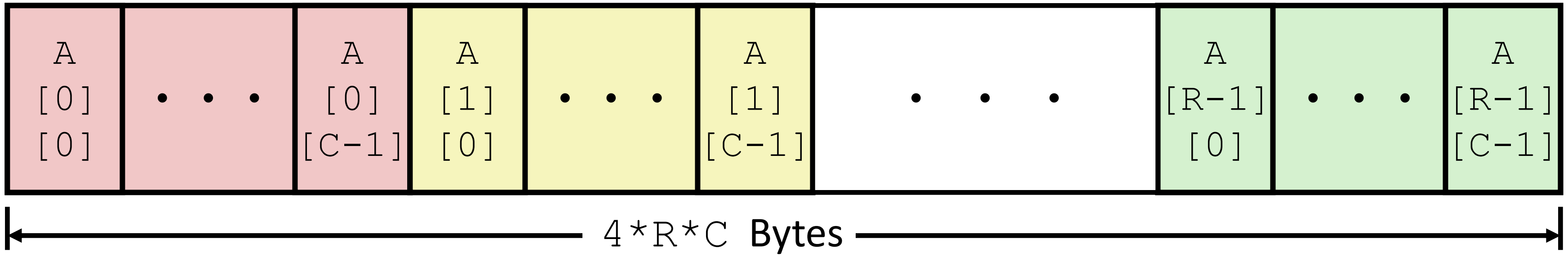
- Types
    - *What is A?* *T [R] [C] -> T***
    - *What is A[i][j]?* *T*

- Arrangement
    - Row-Major Ordering



`int A[R][C];`

Accessing items in the array




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

1. Figure out which row you want to access
   Skip over previous rows

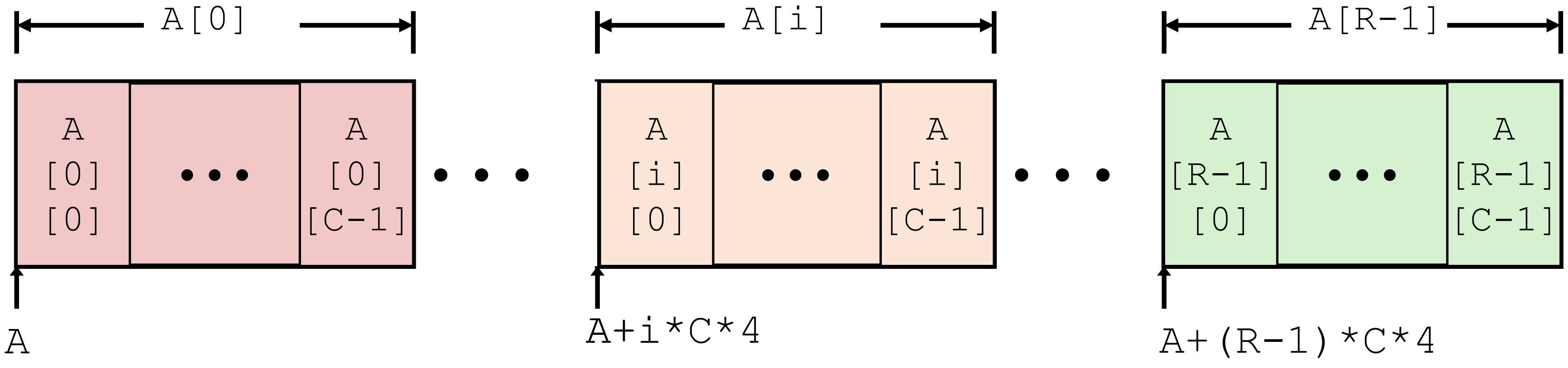
2. Figure out which column you want to access in that row
   Skip over previous columns in that row

Nested Array **Row** Access

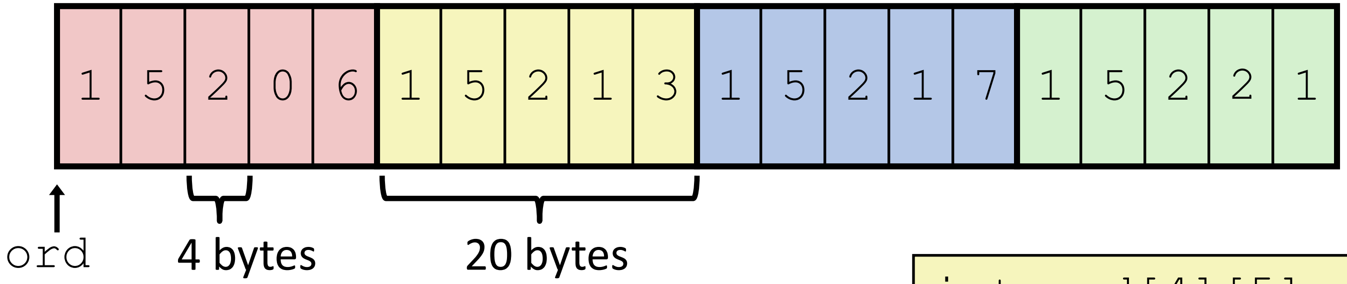
- To figure out how to get the element we want
  - Let's first figure out how to get the **row** we want (its starting address)
- Row Vectors
  - **`A[i]`** (row) is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Nested array formula: **`A +`** $i * (C * K)$

Only gets you to the right ROW

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



Nested Array **Row** Access Code

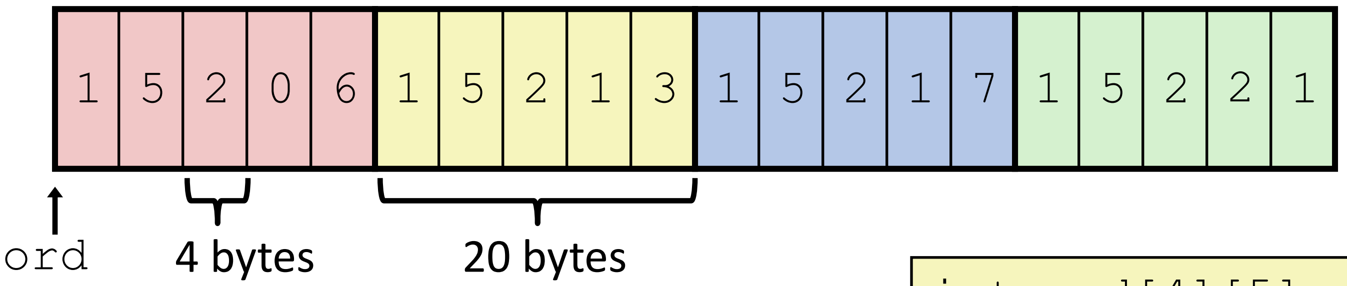



```
int* get_ord_row(size_t index)
{
  return ord[index];
}
```

```
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



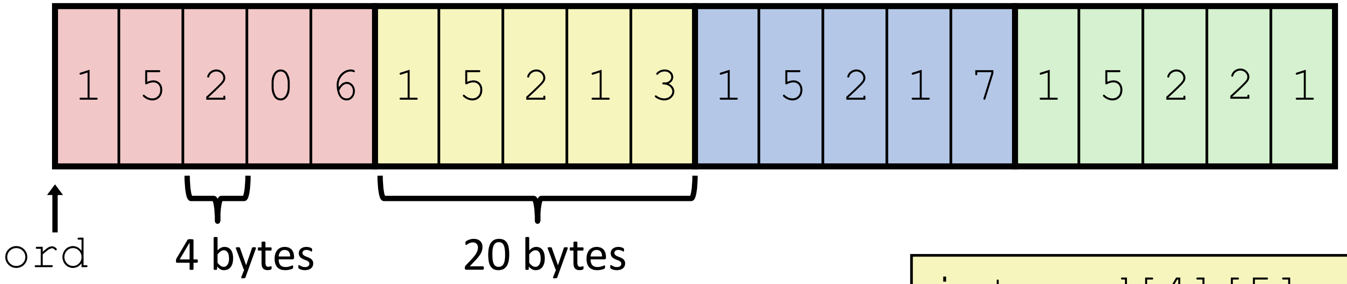

