

Machine-Level Programming IV: Data

15-213: Introduction to Computer Systems
7th Lecture, September 20, 2022

Instructors:

Dave Andersen (15-213)

Zack Weinberg (15-213)

Brian Railing (15-513)

David Varodayan (14-513)

Today

■ Partial recap: Integers

- Word size
- Addresses

■ One-Dimensional Arrays

■ Structs

- Alignment
- Arrays of Structs

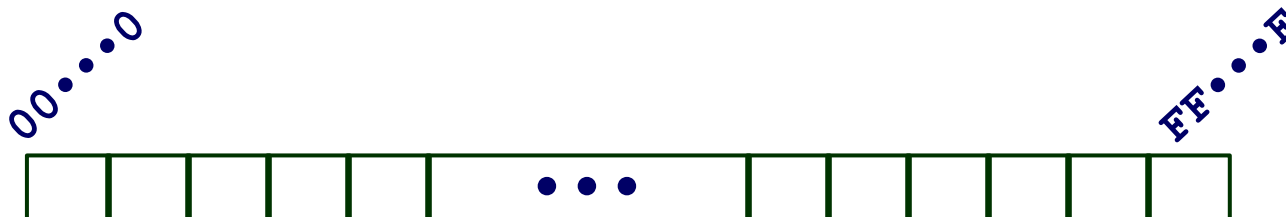
■ Multi-Dimensional Arrays

- Nested (Arrays of Arrays)
- (Arrays of) Pointers to Arrays

■ If we have time:

- Endianness
- Machine Instructions

Byte-Oriented Memory Organization



■ Programs refer to data by address

- Imagine all of RAM as an enormous array of bytes
- An address is an index into that array
 - A pointer variable stores an address

■ System provides a private *address space* to each “process”

- A process is an instance of a program, being executed
- An address space is one of those enormous arrays of bytes
- Each program can see only its own code and data within its enormous array
- We’ll come back to this later (“virtual memory” classes)

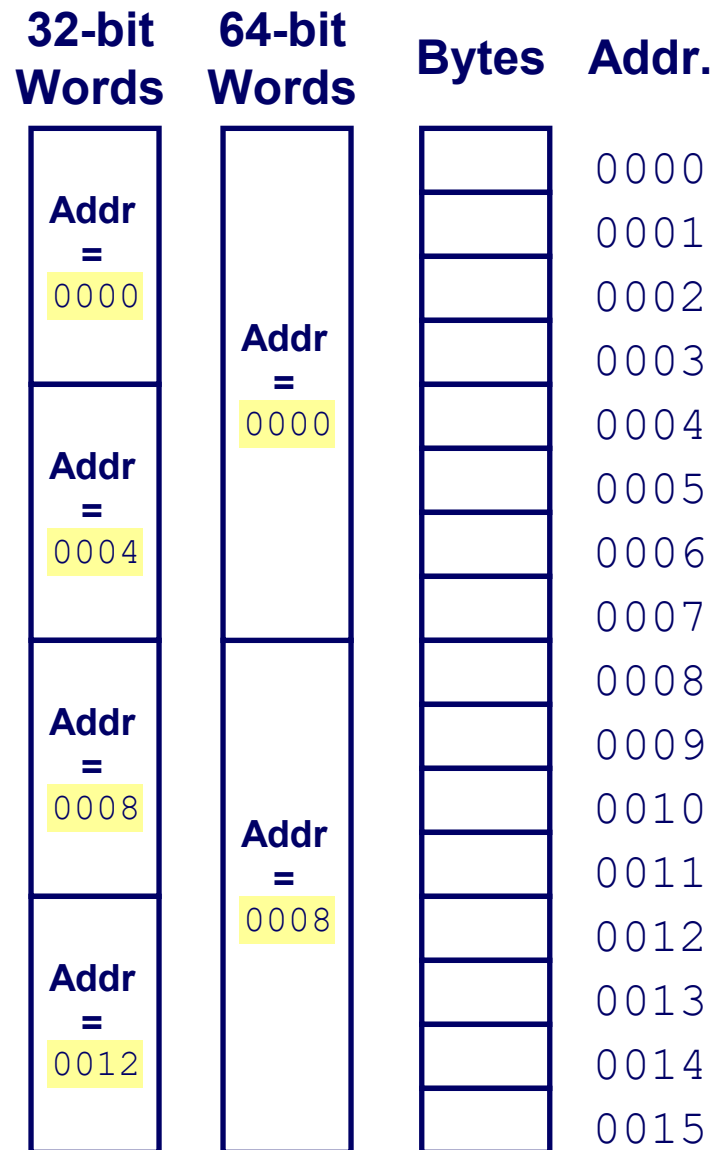
Machine Words

■ Any given computer has a “Word Size”

- Nominal size of integer-valued data
 - and of addresses
- Until recently, most machines used 32 bits (4 bytes) as word size
 - Limits addresses to 4GB (2^{32} bytes)
- Increasingly, machines have 64-bit word size
 - Potentially, could have 16 EB (exabytes) of addressable memory
 - That's 18.4×10^{18} bytes
- Machines still support multiple data formats
 - Fractions or multiples of word size
 - Always integral number of bytes

Addresses *Always* Specify Byte Locations

- Address of a word is address of the first byte in the word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



Today

■ Partial recap: Integers

- Word size
- Addresses

■ One-Dimensional Arrays

■ Structs

- Alignment
- Arrays of Structs

■ Multi-Dimensional Arrays

- Nested (Arrays of Arrays)
- (Arrays of) Pointers to Arrays

■ If we have time:

- Endianness
- Machine Instructions

Activity break:
pick a partner (just one other student this time),
open activity
(<https://canvas.cmu.edu/courses/30386/assignments/528617>),
do parts 1 and 2

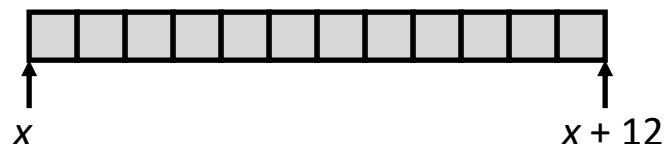
Array Allocation

■ Basic Principle

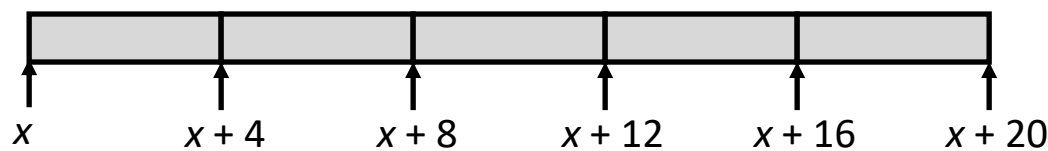
T $A[L]$;

- Array of data type T and length L
- Contiguously allocated region of $L * \text{sizeof}(T)$ bytes in memory

