

Machine-Level Programming IV: Data

15-213/15-513: Introduction to Computer Systems
6th Lecture, May 24, 2024

Today

■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structures

- Allocation
- Access
- Alignment

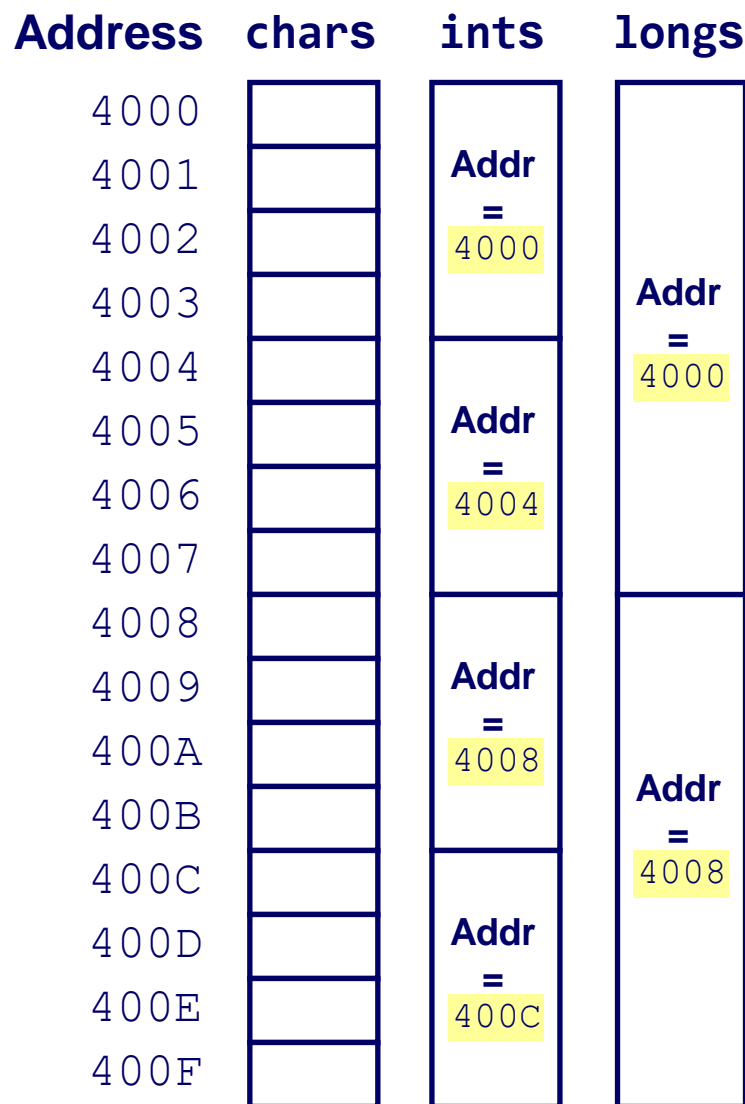
Reminder: Memory Organization

■ Memory locations do not have data types

- Types are implicit in how machine instructions *use* memory

■ Addresses specify byte locations

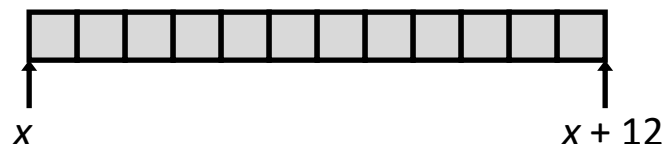
- Address of a larger datum is the address of its first byte
- Addresses of successive items differ by the item's size



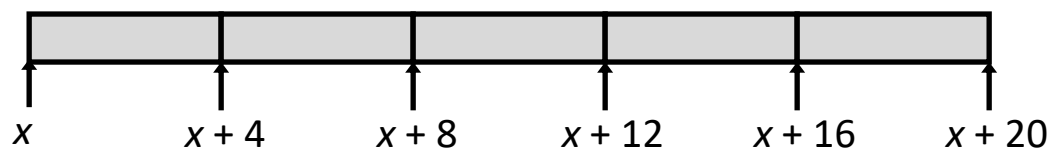
Array Allocation

- C declaration *Type* name [*Length*] ;
 - Array of data type *Type* and length *Length*
 - Contiguously allocated region of $Length * \text{sizeof}(Type)$ bytes in memory

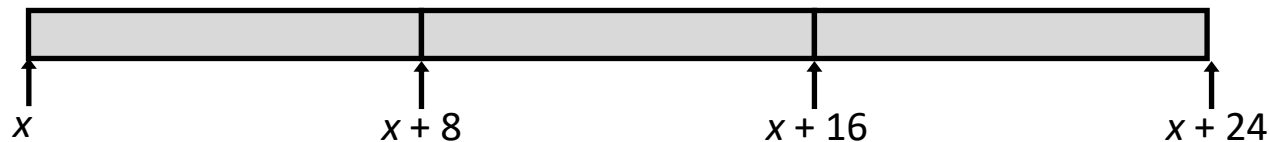
`char string[12];`



`int val[5];`



`double a[3];`



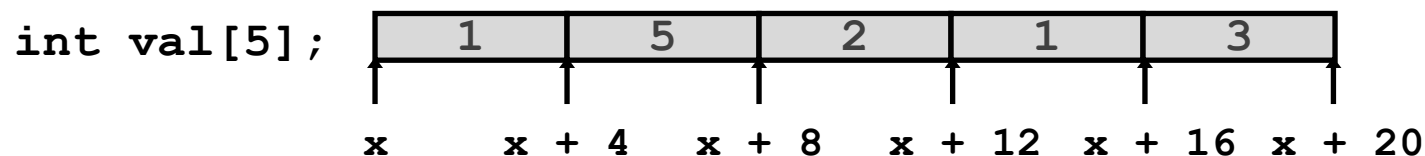
`char *p[3];`



Array Access

■ C declaration *Type name [Length] ;*

- Array of data type *Type* and length *Length*
- Identifier **name** acts like¹ a pointer to array element 0