```
int* get_ord_row(size_t index)
{
  return ord[index];
}
```

```
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 }};
```

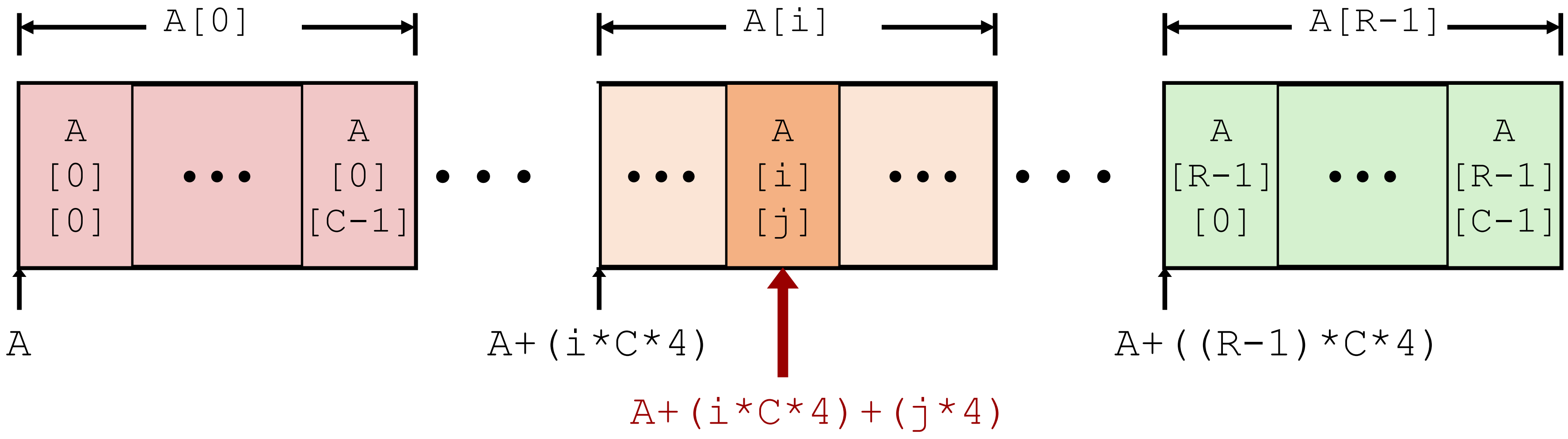
```
  # %rdi = index
   leaq (%rdi,%rdi,4),%rax    # %rax = 5 * index
   leaq ord(,%rax,4),%rax     # %rax = ord + 4*(5*index)
```

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

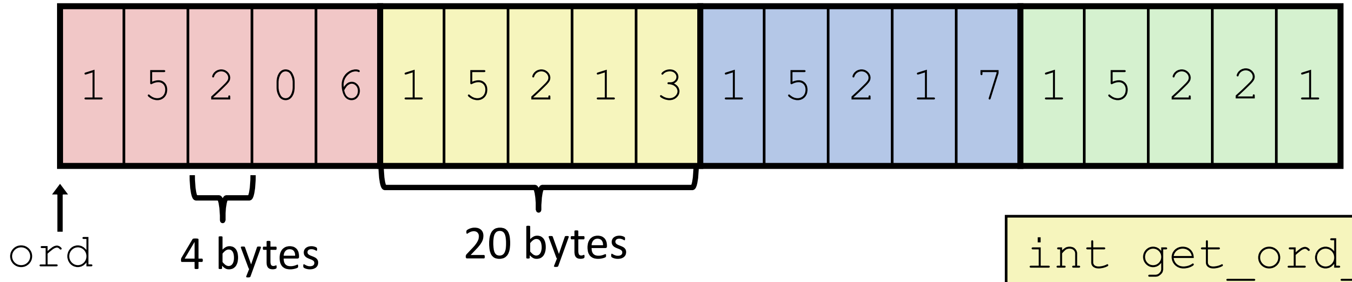
Nested Array **Element** Access

- Now, let's find the *element* that we want
- Array Elements
  - **`A[i][j]`** is element which requires K bytes, within nested arrays of C elements
  - Address $A + i * (C * K) + j * K = A + (i * C + j) * K$ ← Gets you the exact element

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



Nested Array **Element** Access Code

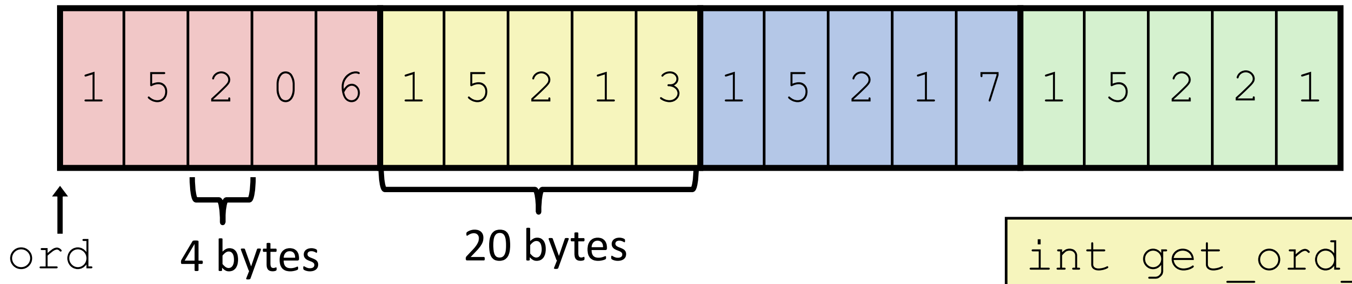



```
int get_ord_digit(size_t index, size_t digit)
{
  return ord[index][digit];
}
```