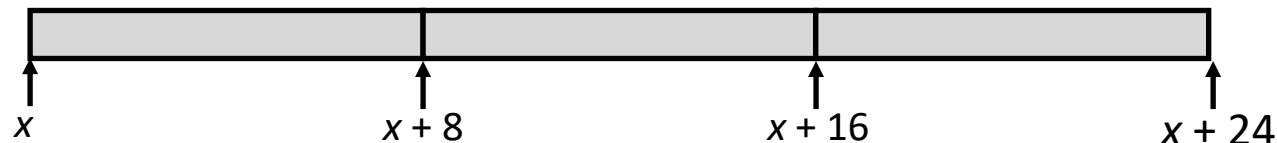
`char string[12];`



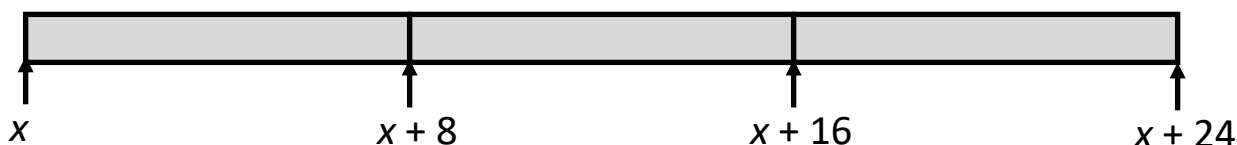
`int val[5];`



`double a[3];`



`char *p[3];`

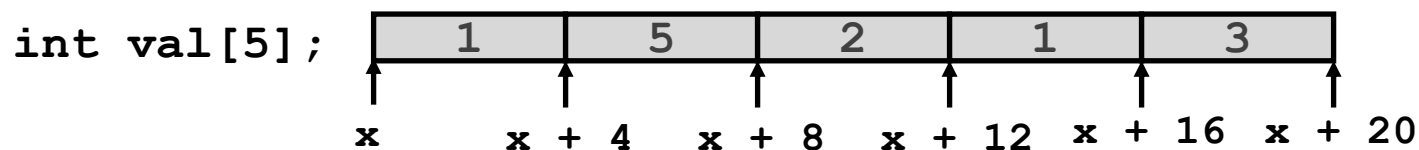


Array Access

Basic Principle

T $A[L]$;

- Array of data type T and length L
- Identifier A can be used as a pointer to array element 0: Type T^*

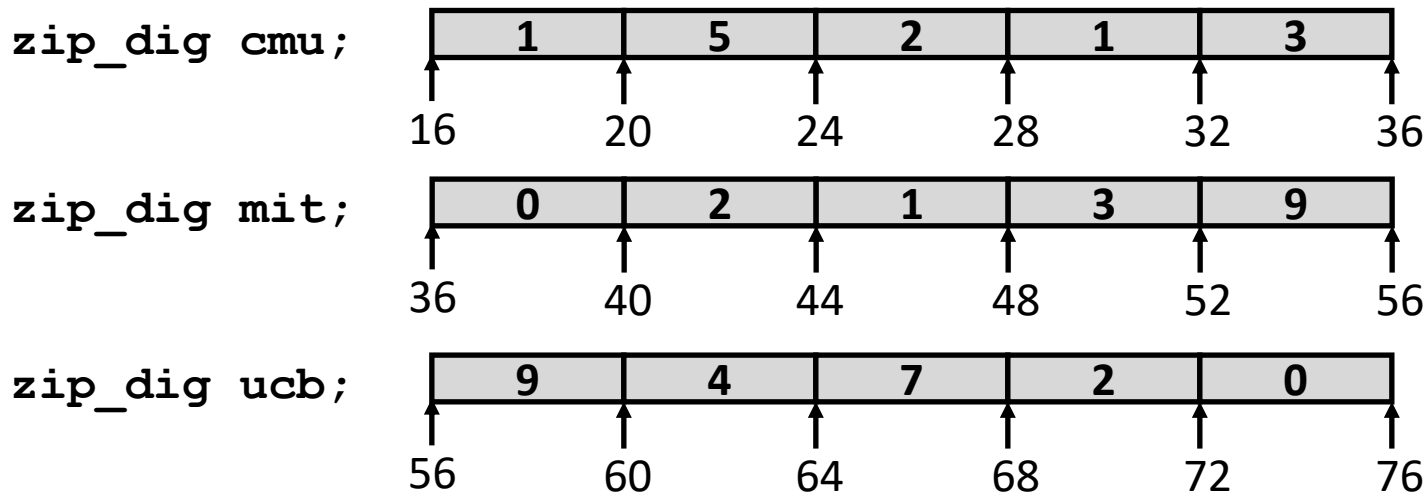


Reference	Type	Value
<code>val[4]</code>	<code>int</code>	3
<code>val</code>	<code>int *</code>	x
<code>val+1</code>	<code>int *</code>	$x + 4$
<code>&val[2]</code>	<code>int *</code>	$x + 8$
<code>val[5]</code>	<code>int</code>	??
<code>*(val+1)</code>	<code>int</code>	5 <code>//val[1]</code>
<code>val + i</code>	<code>int *</code>	$x + 4 * i$ <code>//&val[i]</code>

Array Example

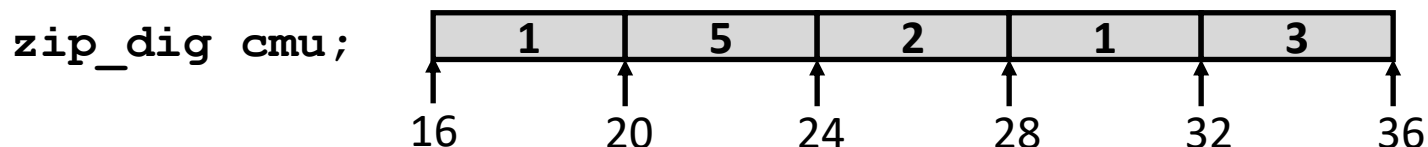
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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



- Declaration “zip_dig cmu” equivalent to “int cmu[5]”
- Example arrays were allocated in successive 20 byte blocks
 - Not guaranteed to happen in general