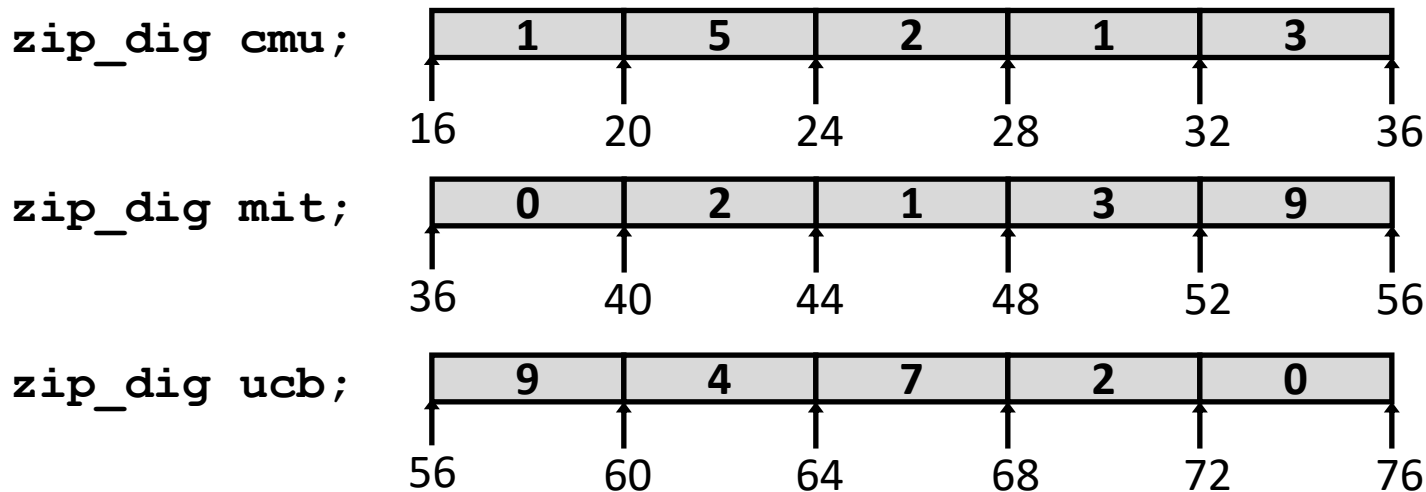
Expression	Type	Value	
<code>val[4]</code>	<code>int</code>	3	
<code>val[5]</code>	<code>int</code>	??	// access past end
<code>*(val+3)</code>	<code>int</code>	1	// same as <code>val[3]</code>
<code>val</code>	<code>int *</code>	x	
<code>val+1</code>	<code>int *</code>	$x + 4$	
<code>&val[2]</code>	<code>int *</code>	$x + 8$	// same as <code>val+2</code>
<code>val + i</code>	<code>int *</code>	$x + 4*i$	// same as <code>&val[i]</code>

¹ in most contexts (but not all)

Array Example

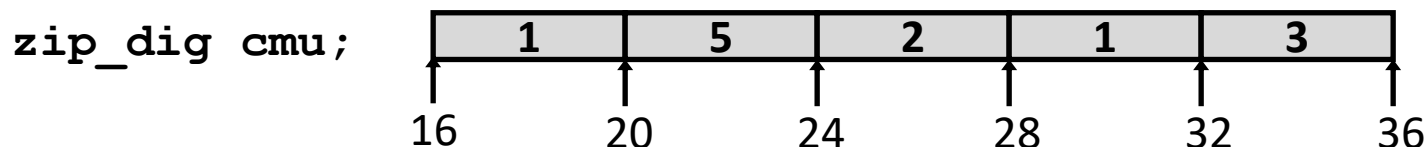
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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 $(\%rdi, \%rsi, 4)$

Array Loop Example

```
void zincr(zip_dig z) {
    size_t i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```
# %rdi = z
movl    $0, %eax
jmp     .L3
.L4:
addl    $1, (%rdi,%rax,4)
addq    $1, %rax
.L3:
cmpq    $4, %rax
jbe     .L4
rep; ret
```


Array Loop Example

```
void zincr(zip_dig z) {
    size_t i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```
# %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 <=, goto loop
rep; ret
```

Multidimensional (Nested) Arrays

■ Declaration

`T A[R][C];`

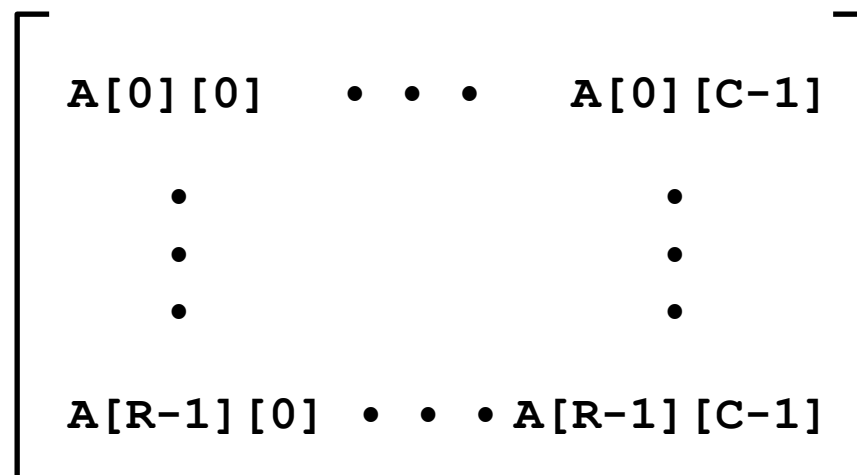
- 2D array of data type T
- R rows, C columns