```
 # %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**

- Struct Layout
- Struct Padding and Alignment

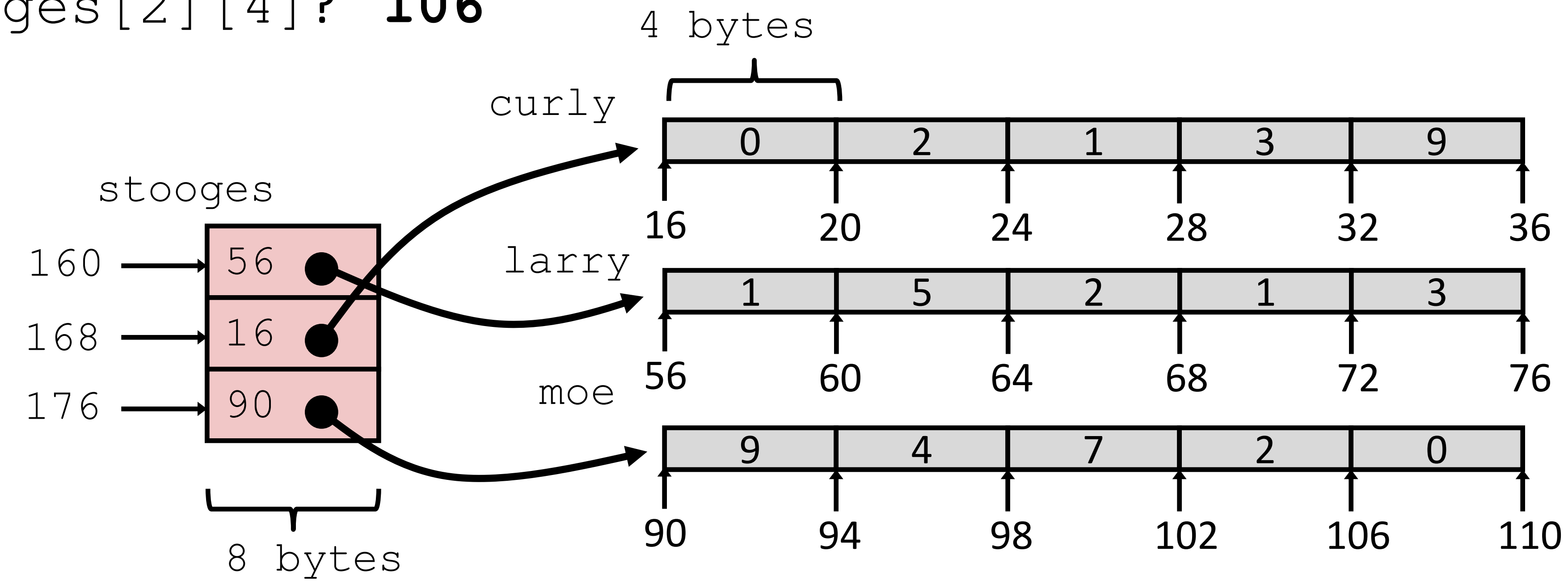
Multi-Level Array Example

```
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**



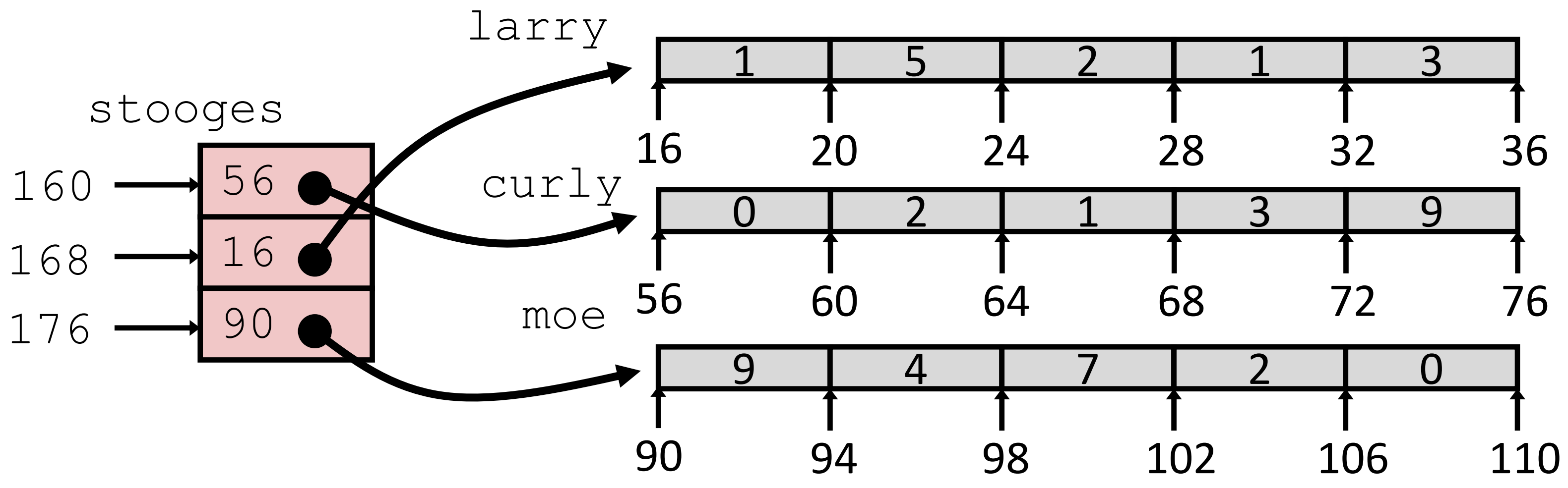
Multi-Level Array Element Access

```
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

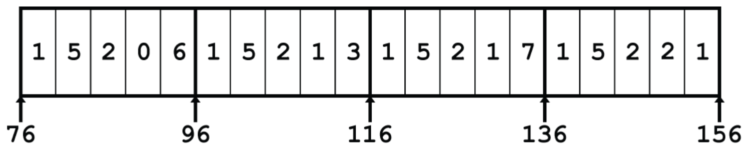
```
salq    $2, %rsi                  # 4*digit
addq    stooges(,%rdi,8), %rsi    # p = stooges[8*index] + 4*digit
movl    (%rsi), %eax              # return *p
ret
```

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

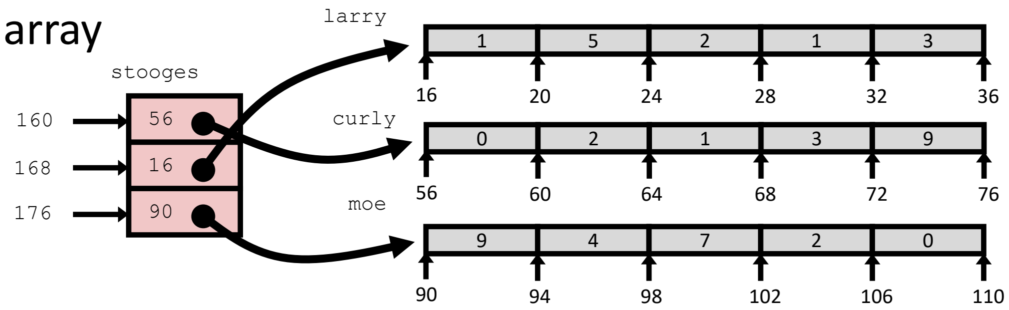


Nested vs. Multi-Level Array Element Accesses

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 sort of like `int*`

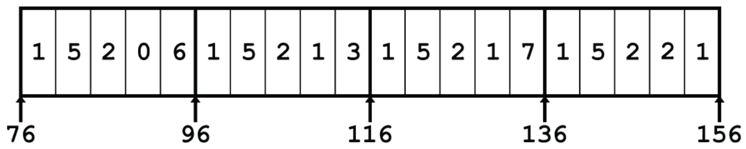
`stooges` is definitely `int**`

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

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

Nested versus Multi-Level Arrays

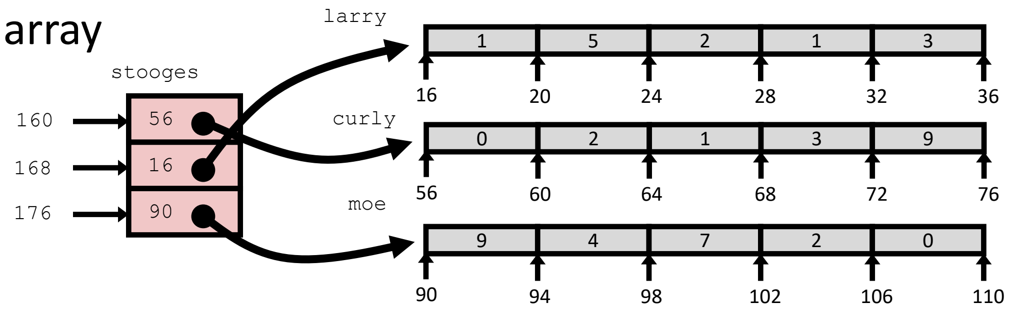
Nested array



`Mem[ord+(20*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

Multi-level array



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

- 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

Break

- That was a lot of math
- And there’s more math to come

- So let’s take a mental break to reset

- To help, I have provided you with a distraction:

Outline

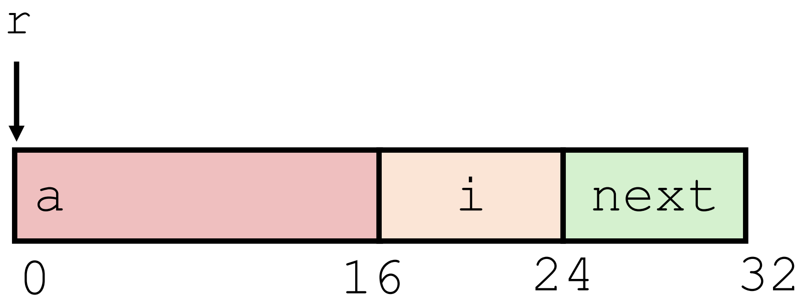
- Pointers

- One-dimensional Arrays
- Multi-dimensional Arrays
- Multi-level Arrays

- **Struct Layout**
- Struct Padding and Alignment

Structure representation in C

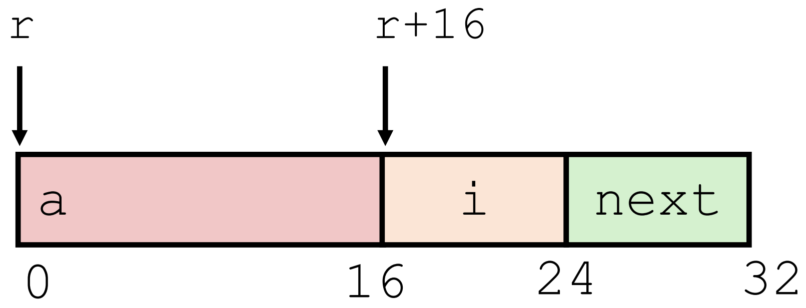
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Structure represented as chunk of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration order
  - Even if another ordering could yield a more compact representation
  - (We’ll see how that could happen in a bit)
- Compiler determines overall size + positions of fields
  - Looking at memory, no way to tell it’s a struct (like arrays); just bytes
  - It’s all in how the code treats that region of memory!