Array Accessing Example



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

x86-64

```
# %rdi = z
# %rsi = 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`
- Use memory reference `(%rdi,%rsi,4)`

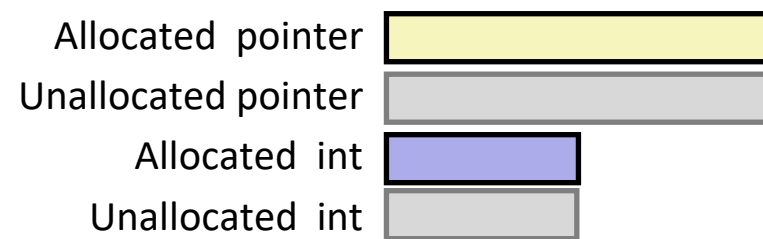
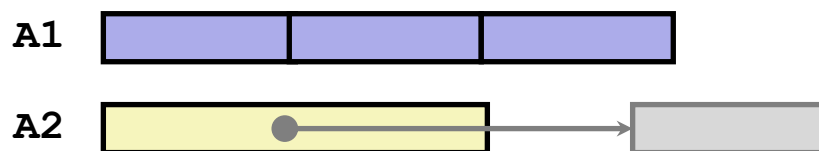
Understanding Pointers & Arrays #1

Decl	<i>An</i>			<i>*An</i>		
	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>						
<code>int *A2</code>						

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

Understanding Pointers & Arrays #1

Decl	<i>A_n</i>			<i>*A_n</i>		
	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>	Y	N	12	Y	N	4
<code>int *A2</code>	Y	N	8	Y	Y	4



- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

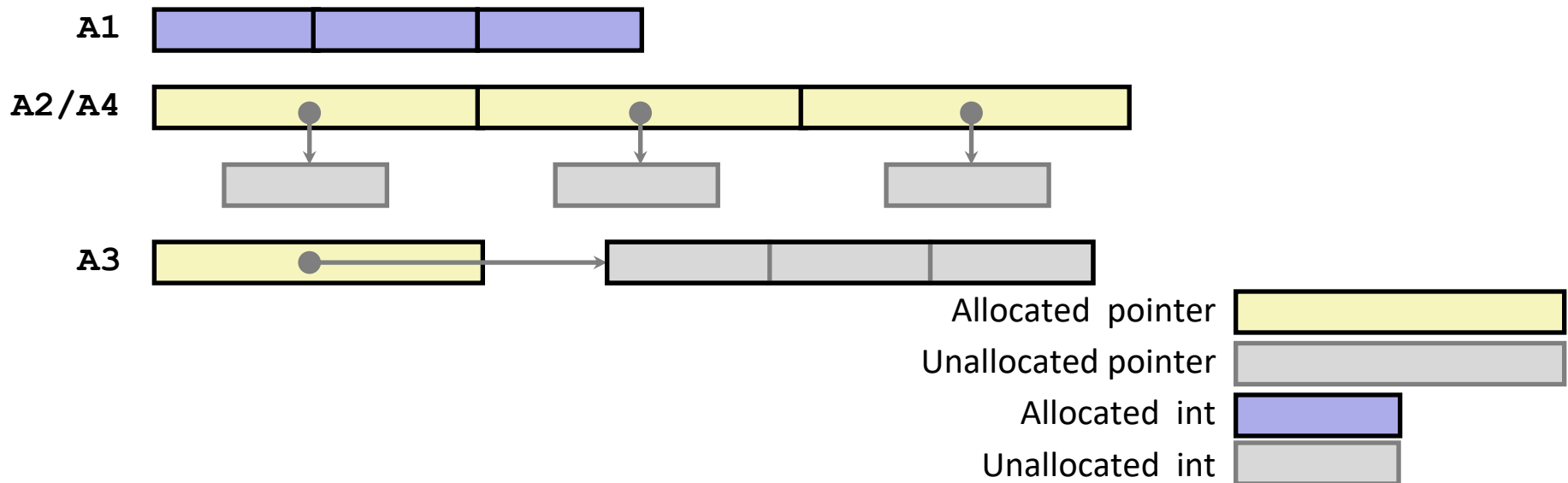
Understanding Pointers & Arrays #2

Decl	<i>A_n</i>			<i>*A_n</i>			<i>**A_n</i>		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>									
<code>int *A2[3]</code>									
<code>int (*A3)[3]</code>									
<code>int (*A4[3])</code>									

- **Cmp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

Understanding Pointers & Arrays #2

Decl	A_n			$*A_n$			$**A_n$		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>	Y	N	12	Y	N	4	N	-	-
<code>int *A2[3]</code>	Y	N	24	Y	N	8	Y	Y	4
<code>int (*A3)[3]</code>	Y	N	8	Y	Y	12	Y	Y	4
<code>int (**A4)[3]</code>	Y	N	24	Y	N	8	Y	Y	4



Today

■ Partial recap: Integers

- Word size
- Addresses

■ One-Dimensional Arrays

■ Structs

- Alignment
- Arrays of Structs

■ Multi-Dimensional Arrays

- Nested (Arrays of Arrays)
- (Arrays of) Pointers to Arrays

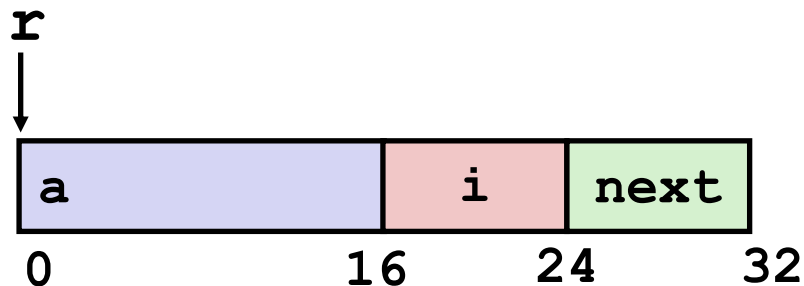
■ If we have time:

- Endianness
- Machine Instructions

Activity break:
do parts 3 and 4 now

Structure Representation

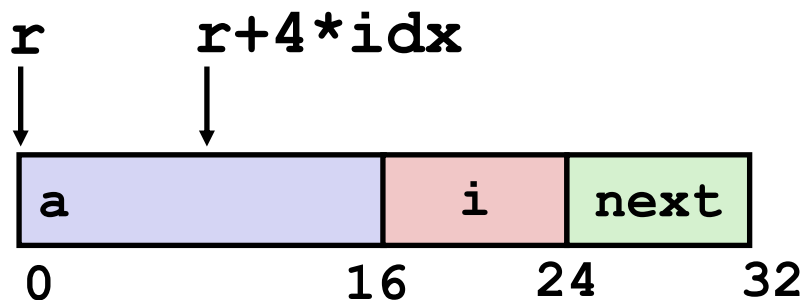
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- **Structure represented as block of memory**
 - Big enough to hold all of the fields
- **Fields ordered according to declaration**
 - Even if another ordering could yield a more compact representation
- **Compiler determines overall size + positions of fields**
 - Machine-level program has no understanding of the structures in the source code

Generating Pointer to Structure Member

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



■ Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as $r + 4 * idx$

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

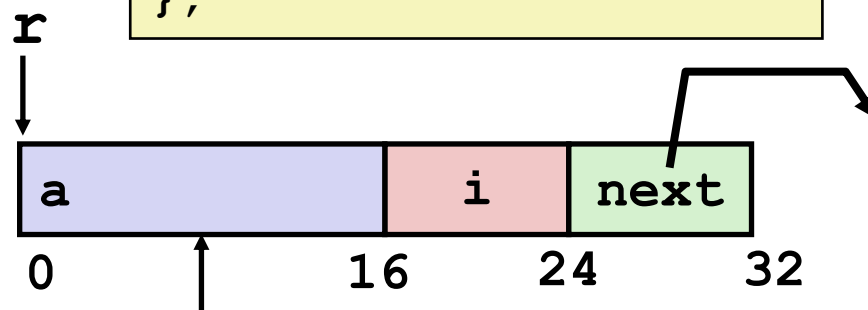
```
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```