■ Array Size

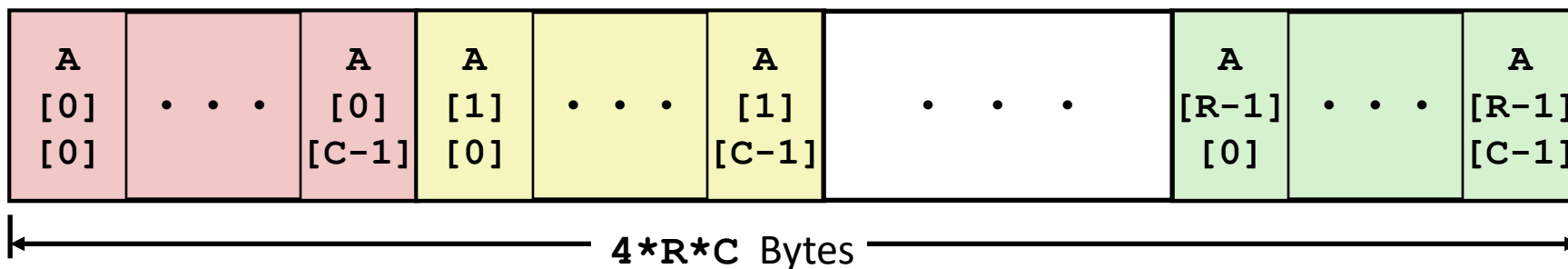
- $R * C * \text{sizeof}(T)$ bytes

■ Arrangement

- Row-Major Ordering



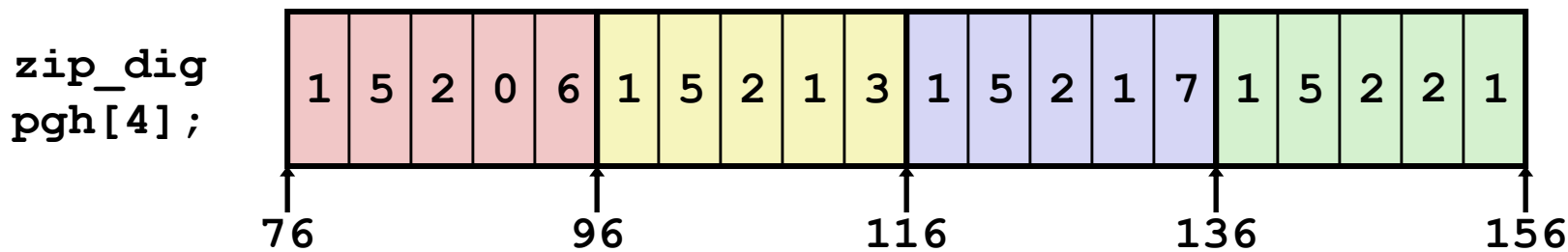
`int A[R][C];`



Nested Array Example

```
#define PCOUNT 4
typedef int zip_dig[5];

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



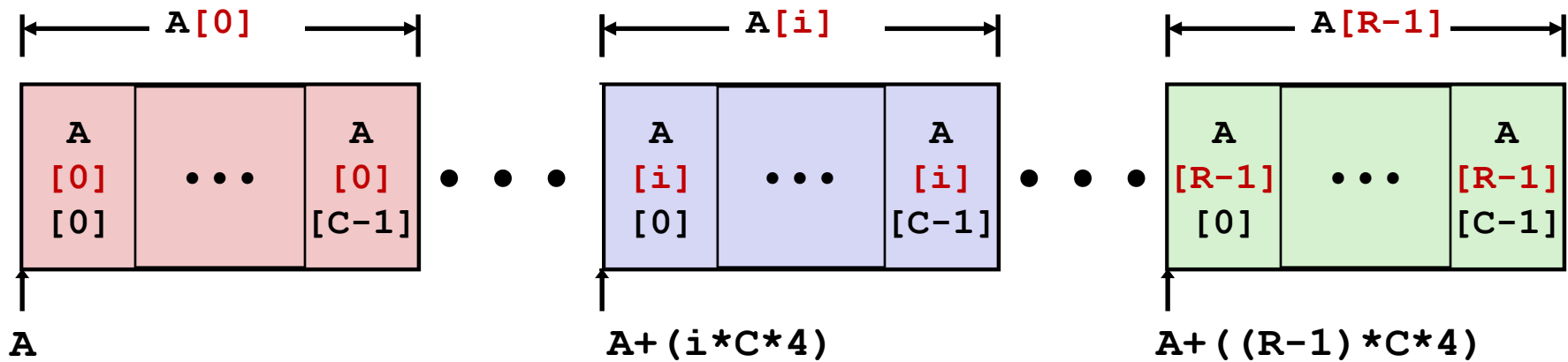
- **“zip_dig pgh[4]” equivalent to “int pgh[4][5]”**
 - Variable **pgh**: array of 4 elements, allocated contiguously
 - Each element is an array of 5 **int**'s, allocated contiguously
- **“Row-Major” ordering of all elements in memory**

Nested Array Row Access

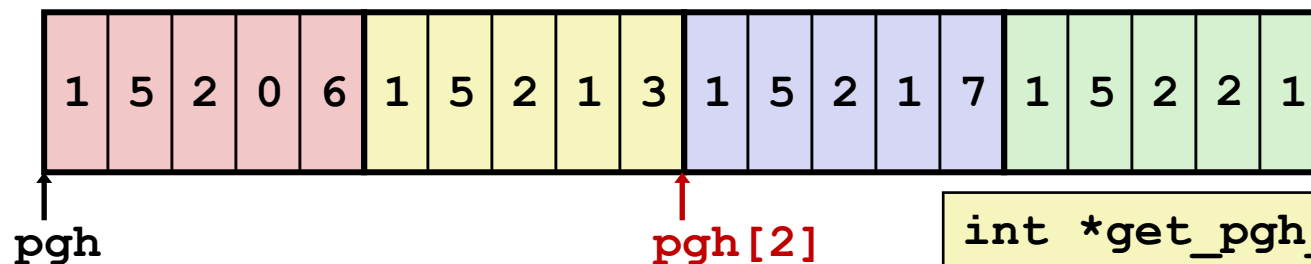
■ Row Vectors

- $A[i]$ is array of C elements of type T
- Starting address $A + i * (C * \text{sizeof}(T))$