Structure access

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Accessing Structure Member
  - Pointer **r** indicates first byte of structure
  - Access member with offsets
  - Offset of each structure member determined at compile time
    - Another use for Displacement in memory addressing!

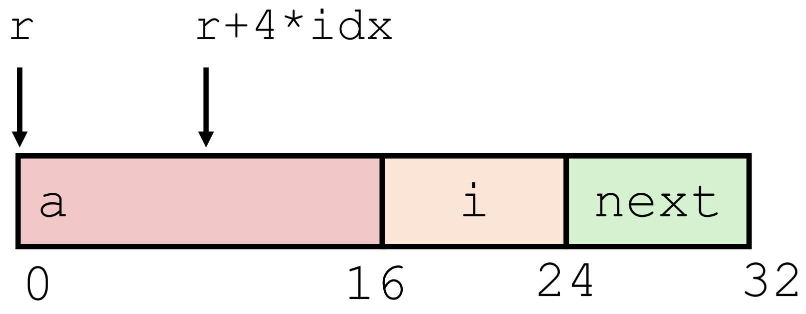
```
size_t get_i(struct rec *r)
{
  return r->i;
}
```

```
  # r in %rdi
  movq 16(%rdi), %rax
  ret
```

`r` is a pointer to a struct.
Dereference the ponter, then get the `i` field of the struct.

Array within a struct

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- Same as before; just need to also index in the array
  - Pointer **`r`** indicates first byte of structure
    - Offset of each structure member determined at compile time
    - Offset into array determined based on index and type
  - Compute as **`*(structAddr + offset + K*index);`**
    - Uses full addressing mode!

```
int get_a (struct rec *r,
           size_t idx)
{
  return r->a[idx];
}
```

```
  # r in %rdi
  # idx in %rsi
  movq 0(%rdi,%rsi,4), %rax
  ret
```

Structure Access Practice 1

Arguments:
1) %rdi
2) %rsi
3) %rdx
4) %rcx
5) %r8
6) %r9

```
struct rec {
  int j;
  int i;
  int a[2];
  struct rec *n;
};
```

```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

```
  movl  %esi , 4(%rdi)
  ret
```

Structure Access Practice 2

Arguments:
1) %rdi
2) %rsi
3) %rdx
4) %rcx
5) %r8
6) %r9

```
struct rec {
  int j;
  int i;
  int a[2];
  struct rec *n;
};
```

```
void
set_i(struct rec *r,
      int val)
{
  r->a[1] = val;
}
```

```
movl %esi , 12(%rdi)
ret
```

Structure Access Practice 3

Arguments:

```
1) %rdi
2) %rsi
3) %rdx
4) %rcx
5) %r8
6) %r9
```

```
struct rec {
  int j;
  int i;
  int a[2];
  struct rec *n;
};
```

```
void
set_i(struct rec *r,
      int val,
      int index)
{
  r->a[index] = val;
}
```

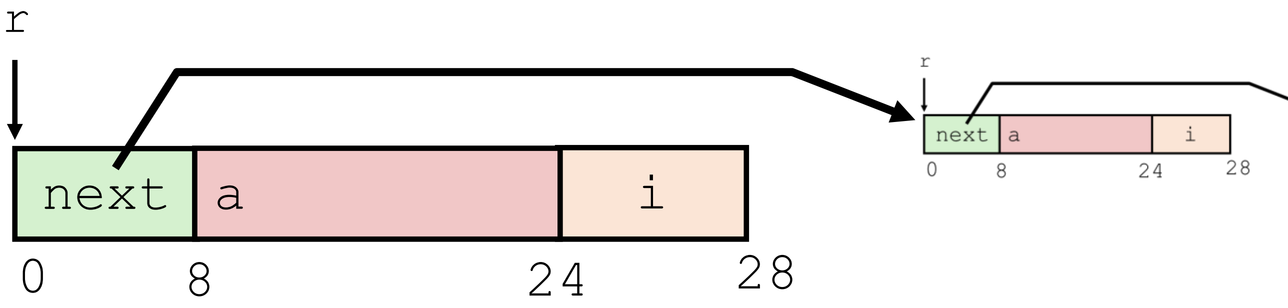
```
movl  %esi , 8(%rdi, %rdx, 4)
ret
```

Following Linked List

By convention, null `next` pointer indicates end of list

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```



Register	Value
%rdi	**r**
%rsi	**val**

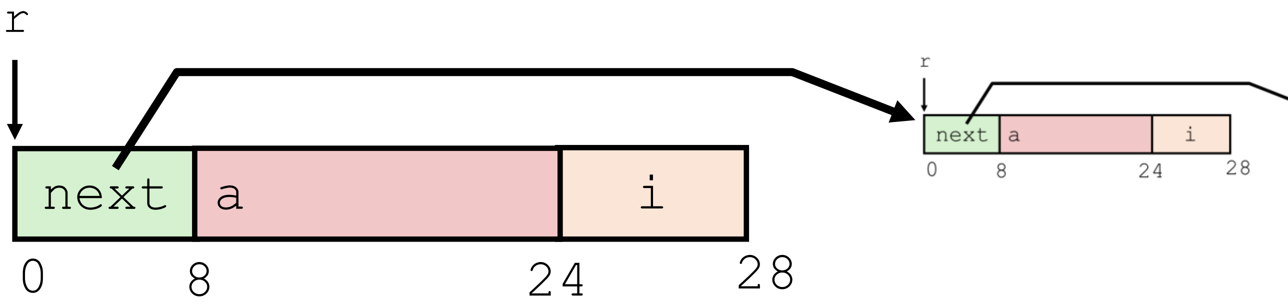
```
.L11:                          # loop:
  movslq  24(%rdi), %rax       #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4) #   M[r+8+4*i] = val
  movq    (%rdi), %rdi         #   r = M[r]
  testq   %rdi, %rdi           #   Test r
  jne     .L11                 #   if !=0 goto loop
```

Following Linked List

By convention, null `next` pointer indicates end of list

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```



Register	Value
%rdi	r
%rsi	val

Load `i`

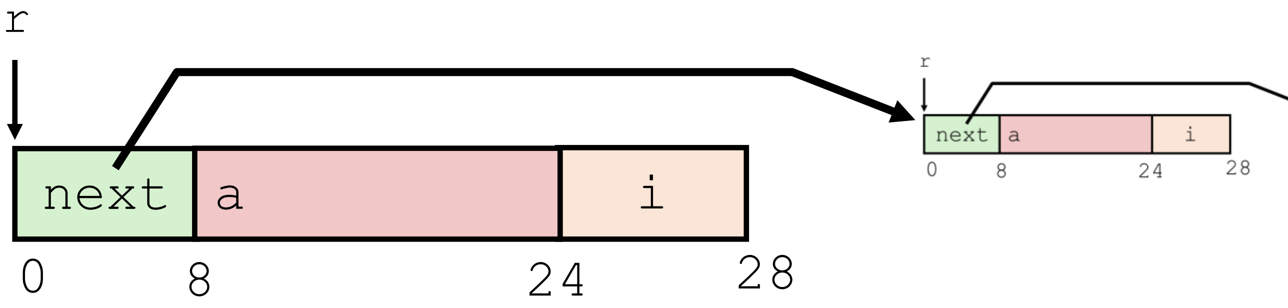
```
.L11:                          # loop:
  movslq  24(%rdi), %rax       #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4) #   M[r+8+4*i] = val
  movq    (%rdi), %rdi         #   r = M[r]
  testq   %rdi, %rdi           #   Test r
  jne     .L11                 #   if !=0 goto loop
```

Following Linked List

By convention, null `next` pointer indicates end of list