Following Linked List

■ C Code

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

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



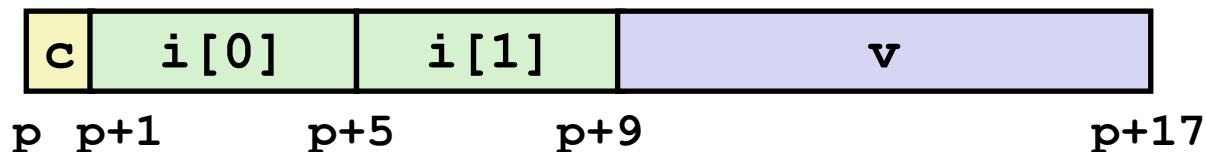
Element *i*

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
```

Structures & Alignment

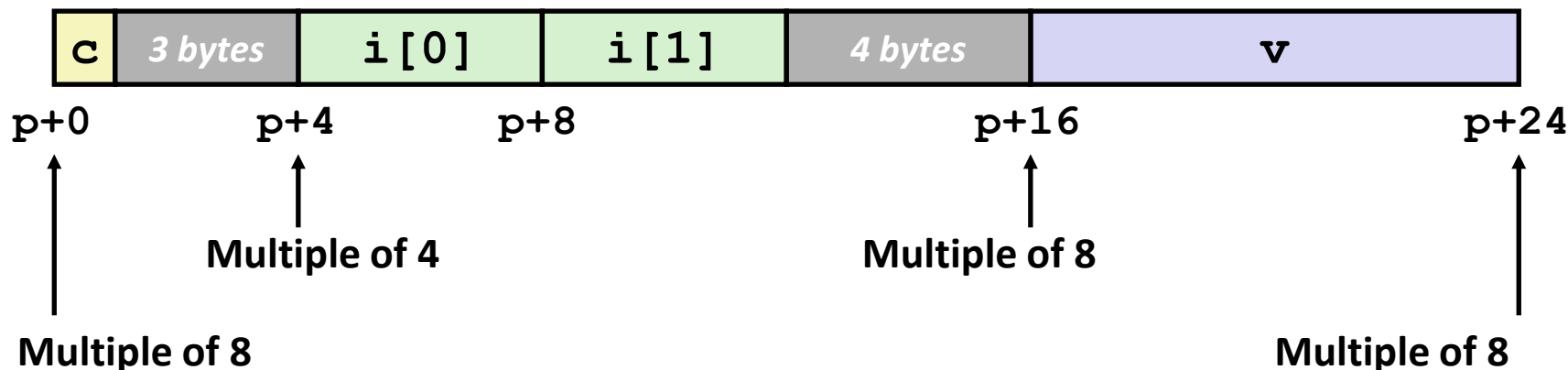
■ Unaligned Data



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

■ Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K



Alignment Principles

■ Aligned Data

- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on x86-64

■ Motivation for Aligning Data

- Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
 - Inefficient to load or store datum that spans quad word boundaries
 - Virtual memory trickier when datum spans 2 pages

■ Compiler

- Inserts gaps in structure to ensure correct alignment of fields

Specific Cases of Alignment (x86-64)

■ 1 byte: `char`, ...

- no restrictions on address

■ 2 bytes: `short`, ...

- lowest 1 bit of address must be 0_2

■ 4 bytes: `int`, `float`, ...

- lowest 2 bits of address must be 00_2

■ 8 bytes: `double`, `long`, `char *`, ...

- lowest 3 bits of address must be 000_2

Satisfying Alignment with Structures

■ Within structure:

- Must satisfy each element's alignment requirement

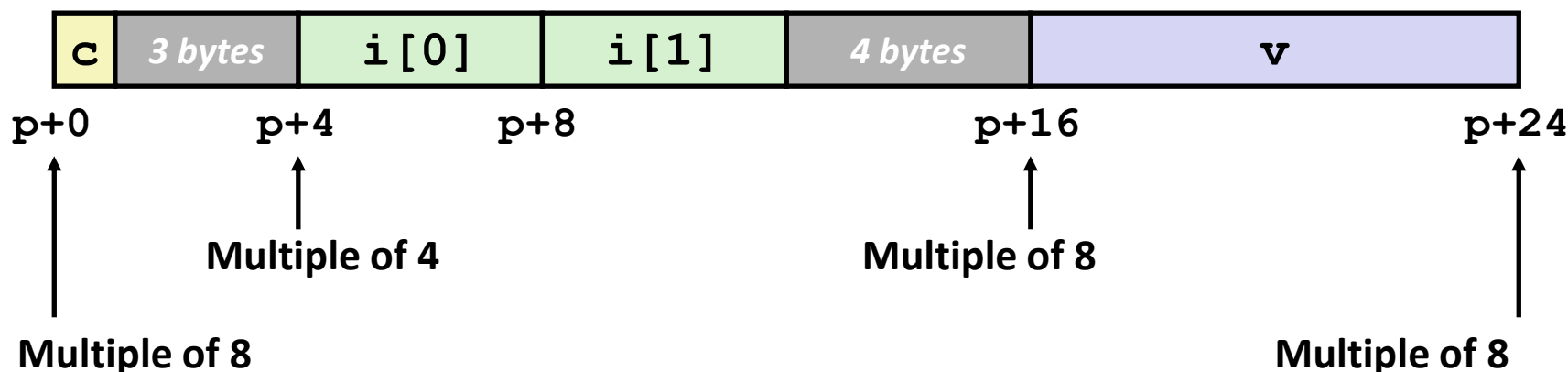
■ Overall structure placement

- Each structure has alignment requirement K
 - K = Largest alignment of any element
- Initial address & structure length must be multiples of K

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

■ Example:

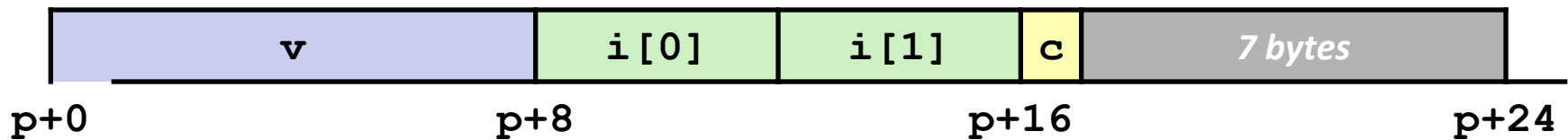
- $K = 8$, due to **double** element



Meeting Overall Alignment Requirement

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

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

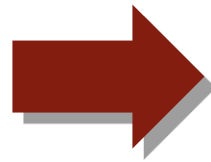


Multiple of $K=8$

Saving Space

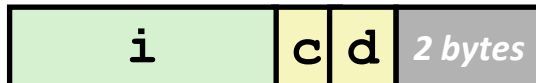
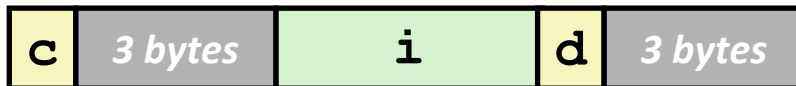
■ Put large data types first

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



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

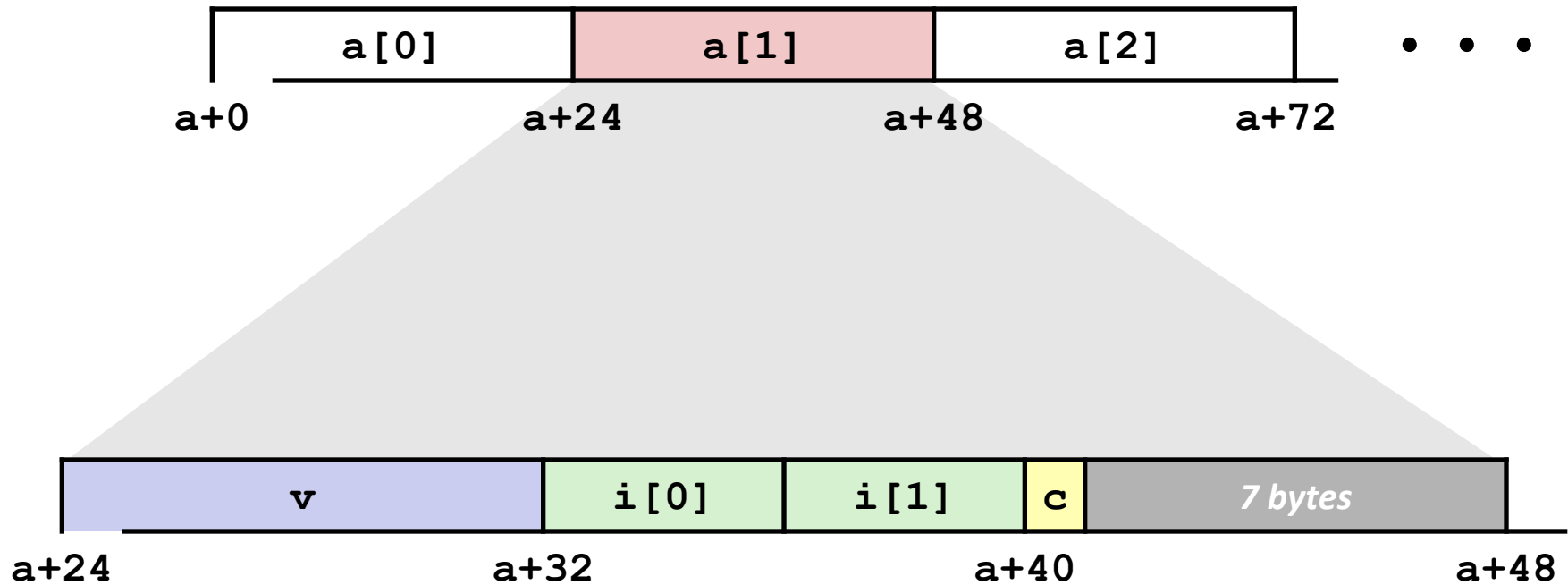
■ Effect (K=4)



Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

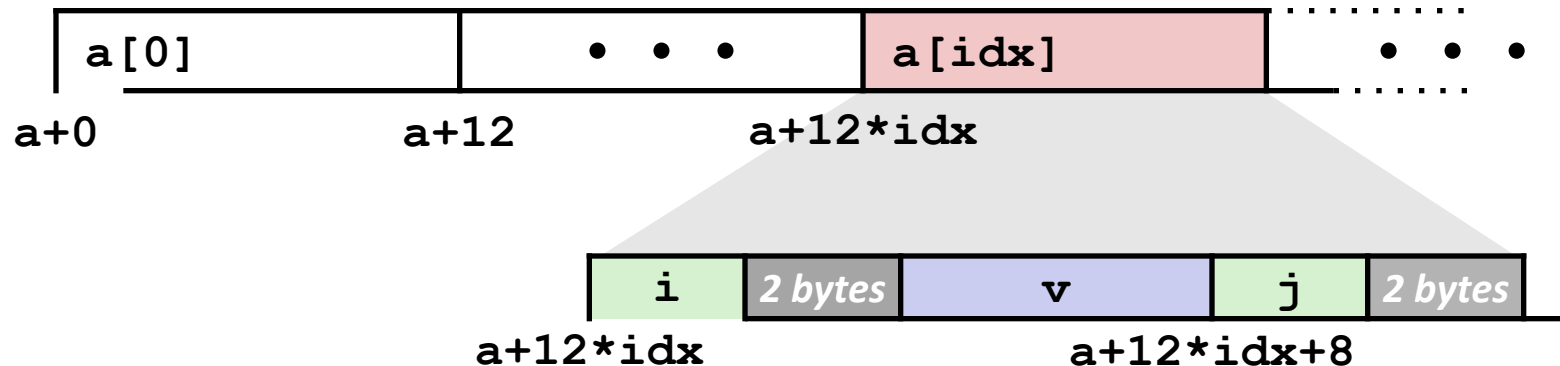
```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```