```
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

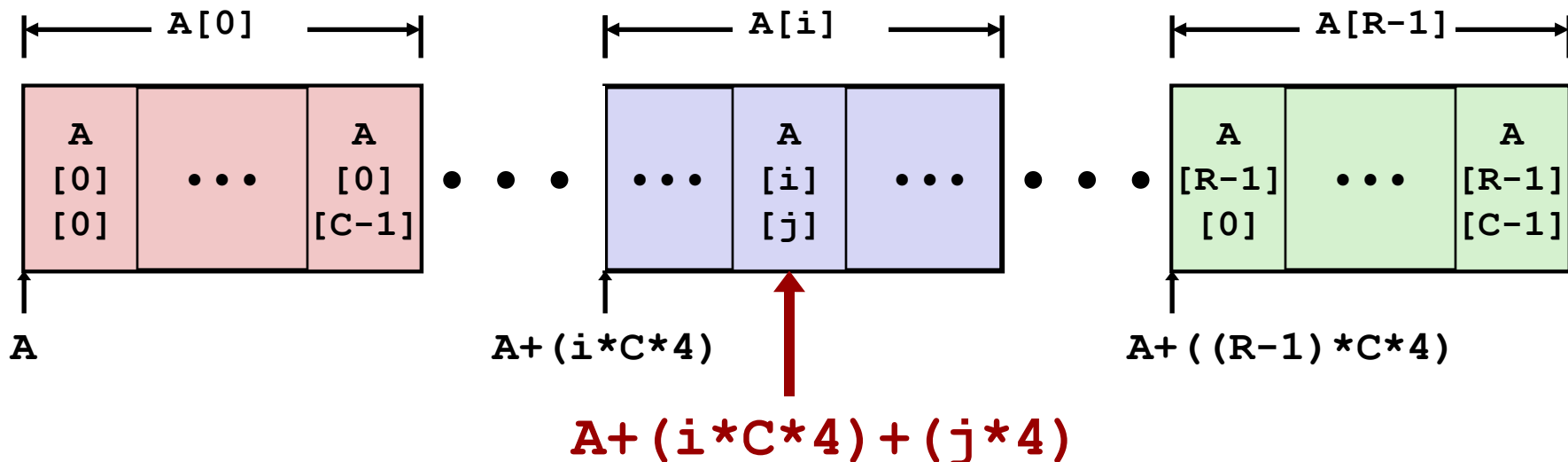
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`

Nested Array Element Access

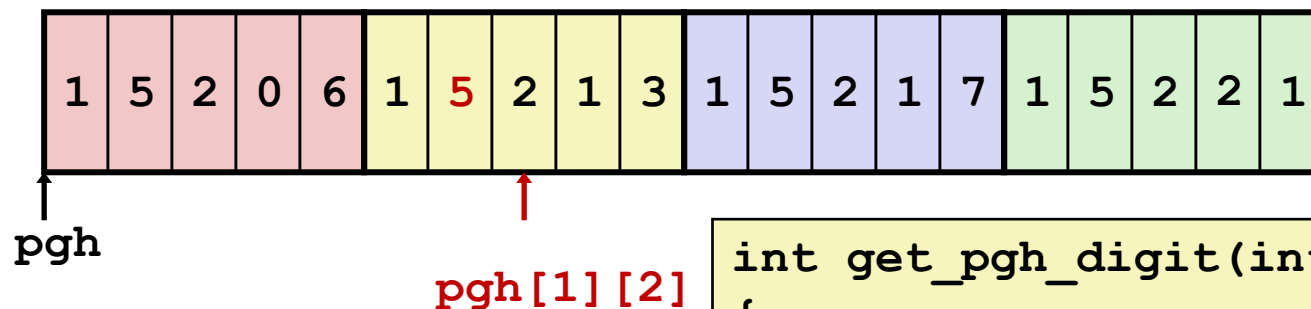
■ Array Elements

- $A[i][j]$ 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

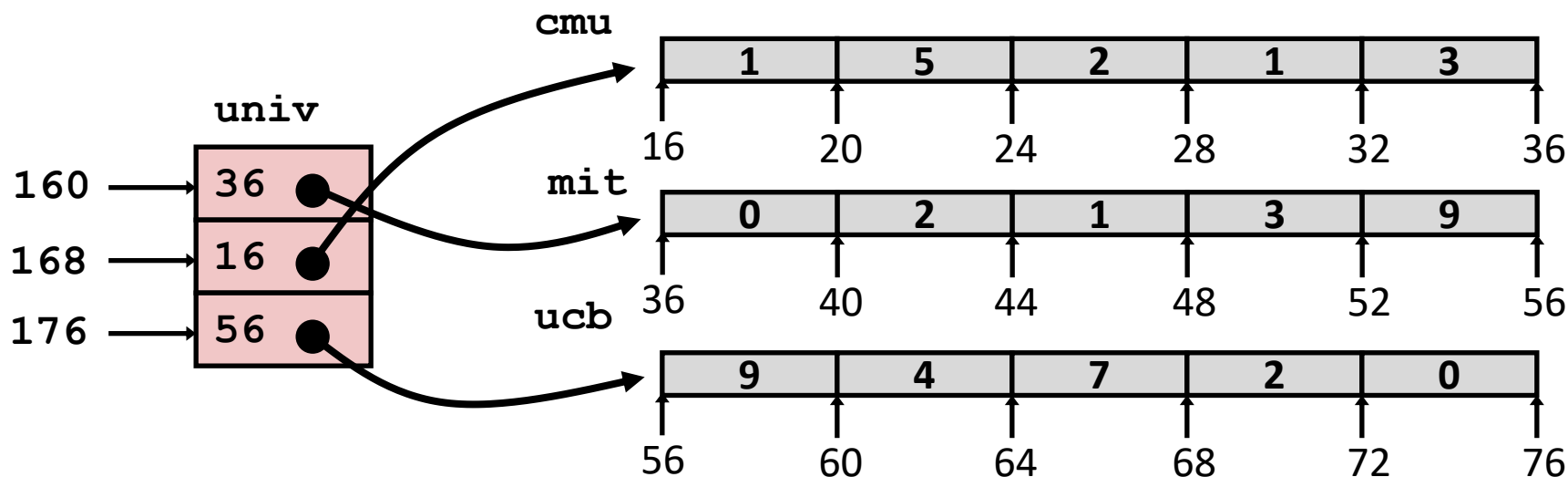
- `pgh[index][dig]` is `int`
- Address: $\text{pgh} + 20 \cdot \text{index} + 4 \cdot \text{dig}$
 $= \text{pgh} + 4 \cdot (5 \cdot \text{index} + \text{dig})$