```
struct rec {
    struct rec *next;
    int a[4];
    int i;
}; // DIFFERENT ORDER!
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```



Register	Value
%rdi	**r**
%rsi	**val**

Write `val` into `r->a[i]`

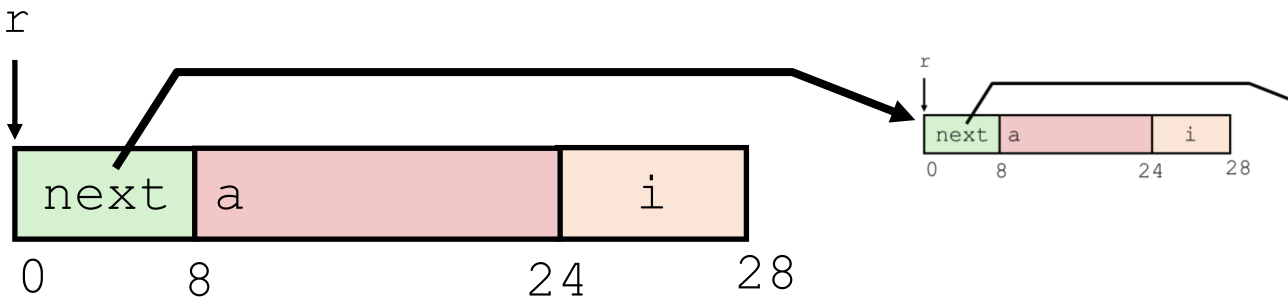
```
.L11:                              # loop:
  movslq  24(%rdi), %rax           #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4)     #   M[r+8+4*i] = val
  movq    (%rdi), %rdi             #   r = M[r]
  testq   %rdi, %rdi               #   Test r
  jne     .L11                     #   if !=0 goto loop
```

Following Linked List

By convention, null `next` pointer indicates end of list

```
struct rec {
     struct rec *next;
     int a[4];
     int i;
}; // DIFFERENT ORDER!
```

```
void set_val
   (struct rec *r, int val)
{
   while (r) {
     int i = r->i;
     r->a[i] = val;
     r = r->next;
   }
}
```



Register	Value
%rdi	**r**
%rsi	**val**

Move to next node in list →

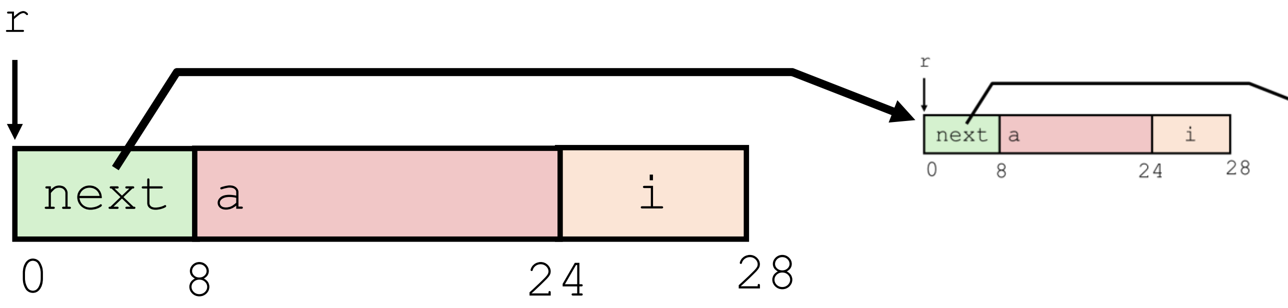
```
.L11:                               # loop:
  movslq  24(%rdi), %rax            #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4)      #   M[r+8+4*i] = val
  movq    (%rdi), %rdi              #   r = M[r]
  testq   %rdi, %rdi                #   Test r
  jne     .L11                      #   if !=0 goto loop
```

Following Linked List

By convention, null `next` pointer indicates end of list

```
struct rec {
     struct rec *next;
     int a[4];
     int i;
}; // DIFFERENT ORDER!
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```



Register	Value
%rdi	**r**
%rsi	**val**

```
.L11:                             # loop:
  movslq  24(%rdi), %rax          #   i = M[r+24]
  movl    %esi, 8(%rdi,%rax,4)    #   M[r+8+4*i] = val
  movq    (%rdi), %rdi            #   r = M[r]
  testq   %rdi, %rdi              #   Test r
  jne     .L11                    #   if !=0 goto loop
```

`NULL` check

Outline

- Pointers

- One-dimensional Arrays
- Multi-dimensional Arrays
- Multi-level Arrays

- Struct Layout
- **Struct Padding and Alignment**

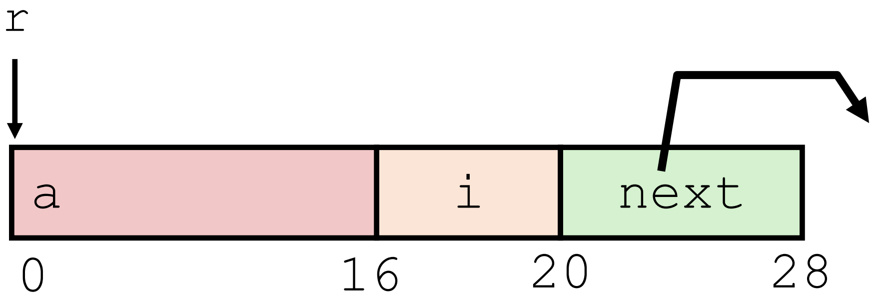
Alignment

- Aligned data
  - Primitive data type requires K bytes
  - Address must typically be a multiple of K (e.g., 1,2,4 or 8)
    - an address that is a multiple of K is called “K-byte aligned”
- Required on some machines; recommended on x86-64
  - But not doing it will really slow down your program
- For example, pointers need 8-byte alignment
  - Multiple of 8 is fine, non-multiple of 8 is bad

Problem: reordering can lead to different layouts

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```



Register	Value
%rdi	r
%rsi	val

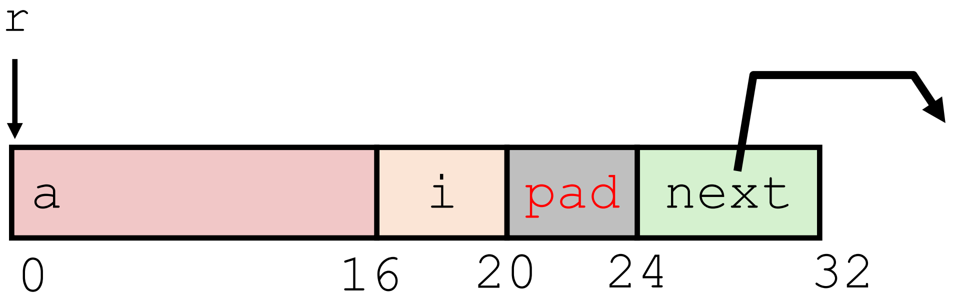
```
.L11:                              # loop:
  movslq  16(%rdi), %rax           #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)      #   M[r+4*i] = val
  movq    ??(%rdi), %rdi           #   r = M[r+??]
  testq   %rdi, %rdi               #   Test r
  jne     .L11                     #   if !=0 goto loop
```

Alignment is wrong!!

Can't load 8 bytes from address 20 efficiently

Padding is added to struct to preserve *alignment*

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```




```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

Register	Value
%rdi	r
%rsi	val

```
.L11:                             # loop:
  movslq  16(%rdi), %rax          #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)     #   M[r+4*i] = val
  movq    24(%rdi), %rdi          #   r = M[r+24]
  testq   %rdi, %rdi              #   Test r
  jne     .L11                    #   if !=0 goto loop
```

The why and how of alignment

- Motivation for aligning data
  - Inefficient to load or store values that span quad word boundaries

  - Hardware is really good at loading, e.g., 8 bytes at address 16, or 24, or 32
    - If you want 8 bytes at address 12, may need two memory reads. Oops…
    - Some unaligned accesses may even crash your code

- Secondary motivations
  - Having one value spanning 2 cache lines = two cache accesses per access
  - Virtual memory very tricky when a datum spans 2 pages
  - See upcoming lectures on caches and virtual memory

- The compiler manages alignment
  - Inserts gaps in structure to ensure correct alignment of fields
  - All variables need alignment, so the stack pointer is aligned too!