Accessing Array Elements

- **Compute array offset $12 \cdot \text{idx}$**
 - `sizeof(S3)`, including alignment spacers
- **Element `j` is at offset 8 within structure**
- **Assembler gives offset `a+8`**
 - Resolved during linking

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



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

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

Today

■ Partial recap: Integers

- Word size
- Addresses

■ One-Dimensional Arrays

■ Structs

- Alignment
- Arrays of Structs

■ Multi-Dimensional Arrays

- Nested (Arrays of Arrays)
- (Arrays of) Pointers to Arrays

■ If we have time:

- Endianness
- Machine Instructions

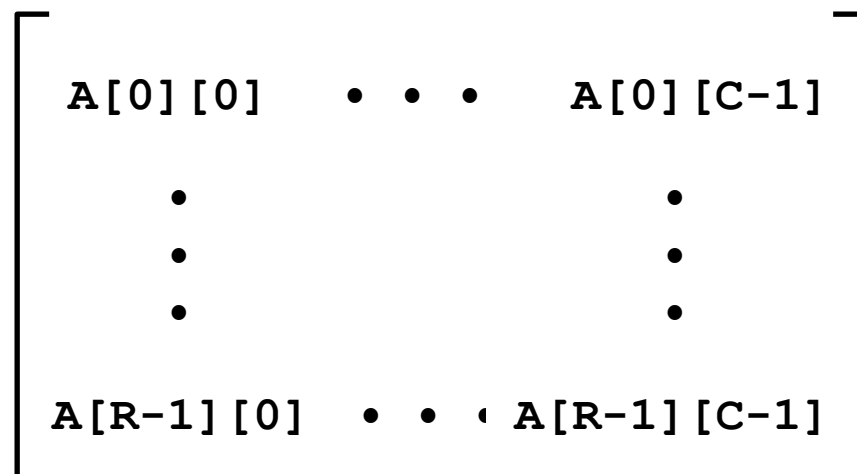
Activity break:
do part 5 now

Multidimensional (Nested) Arrays

Declaration

```
T A[R][C];
```

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



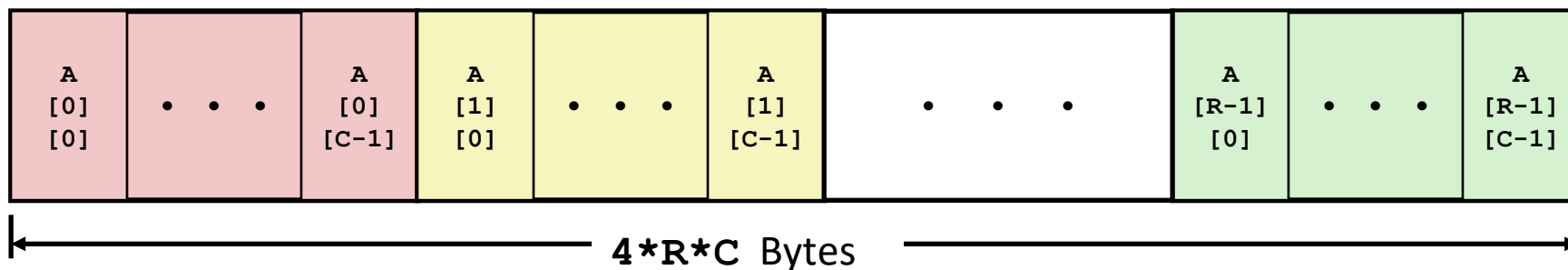
Array Size

- $R * C * K$ bytes

Arrangement

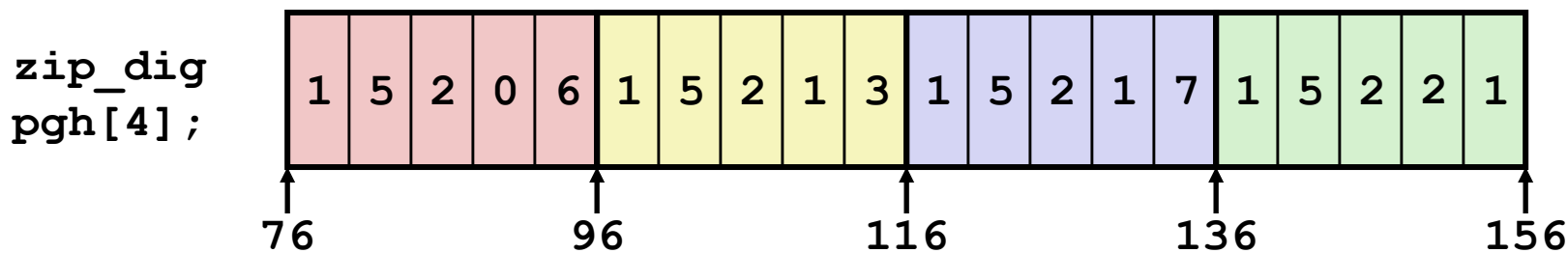
- Row-Major Ordering

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



Nested Array Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```



