



# Machine-Level Programming IV: Data

18-213/18-613: Introduction to Computer Systems  
7<sup>th</sup> Lecture, May 30<sup>th</sup>, 2023

# Announcements

- **Lab 2 (bomblab) due Wednesday**
  - Turns in automatically via Autolab
- **Lab 3 (attacklab) out Wednesday**
- **As usual, homework due on Thursdays**
  - Goes out on Tuesdays
  - Live for 1.5 weeks
  - Available on Canvas
- **Lab 0 last hand-in Monday, June 5<sup>th</sup> (Less than one week)**

# Autolab CLI

- **Command-line tool that lets you access some Autolab features, including submitting assignments, from the command-line**
- **`/afs/cs/academic/class/18213-m23/bin/autolabv3`**
  - You might want to alias this or add it to your path so you don't have to type this long thing each time you want to use it.
  - Your group TA can help if you are unfamiliar
- **It needs to be authorized to access your account:**
  - 1) `/afs/cs/academic/class/18213-m23/bin/autolabv3 setup`
  - 2) It produces a 6 letter/number key
  - 3) Feed that key in here: `https://ics.autolabproject.com/activate`
- **You need to activate the CLI for your account even if you don't intend to (directly) use it**
  - Attacklab uses it to make your submissions each time you achieve a phase.

# Today

## ■ Arrays

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

## ■ Structures

- Allocation
- Access
- Alignment

## ■ Floating Point

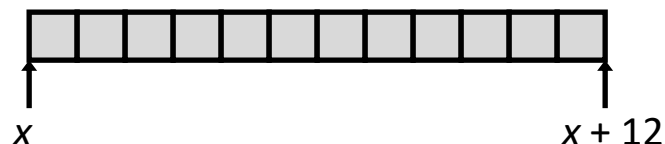
# Array Allocation

## ■ Basic Principle

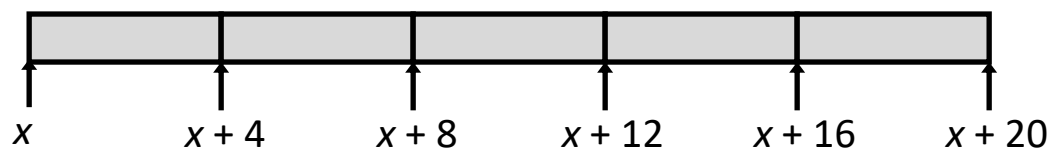
$T$   $A[L]$  ;

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

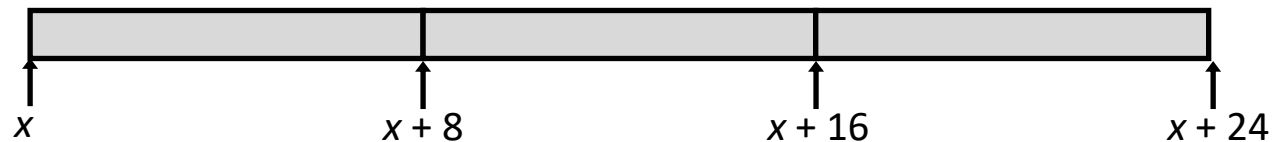
`char string[12];`



`int val[5];`



`double a[3];`



`char *p[3];`

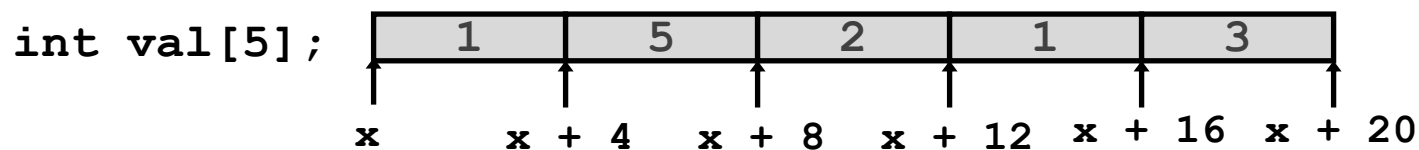


# Array Access

## ■ Basic Principle

$T$   $\mathbf{A}[L]$  ;

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



## ■ Reference                    Type                    Value

`val[4]`                    `int`                    3

`val`                    `int *`

`val+1`                    `int *`

`&val[2]`                    `int *`

`val[5]`                    `int`

`*(val+1)`                    `int`

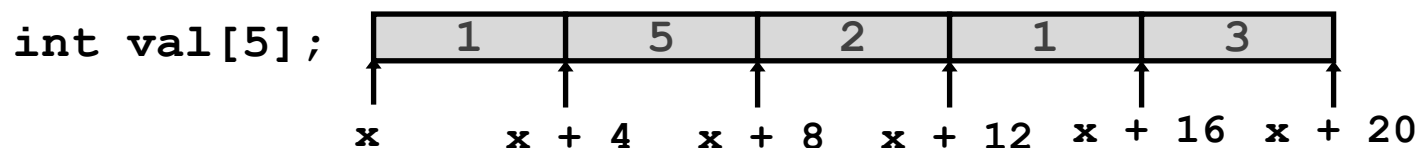
`val + i`                    `int *`

# Array Access

## ■ Basic Principle

$T$   $\mathbf{A}[L]$  ;