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 };
```

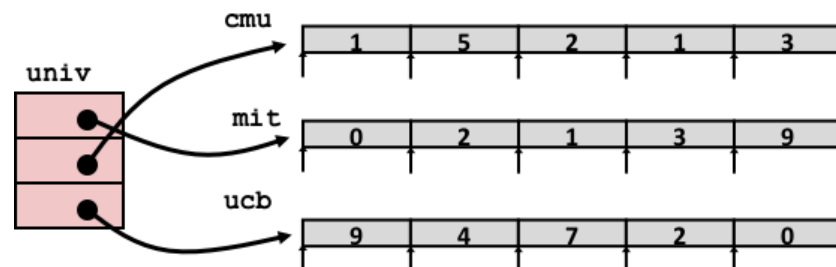
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- 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
```

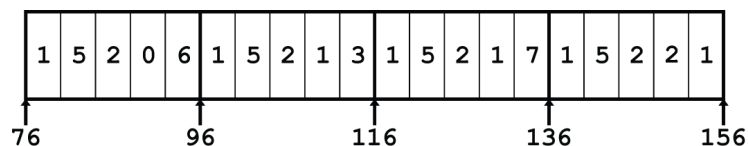
■ Computation

- 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

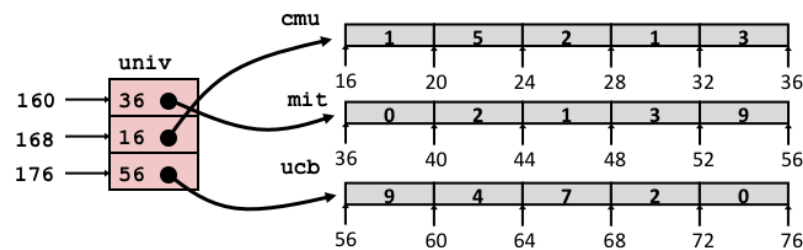
Nested array

```
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

- Not in K&R; added to language in 1999

```
/* 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];
}
```

16 X 16 Matrix Access

■ Array Elements

- `int A[16][16];`
- Address $A + i * (C * K) + j * K$
- $C = 16, K = 4$

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

```
# A in %rdi, i in %rsi, j in %rdx
salq    $6, %rsi           # 64*i
addq    %rsi, %rdi         # A + 64*i
movl    (%rdi,%rdx,4), %eax # Mem[A + 64*i + 4*j]
ret
```

$n \times n$ Matrix Access

■ Array Elements

- `size_t n;`
- `int A[n][n];`
- Address $A + i * (C * K) + j * K$
- $C = n, K = 4$
- Must perform integer multiplication

```

/* 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];
}

```

```

# n in %rdi, A in %rsi, i in %rdx, j in %rcx
imulq    %rdx, %rdi          # n*i
leaq     (%rsi,%rdi,4), %rax  # A + 4*n*i
movl     (%rax,%rcx,4), %eax  # Mem[A + 4*n*i + 4*j]
ret

```

Example: Array Access

```
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
    zip_dig pgh[PCOUNT] =
        {{1, 5, 2, 0, 6},
         {1, 5, 2, 1, 3 },
         {1, 5, 2, 1, 7 },
         {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

```
linux> ./array
```

Example: Array Access

```
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
    zip_dig pgh[PCOUNT] =
        {{1, 5, 2, 0, 6},
         {1, 5, 2, 1, 3 },
         {1, 5, 2, 1, 7 },
         {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

```
linux> ./array
result: 9
```

Today

■ Arrays

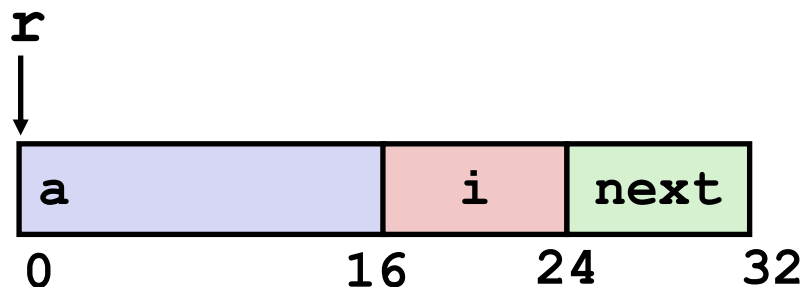
- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structures

- Allocation
- Access
- Alignment

Structure Representation

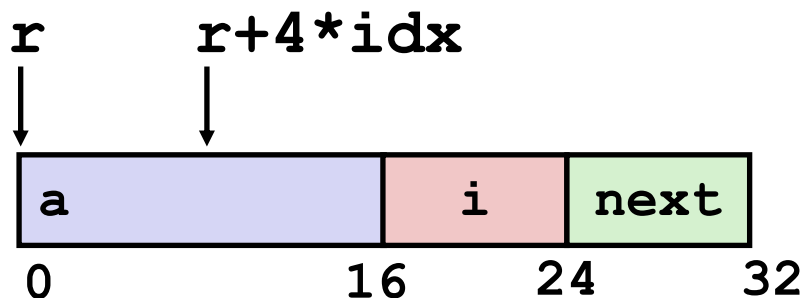
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- **Structure represented as block of memory**
 - Big enough to hold all the fields
- **Fields ordered according to declaration**
 - Even if another ordering could be more compact
- **Compiler determines overall size + positions of fields**
 - In assembly, we see only offsets, not field names

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];
}
```