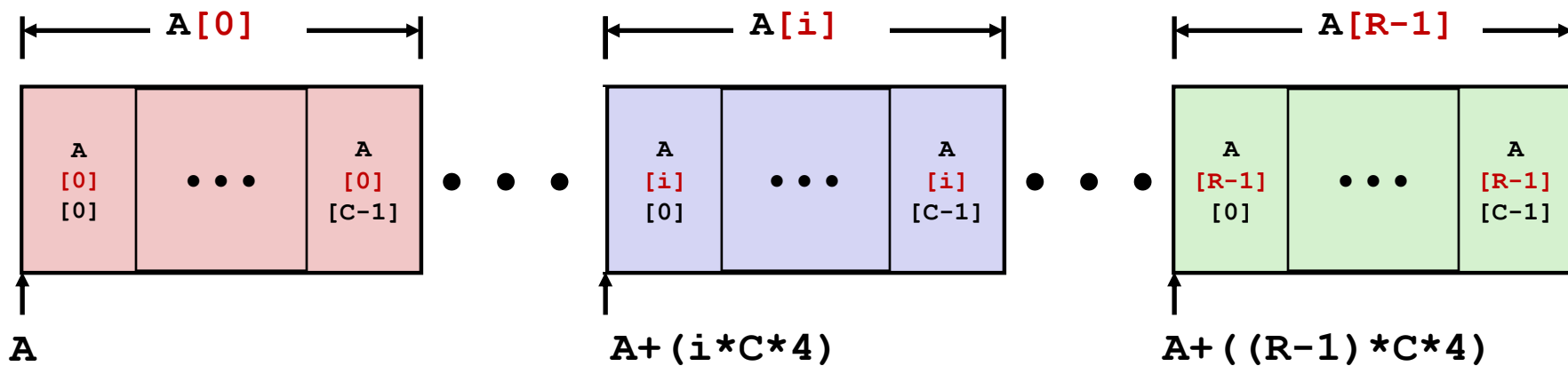
- **“zip_dig pgh[4]” equivalent to “int pgh[4][5]”**
 - Variable `pgh`: array of 4 elements, allocated contiguously
 - Each element is an array of 5 `int`'s, allocated contiguously
- **“Row-Major” ordering of all elements in memory**

Nested Array Row Access

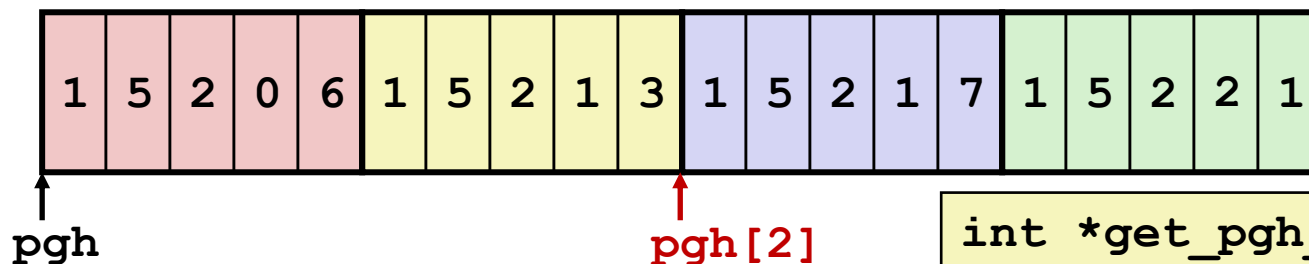
■ Row Vectors

- $A[i]$ 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_pgh_zip(int index)
{
    return pgh[index];
}
```

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

■ Row Vector

- `pgh[index]` is array of 5 `int`'s
- Starting address `pgh+20*index`

■ Machine Code

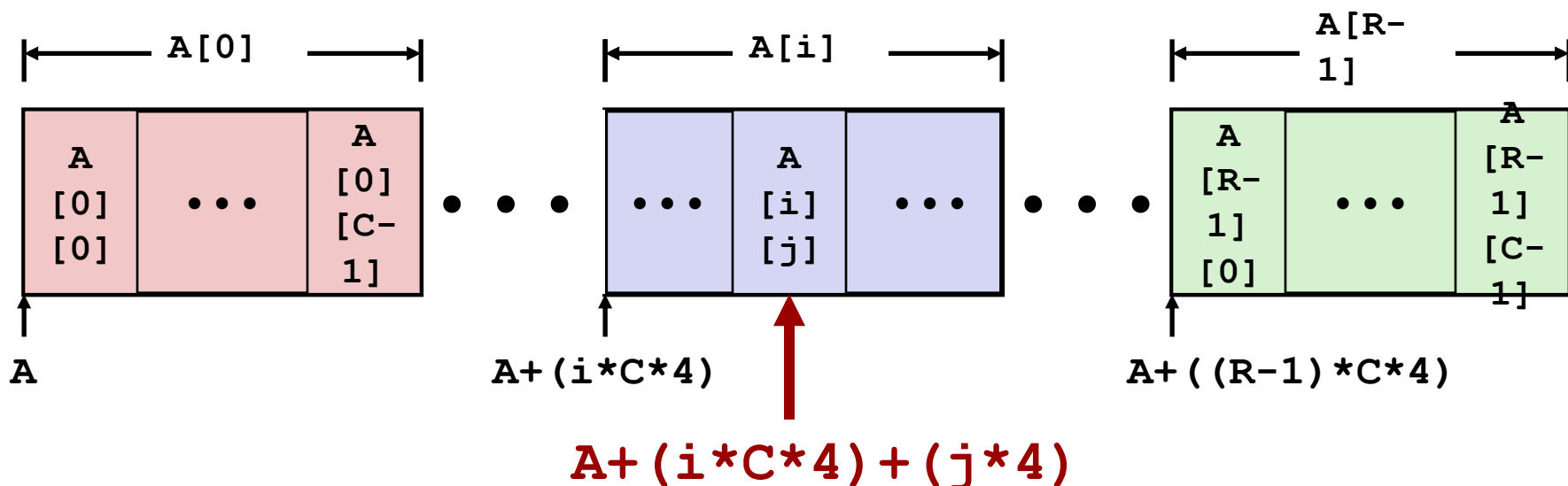
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`

Nested Array Element Access

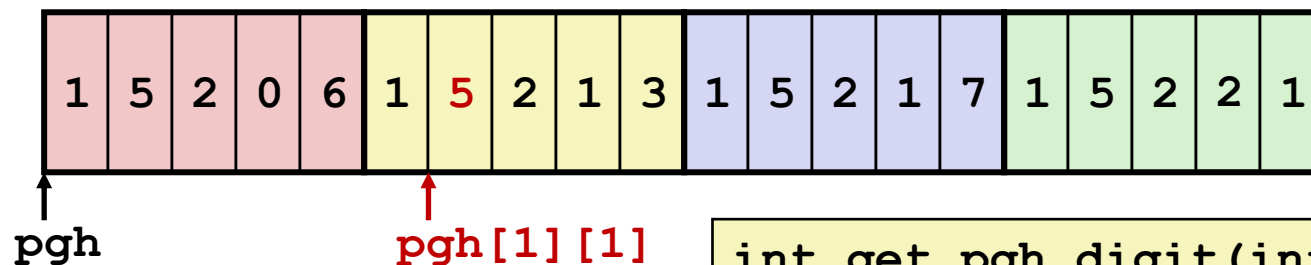
■ Array Elements

- $A[i][j]$ 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_pgh_digit(int index, int dig)
{
    return pgh[index][dig];
}
```

```
leaq    (%rdi,%rdi,4), %rax # 5*index
addl    %rax, %rsi      # 5*index+dig
movl    pgh(,%rsi,4), %eax # M[pgh + 4*(5*index+dig)]
```