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



## ■ Reference

**Type**

**Value**

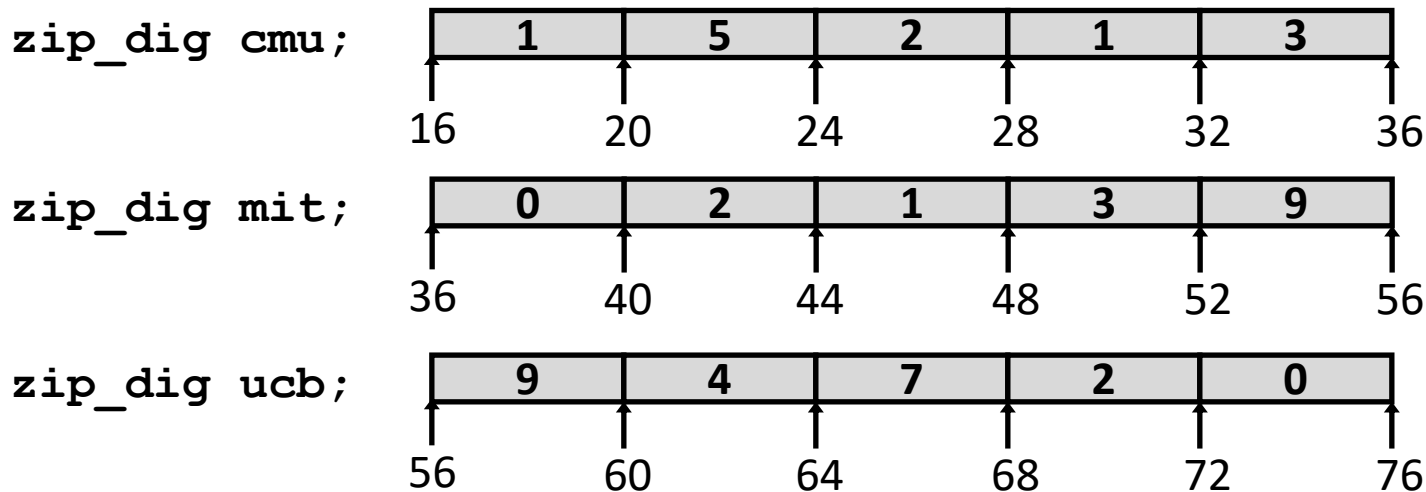
<code>val[4]</code>	<code>int</code>	<code>3</code>	
<code>val</code>	<code>int *</code>	<code>x</code>	
<code>val+1</code>	<code>int *</code>	<code>x + 4</code>	
<code>&amp;val[2]</code>	<code>int *</code>	<code>x + 8</code>	
<code>val[5]</code>	<code>int</code>	<code>??</code>	
<code>*(val+1)</code>	<code>int</code>	<code>5</code>	<code>//val[1]</code>
<code>val + i</code>	<code>int *</code>	<code>x + 4 * i</code>	<code>//&amp;val[i]</code>



# Array Example

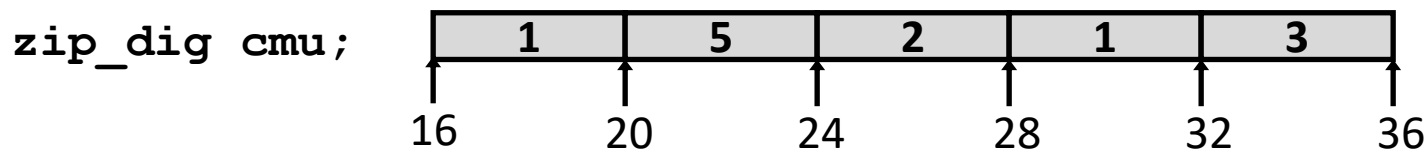
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

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

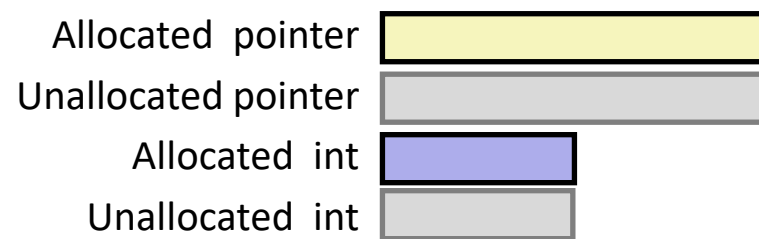
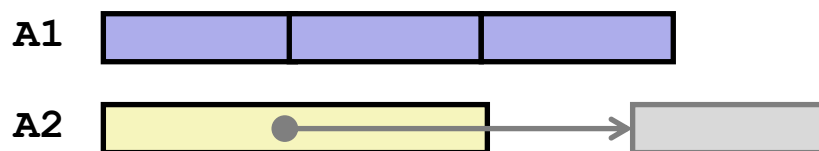
# Array Loop Example

```
#define ZLEN 5
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
```