Specific Cases of Alignment (x86-64, Linux)

- 1 byte: **`char`**
  - 1-byte aligned (no restrictions on address)

- 2 bytes: **`short`**
  - 2-byte aligned (lowest 1 bit of address must be **`0`**)

- 4 bytes: **`int`**, **`float`**
  - 4-byte aligned (lowest 2 bits of address must be **`00`**)

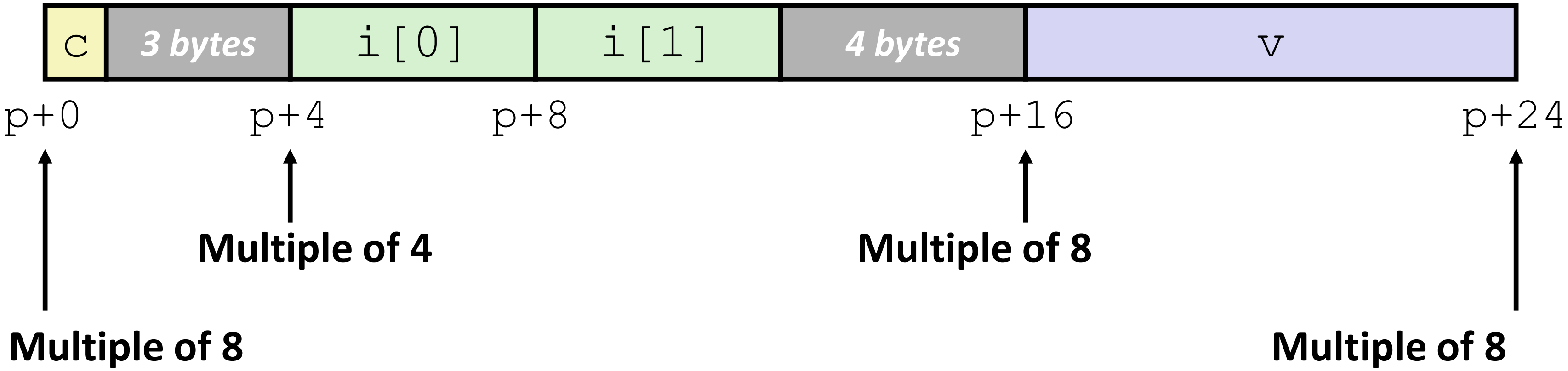
- 8 bytes: **`long`**, **`long long`**, **`double`**, **`char*`** (any pointer)
  - 8-byte aligned (lowest 3 bits of address must be **`000`**)

- 16 bytes: **`long double`**
  - 16-byte aligned (lowest 3 bits of address must be **`0000`**)
  - Max possible alignment requirement on x86-64
    - This is where the "stack moves by 16s" rule comes from

Satisfying Alignment within Structures

- Within structure
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement **`K`**
    - Where **`K`** = Largest alignment of any element
  - Initial address & structure length must be multiples of **`K`**
- Example:
  - K = 8, due to **`double`** element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

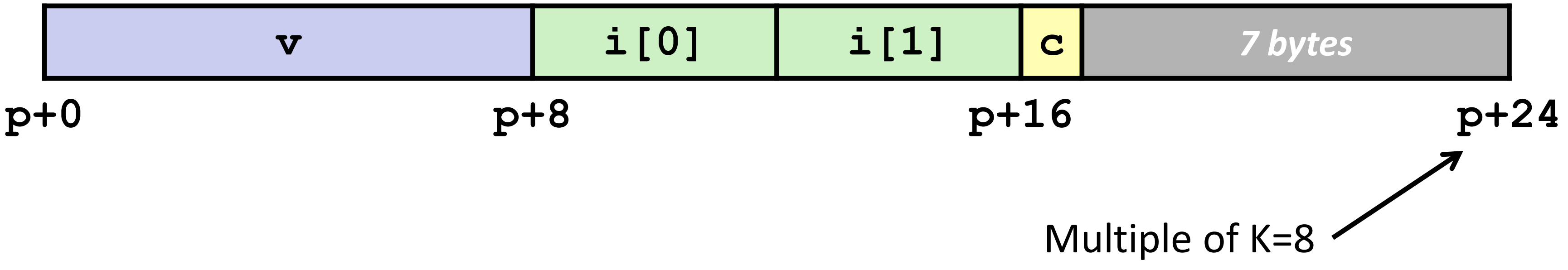


Meeting Overall Alignment Requirement

- Entire struct must be a multiple of its largest element

- For largest alignment requirement K
- Overall structure must be multiple of K
  - Trailing padding

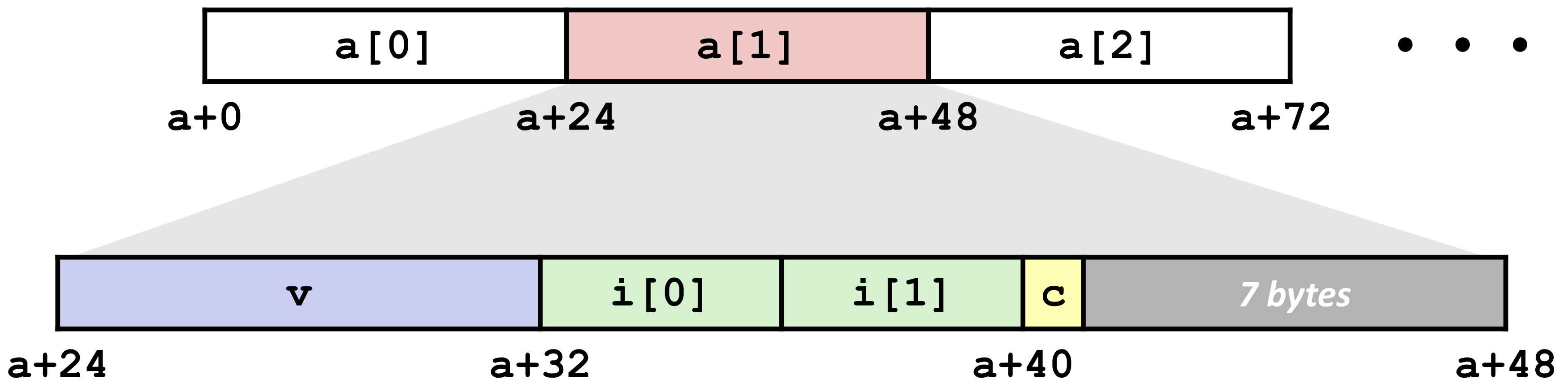
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

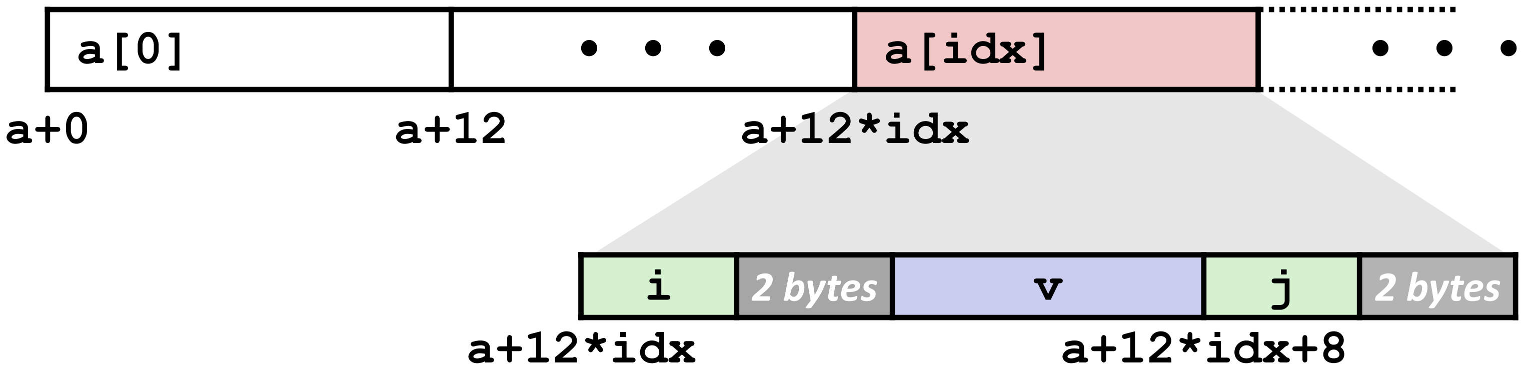
- Arrays are the reason for the overall length requirement
  - Each struct must start at a multiple of its largest member. This means the member is aligned
- The compiler adds trailing padding even without array declaration