■ Array Elements

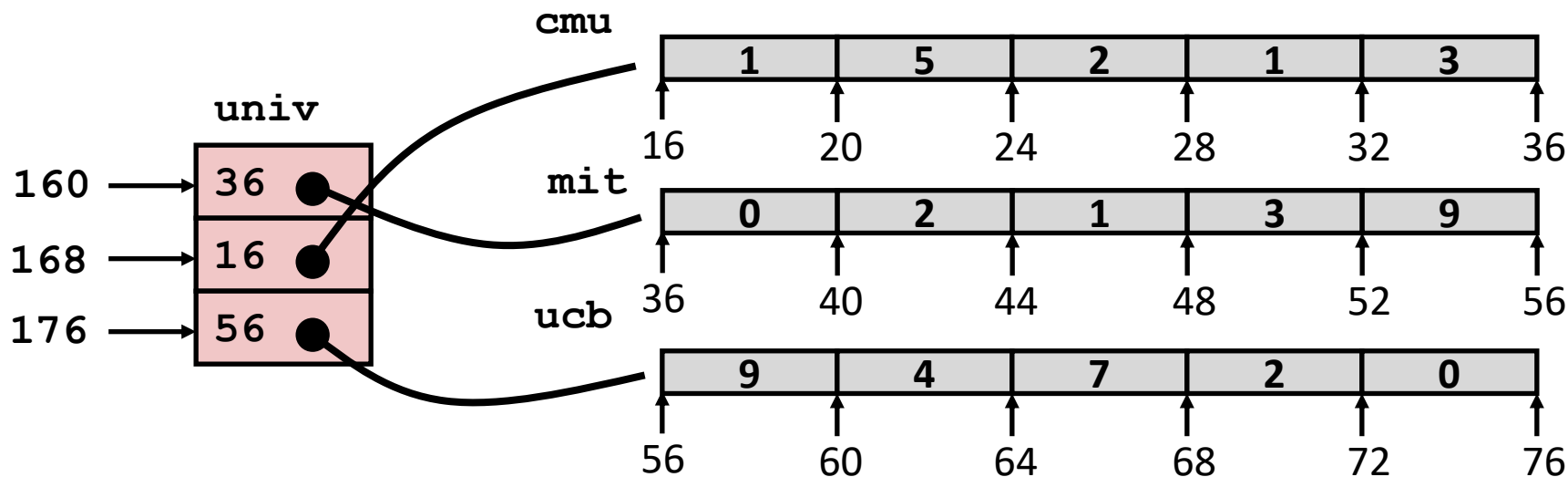
- `pgh[index][dig]` is `int`
- Address: $\text{pgh} + 20 \cdot \text{index} + 4 \cdot \text{dig}$
 $= \text{pgh} + 4 \cdot (5 \cdot \text{index} + \text{dig})$

Multi-Level Array Example

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

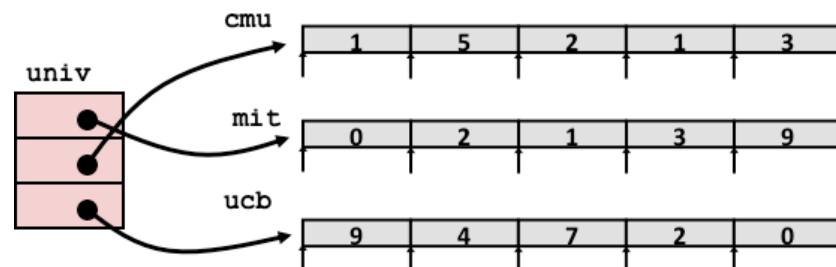
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

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



Element Access in Multi-Level Array

```
int get_univ_digit
(size_t index, size_t digit)
{
    return univ[index][digit];
}
```



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

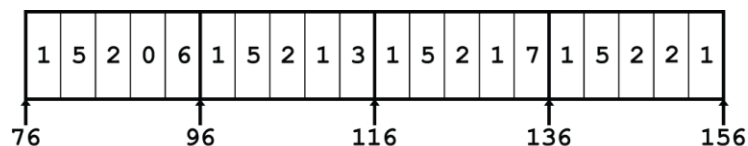
■ Computation

- 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

Array Element Accesses

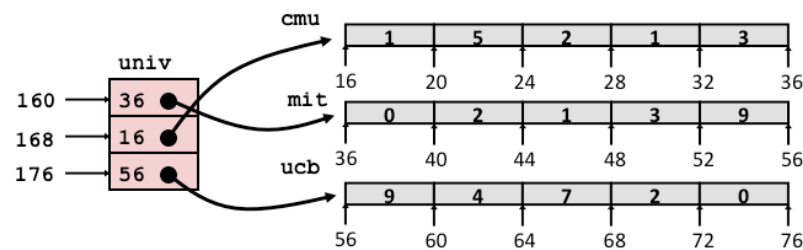
Nested array

```
int get_pgh_digit
(size_t index, size_t digit)
{
    return pgh[index][digit];
}
```



Multi-level array

```
int get_univ_digit
(size_t index, size_t digit)
{
    return univ[index][digit];
}
```



Accesses looks similar in C, but address computations very different:

$\text{Mem}[\text{pgh} + 20 * \text{index} + 4 * \text{digit}]$

$\text{Mem}[\text{Mem}[\text{univ} + 8 * \text{index}] + 4 * \text{digit}]$

$N \times N$ Matrix

Code

■ Fixed dimensions

- Know value of N at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element A[i][j] */
int fix_ele(fix_matrix A,
           size_t i, size_t j)
{
    return A[i][j];
}
```

■ Variable dimensions, explicit indexing

- Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element A[i][j] */
int vec_ele(size_t n, int *A,
           size_t i, size_t j)
{
    return A[IDX(n,i,j)];
}
```

■ Variable dimensions, implicit indexing

- “New” feature in C99

```
/* Get element a[i][j] */
int var_ele(size_t n, int A[n][n],
           size_t i, size_t j) {
    return A[i][j];
}
```

Summary

■ Arrays

- Elements packed into contiguous region of memory
- Use index arithmetic to locate individual elements

■ Structures

- Elements packed into single region of memory
- Access using offsets determined by compiler
- Possible require internal and external padding to ensure alignment

■ Combinations

- Can nest structure and array code arbitrarily

Today

■ Partial recap: Integers

- Word size
- Addresses

■ One-Dimensional Arrays

■ Structs

- Alignment
- Arrays of Structs

■ Multi-Dimensional Arrays

- Nested (Arrays of Arrays)
- (Arrays of) Pointers to Arrays

■ If we have time:

- Endianness
- Machine Instructions

Activity break:
do part 6 now

Byte Ordering

■ So, how are the bytes within a multi-byte word ordered in memory?