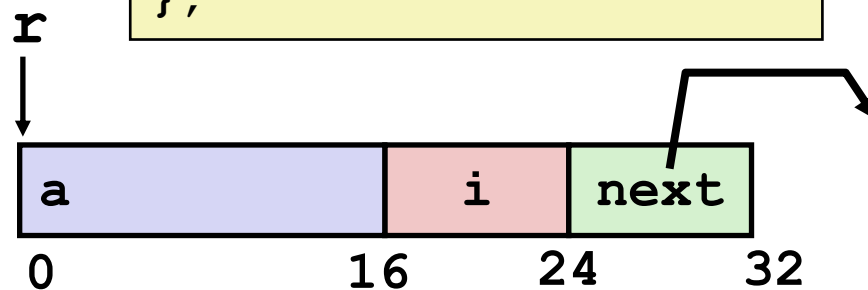
```
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```

Following Linked List #1

■ C Code

```
long length(struct rec*r) {
    long len = 0L;
    while (r) {
        len ++;
        r = r->next;
    }
    return len;
}
```

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



Register	Value
<code>%rdi</code>	<code>r</code>
<code>%rax</code>	<code>len</code>

■ Loop assembly code

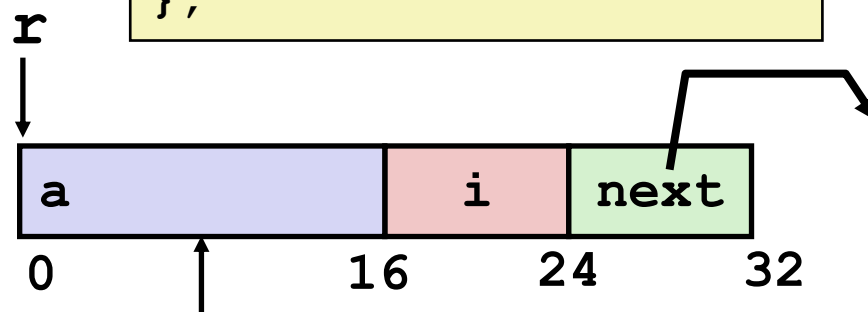
```
.L11:                # loop:
    addq    $1, %rax    # len ++
    movq    24(%rdi), %rdi # r = Mem[r+24]
    testq   %rdi, %rdi  # Test r
    jne     .L11        # If != 0, goto loop
```

Following Linked List #2

■ C Code