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```



- `sizeof(S3)=12`, including padding

- Compute array offset `12*idx`
- Element `j` is at offset 8 within structure

```
short get_j(int idx)
{
  return a[idx].j;
}
```

- Assembly contains displacement `a+8`
  - Compile-time constant resolved during linking

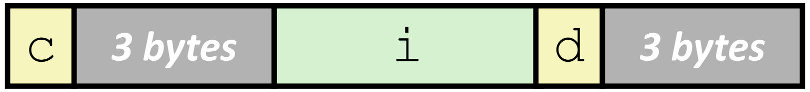
```
 # %rdi = idx
 leaq (%rdi,%rdi,2),%rax # 3*idx
 movzwl a+8(,%rax,4),%eax
```

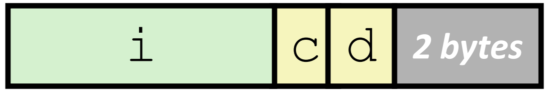
Saving Space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```





- Effect: saved 4 bytes of memory

- C compilers cannot do this automatically!
  - They have to preserve field ordering
  - Programmers must do it manually
  - Other languages aren't bound to preserve ordering. Rust may reorder for you

Break + Quiz

- What is the total size of this struct?

```
typedef struct {
  short a;
  int b;
  char* c[3];
  char d;
}
```

Break + Quiz

- What is the total size of this struct?

```
typedef struct {
  short a;
  int b;
  char* c[3];
  char d;
}
```

2 bytes for **a**

(2 bytes for padding)

4 bytes for **b**

(no padding needed, 8-aligned)

24 bytes for **c**

(no padding needed, 1-aligned)

1 byte for **d**

(7 bytes padding after struct)

= 40 bytes total

Could have been 32 bytes if reordered

How about C++?

- We’ve covered everything we need to from assembly

- Do we know enough to “compile” C++ in x86-64?
  - Yes!

  - Classes are structs
    - Likely with extra members to keep track of things
    - And function pointers as members

  - References are just pointers that the compiler handles for you

Outline

- Pointers

- One-dimensional Arrays
- Multi-dimensional Arrays
- Multi-level Arrays

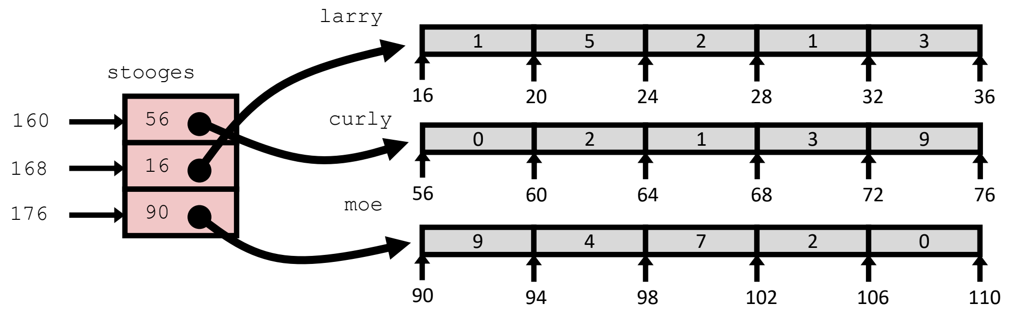
- Struct Layout
- Struct Padding and Alignment

Bonus Material

- **Bonus: Dynamic arrays**

Dynamic Multi-dimensional arrays – multi-level

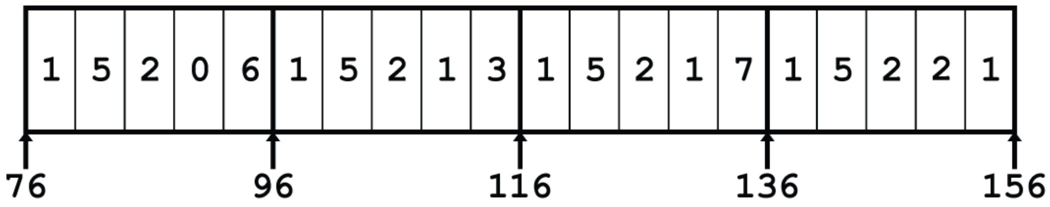
- Multi-level is one way to make them




```
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
    - Compiler won’t do it for you 😢
  - 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

```c
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
```

Bonus Material

- **Bonus: Unions**

Unions

- Structs = combine multiple pieces of data into one
  - Think: "all of the above"

- Unions = choose between multiple different kinds of data
  - Think: "any of the above"

- Typically used in conjunction with a struct: *variants*
  - That tells us which branch of the union is used
  - E.g., **`which_kind`** of 0 to mean sandwich meal, 1 for pizza, etc.

```
typedef struct {
  char which_kind;
  char n_sides;
  char cost;
  MealKind_t mk;
} Meal_t;
```

```
typedef union {
  Sandwich_t s;
  Pizza_t p;
  Burrito_t b;
} MealKind_t;
```

```
typedef struct {
  int n_pieces_bread;
  char *toppings[2];
  float mayo_ounces;
} Sandwich_t;
```

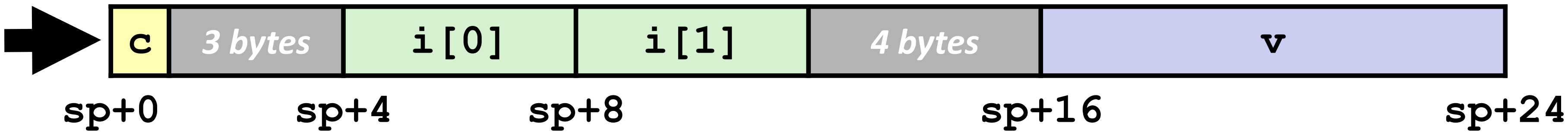
Union allocation

- Principles
  - Overlay union elements
  - Allocate according to largest element (strictest)
  - Can only use one field at a time

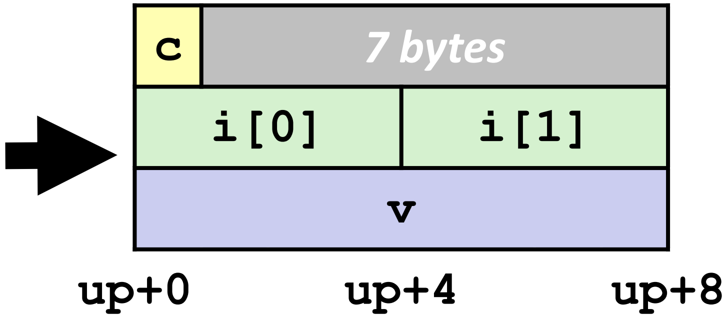
Structs: *All* of the above, together, one after the other.

Unions: *One* of the above, you pick the one you want.