■ Conventions

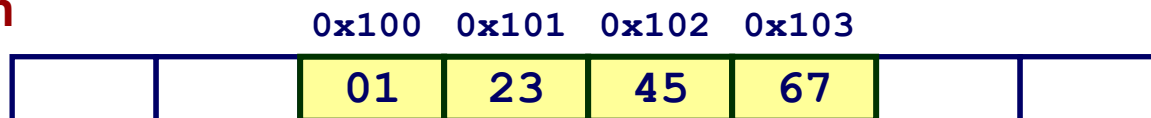
- Big Endian: Sun, PPC Mac, *network packet headers*
 - Least significant byte has highest address
- Little Endian: *x86*, ARM processors running Android, iOS, and Windows
 - Least significant byte has lowest address

Byte Ordering Example

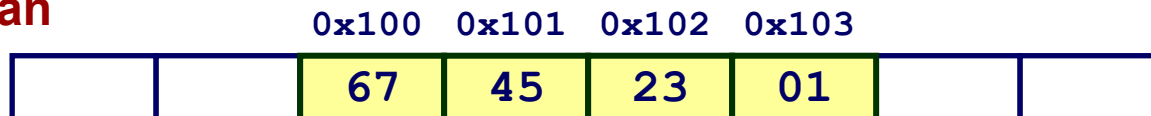
■ Example

- Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100

Big Endian



Little Endian



Examining Data Representations

■ Code to Print Byte Representation of Data

- Casting pointer to unsigned char * allows treatment as a byte array

```
void show_bytes(unsigned char *start, size_t len){
    size_t i;
    for (i = 0; i < len; i++) {
        printf("%p\t%.2x\n",
              (void *)&start[i], start[i]);
    }
}
```

Printf directives:

- %p: Print pointer (must be void *)
- %.2x: Print integer in hexadecimal, with at least two digits

show_bytes Execution Example

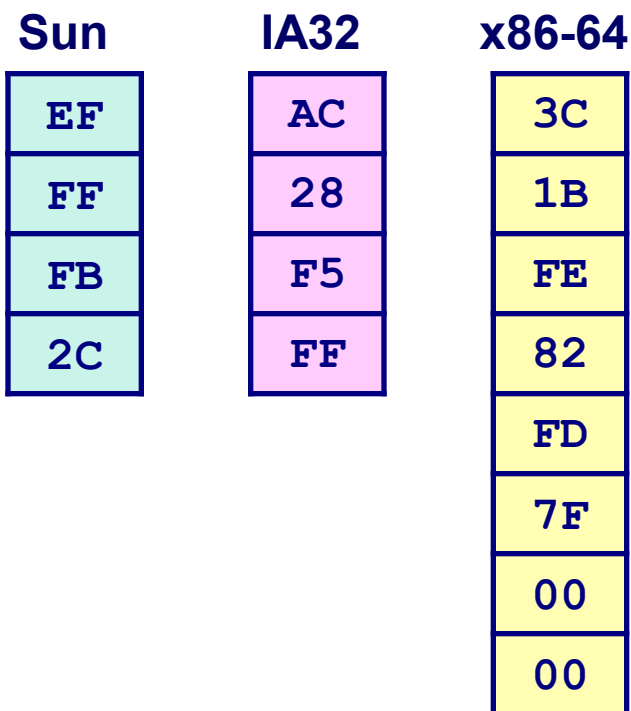
```
int a = 15213;
printf("int a = %d;\n", a);
show_bytes((unsigned char *) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 15213;
0x7ffffb7f71dbc    6d
0x7ffffb7f71dbd    3b
0x7ffffb7f71dbe    00
0x7ffffb7f71dbf    00
```

Representing Pointers

```
int B = -15213;  
int *P = &B;
```



Different compilers & machines assign different locations to objects

May even get different results each time program is run (ASLR)

Representing Strings

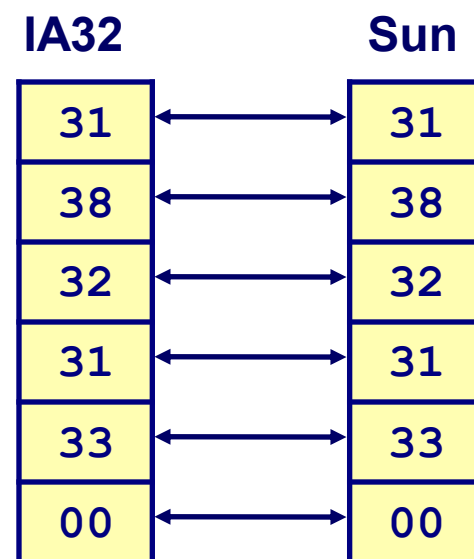
```
char S[6] = "18213";
```

■ Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
 - Standard 7-bit encoding of character set
 - Character "0" has code 0x30
 - Digit i has code $0x30+i$
- String should be null-terminated
 - Final character = 0

■ Compatibility

- Byte ordering not an issue



A note about x86 machine code

■ x86 machine code is a sequence of *bytes*

- Grouped into variable-length instructions, which look like strings...
- But they contain embedded little-endian numbers...

■ Example Fragment

Address	Instruction Code	Assembly Rendition
8048365:	5b	pop %ebx
8048366:	81 c3 ab 12 00 00	add \$0x12ab, %ebx
804836c:	83 bb 28 00 00 00 00	cmpl \$0x0, 0x28(%ebx)

■ Deciphering Numbers

- Value: 0x12ab
- Pad to 32 bits: 0x000012ab
- Split into bytes: 00 00 12 ab
- Reverse: ab 12 00 00

A peek at x86 instruction encoding

and its long, complex history

64-bit mode...

	88	0f	mov	%cl,	(%rdi)
66	89	0f	mov	%cx,	(%rdi)
	89	0f	mov	%ecx,	(%rdi)
	48	89	mov	%rcx,	(%rdi)
	44	88	mov	%r9b,	(%rdi)
66	44	89	mov	%r9w,	(%rdi)
	44	89	mov	%r9d,	(%rdi)
	4c	89	mov	%r9,	(%rdi)

Operand size
= 16 bits

REX prefix:
adjust sizes and
register numbers

ModRM byte:
cx/r9, di,
"addressing mode"

Primary opcode:
MOV reg → mem
+ some operand size info

Same bytes interpreted in 32-bit mode...

	88	0f	mov	%cl,	(%edi)
66	89	0f	mov	%cx,	(%edi)
	89	0f	mov	%ecx,	(%edi)
	48		dec	%eax	
	44		inc	%esp	
66	44		inc	%sp	
	4c		dec	%esp	

Address size
changes to 32 bits

REX becomes
a set of primary
opcodes

and 16-bit mode ...

	88	0f	mov	%cl,	(%bx)
66	89	0f	mov	%ecx,	(%bx)
	89	0f	mov	%cx,	(%bx)
	44		inc	%sp	
66	44		inc	%esp	

Address size
changes to 16 bits,
register numbering
is different

Now means:
Operand size
= 32 bits