```
void set_val
(struct rec *r, int val)
{
    while (r) {
        size_t i = r->i;
        // No bounds check
        r->a[r->i] = val;
        r = r->next;
    }
}
```

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



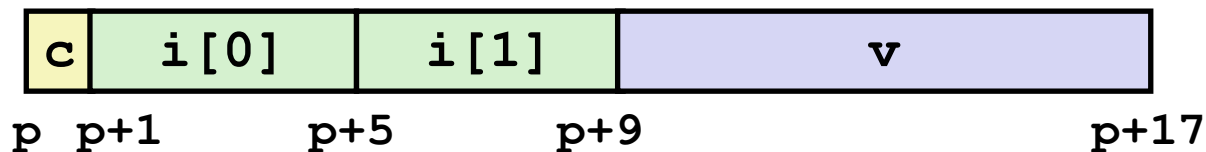
Element i

Register	Value
%rdi	r
%rsi	val

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

Structures & Alignment

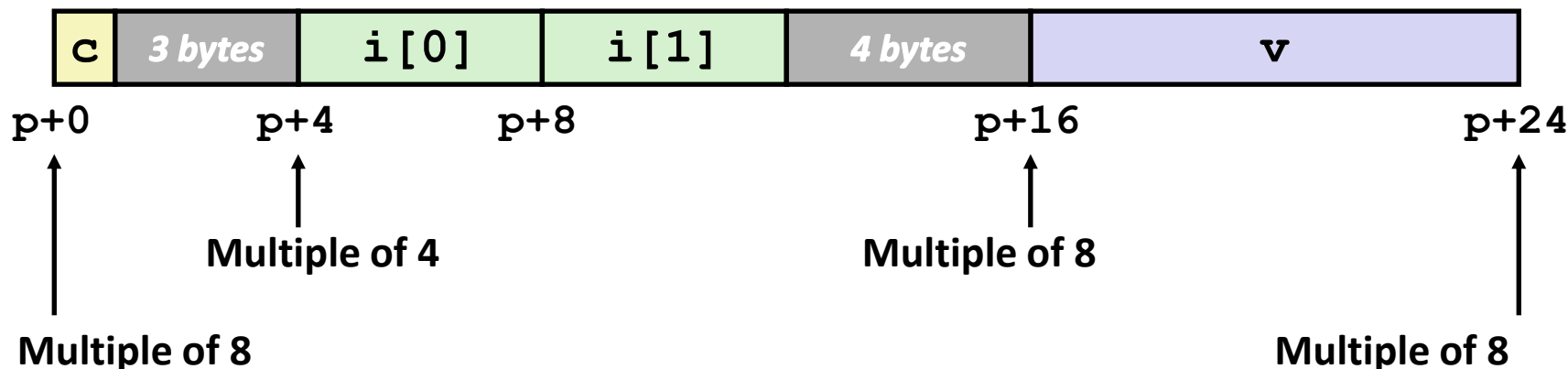
■ Unaligned Data



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

■ Aligned Data

- Primitive data type requires B bytes implies Address must be multiple of B



Alignment Principles

■ Aligned Data

- Primitive data type requires B bytes
- Address must be multiple of B
- 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 cache lines (64 bytes).
Intel states should avoid crossing 16 byte boundaries.

[Cache lines will be discussed in Lecture 10.]

- Virtual memory trickier when datum spans 2 pages (4 KB pages)

[Virtual memory pages will be discussed in Lecture 17.]

■ 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

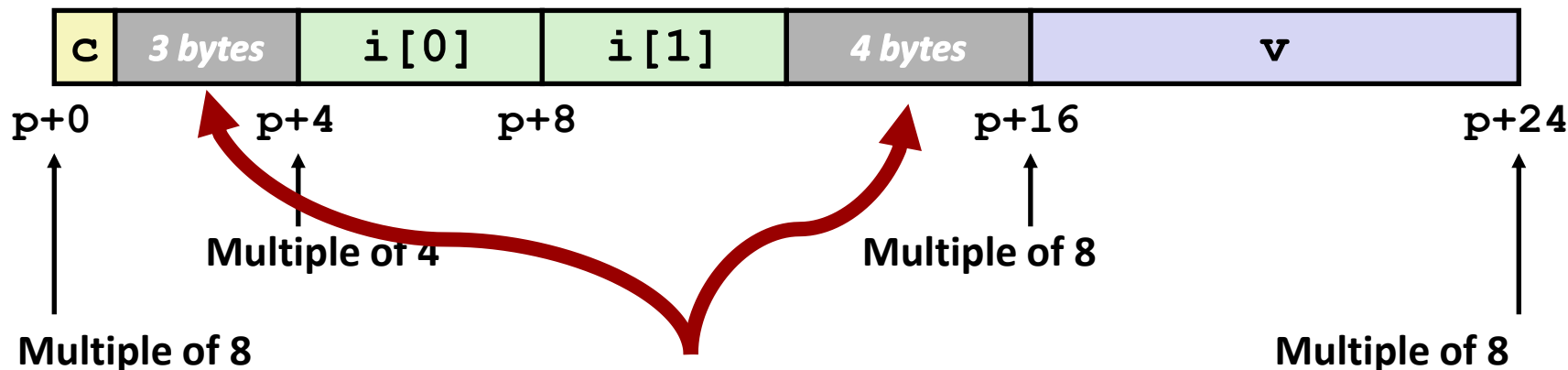
■ Overall structure placement

- 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
NOTE: $K < \text{sizeof}(\text{struct S1})$

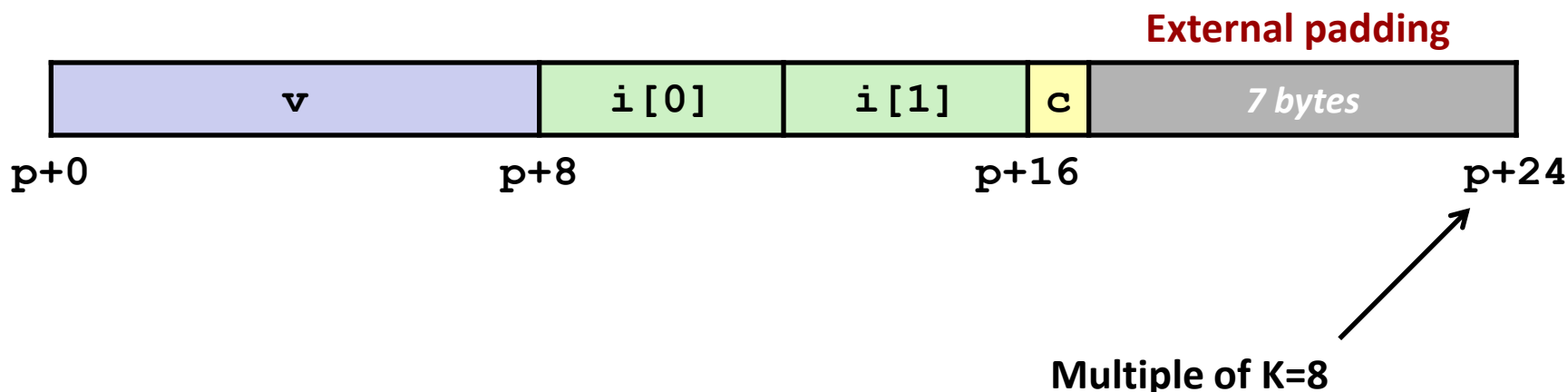


Internal padding

Meeting Overall Alignment Requirement

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

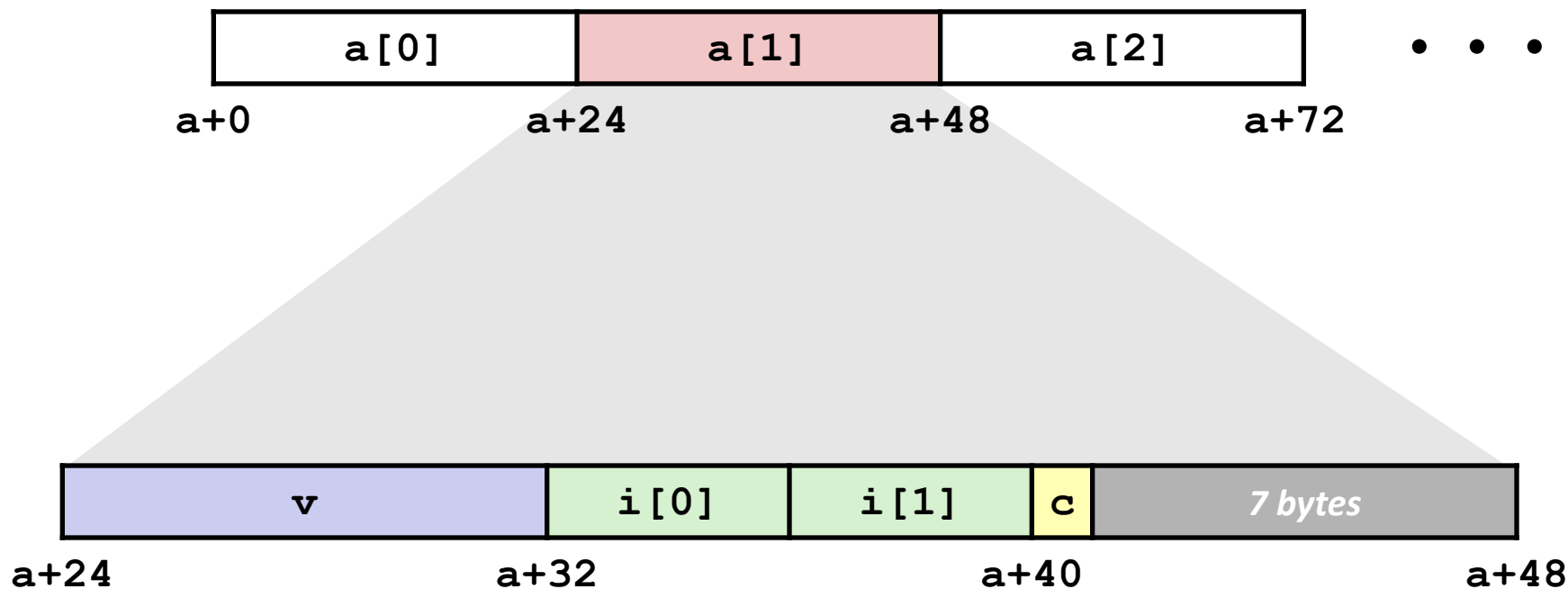
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



Arrays of Structures

- No padding in between array elements
- Overall structure length multiple of K
- Satisfy alignment requirement for every element

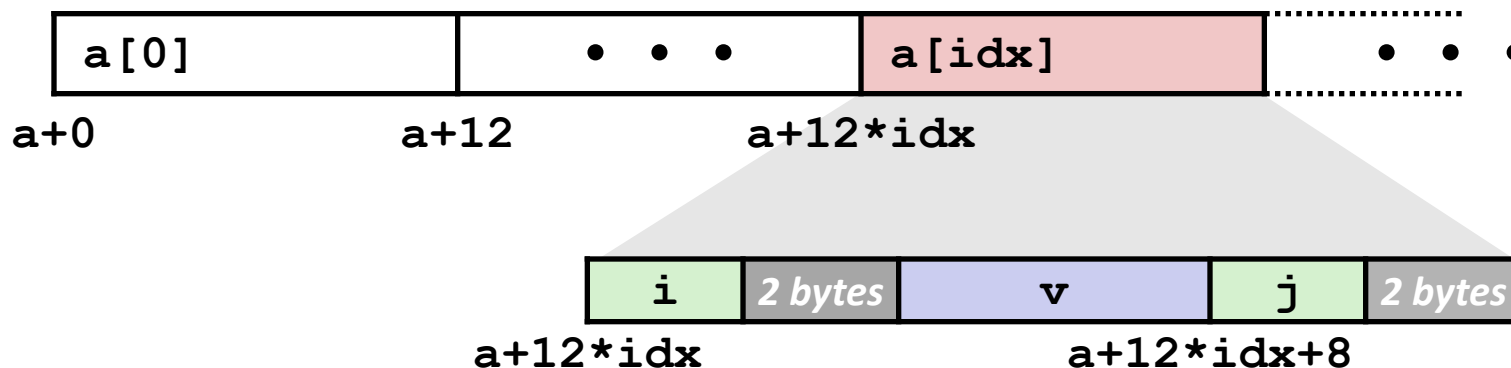
```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```