```
struct S1 {
  char c;
  int i[2];
  double v;
} sp;
```




```
union U1 {
  char c;
  int i[2];
  double v;
} up;
```



- Union: same bits, different contexts
  - 8 bytes are allocated for the union
  - Can be interpreted as any member
  - Changing one member will change some bits of the others

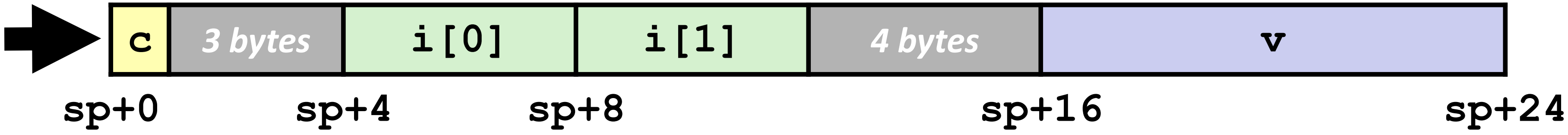
Union allocation

- Principles
  - Overlay union elements
  - Allocate according to largest element (strictest)
  - Can only use one field at a time

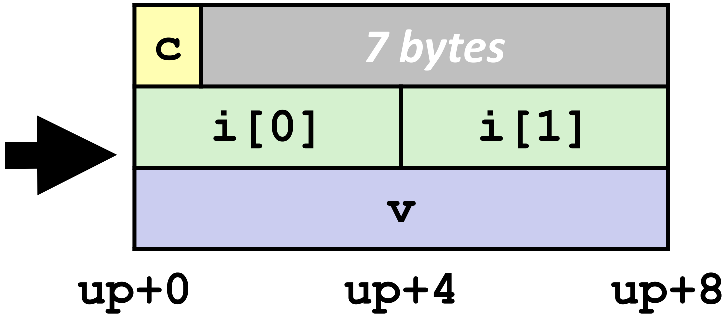
Structs: *All* of the above, together, one after the other.

Unions: *One* of the above, you pick the one you want.

```
struct S1 {
  char c;
  int i[2];
  double v;
} sp;
```




```
union U1 {
  char c;
  int i[2];
  double v;
} up;
```



Quiz: If we had 3 `int`s in that array, how much space would the union take?

Answer: 16 bytes (8-byte aligned)

Using union to access bit patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```


```
unsigned float2bit(float f) {
  bit_float_t temp;
  temp.f = f;
  return temp.u;
}
```

```
  # procedure with float arg
  # arg1 passed in %xmm0
  # movss = move single-precision
  movss %xmm0, -4(%rsp)
  movl -4(%rsp), %eax
  ret
```

- Store union using one type & access it with another one
- Get direct access to bit representation of float
- float2bit generates bit pattern from float
  - NOT the same as `(unsigned) f` !
  - Doesn’t convert value to unsigned
  - Keeps the same bits but interprets them differently
- Assembly doesn’t have type info
  - Just moves the bytes

Access to Bit Pattern Non-Solution

```
unsigned float2bit(float f)
{
  unsigned *p;
  p = (unsigned *) &f;
  return *p;
}
```

Undefined behavior in C.

Don’t do that.

Byte ordering revisited

- Idea
  - Words/long words/quad words stored in memory as 2/4/8 consecutive bytes
  - At which byte address in memory is the most (least) significant byte stored?
  - Can cause problems when exchanging binary data between machines

- Little Endian
  - Least significant byte has lowest address
  - Intel x86(-64), ARM Android and IOS

- Big Endian
  - Most significant byte has lowest address
  - Sun/Sparc, Networks

- Have to worry about it when working with unions!

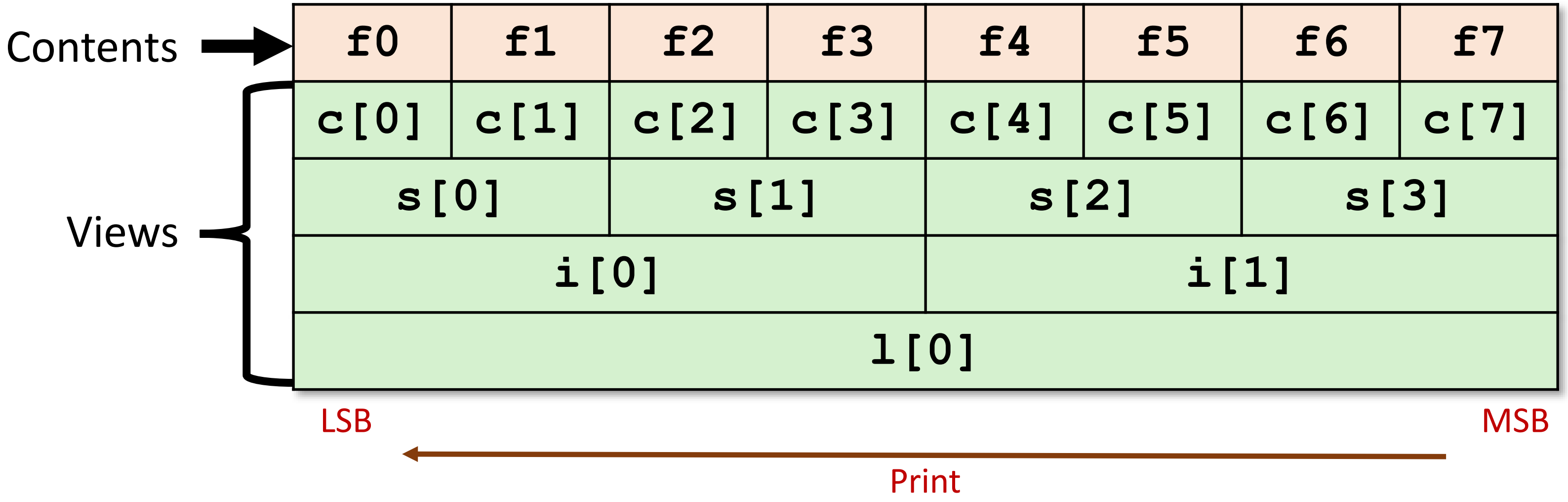
Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

```
for (int j = 0; j < 8; j++) {
    dw.c[j] = 0xf0 + j;
}

printf("Chars 0-7 ==  [0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x, 0x%x, 0x%x, 0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x, 0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte ordering on Little Endian

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

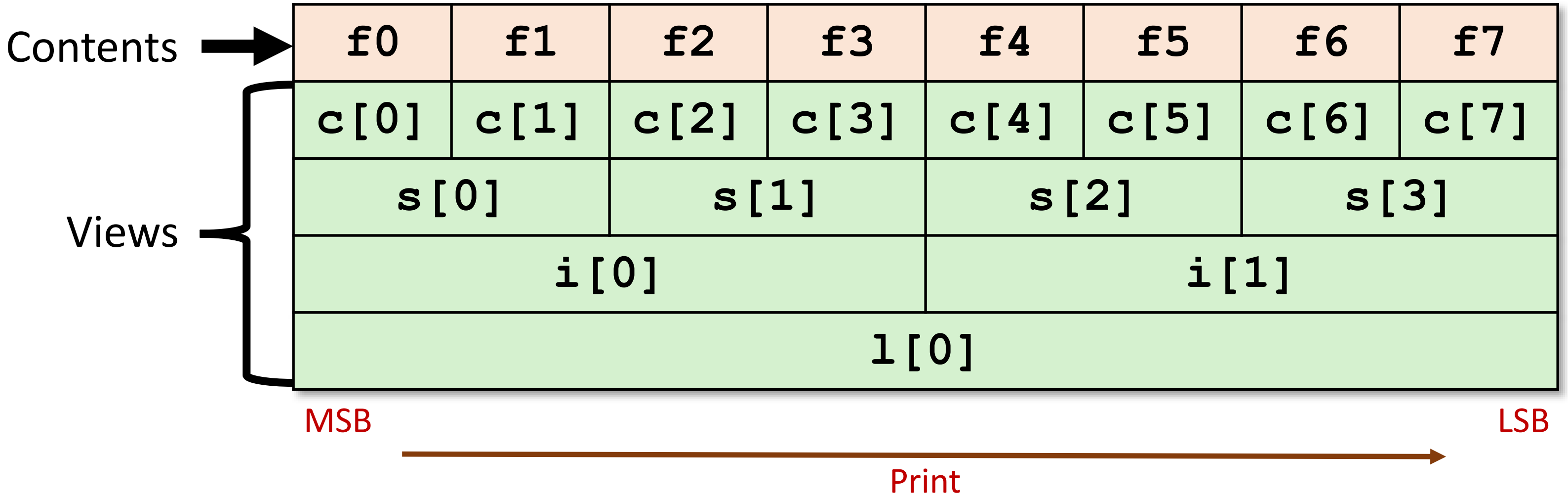


Output:

```
Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts     0-3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Ints       0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Byte ordering on Big Endian

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```



Output:

```
Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts     0-3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
Ints       0-1 == [0xf0f1f2f3, 0xf4f5f6f7]
Long       0   == [0xf0f1f2f3f4f5f6f7]
```