Accessing Array Elements

- Compute array offset $12 * \text{idx}$
 - `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
```

Saving Space

- Put large data types first

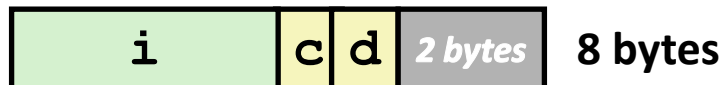
```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```



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



- Effect (largest alignment requirement $K=4$)



Quiz

<https://canvas.cmu.edu/courses/40739/quizzes/123403>

Today

■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structures

- Allocation
- Access
- Alignment

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

■ Floating Point

- Data held and operated on in XMM registers

Today

■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structures

- Allocation
- Access
- Alignment

■ Byte Ordering

Byte Ordering

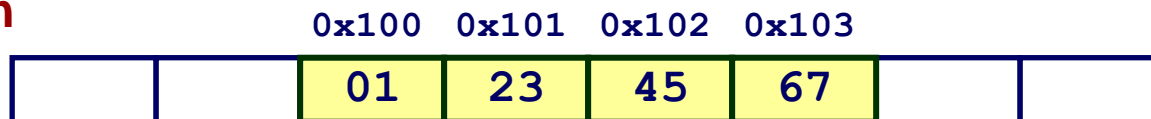
- **So, how are the bytes within a multi-byte word ordered in memory?**
- **Conventions**
 - Big Endian: Sun (Oracle SPARC), PPC Mac, *Internet*
 - Least significant byte has highest address
 - Little Endian: *x86*, ARM processors running Android, iOS, and Linux
 - Least significant byte has lowest address
- **Becomes a concern when data is communicated**
 - Over a network, via files, etc.
- **Important notes**
 - Bits are not reversed, as the low order bit is the reference point.
 - Doesn't affect chars, or strings (arrays of chars), as chars are only one byte

Byte Ordering Example

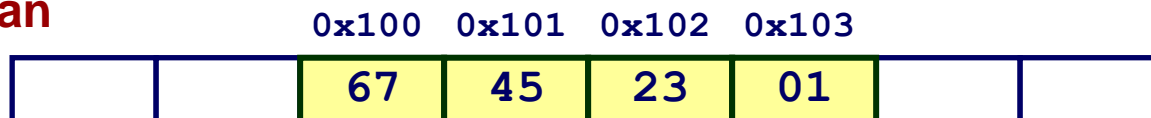
■ Example

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

Big Endian



Little Endian



Reading Byte-Reversed Listings

■ Disassembly

- Text representation of binary machine code
- Generated by program that reads the machine code

■ 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