# Understanding Pointers & Arrays #1

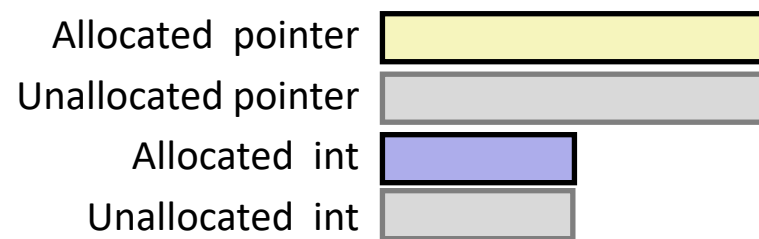
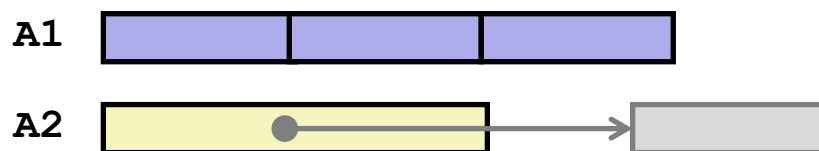
Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
<code>int A1[3]</code>						
<code>int *A2</code>						



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

# Understanding Pointers & Arrays #1

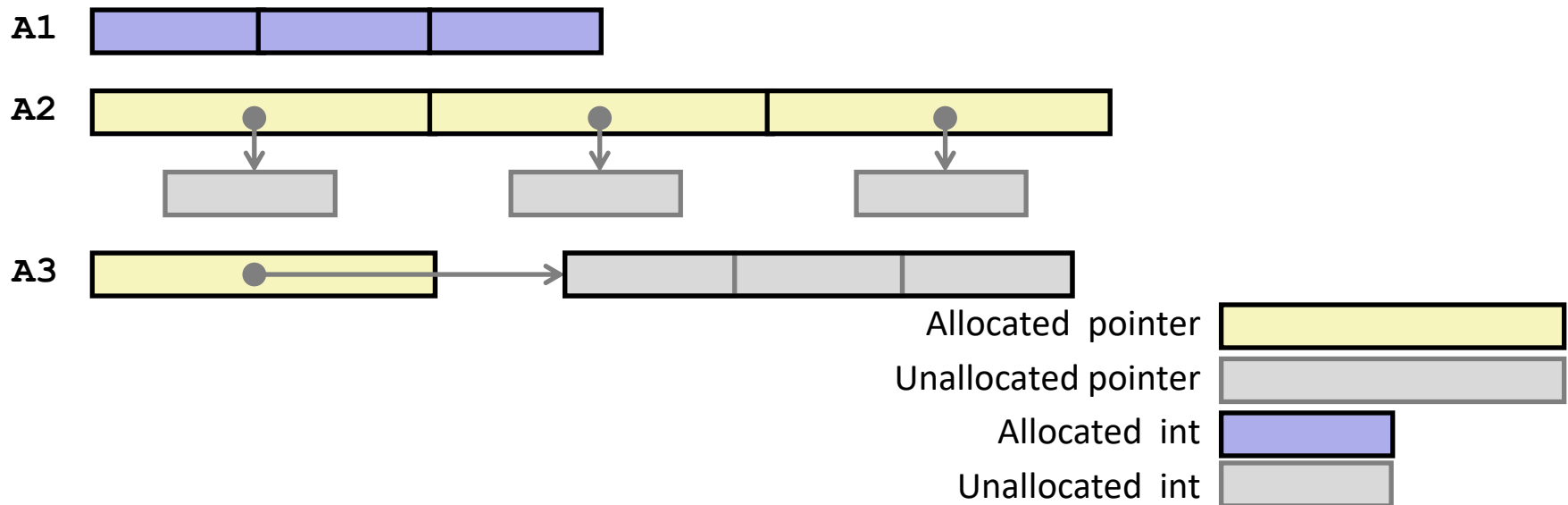
Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
<code>int A1[3]</code>	<b>Y</b>	<b>N</b>	<b>12</b>	<b>Y</b>	<b>N</b>	<b>4</b>
<code>int *A2</code>	<b>Y</b>	<b>N</b>	<b>8</b>	<b>Y</b>	<b>Y</b>	<b>4</b>



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

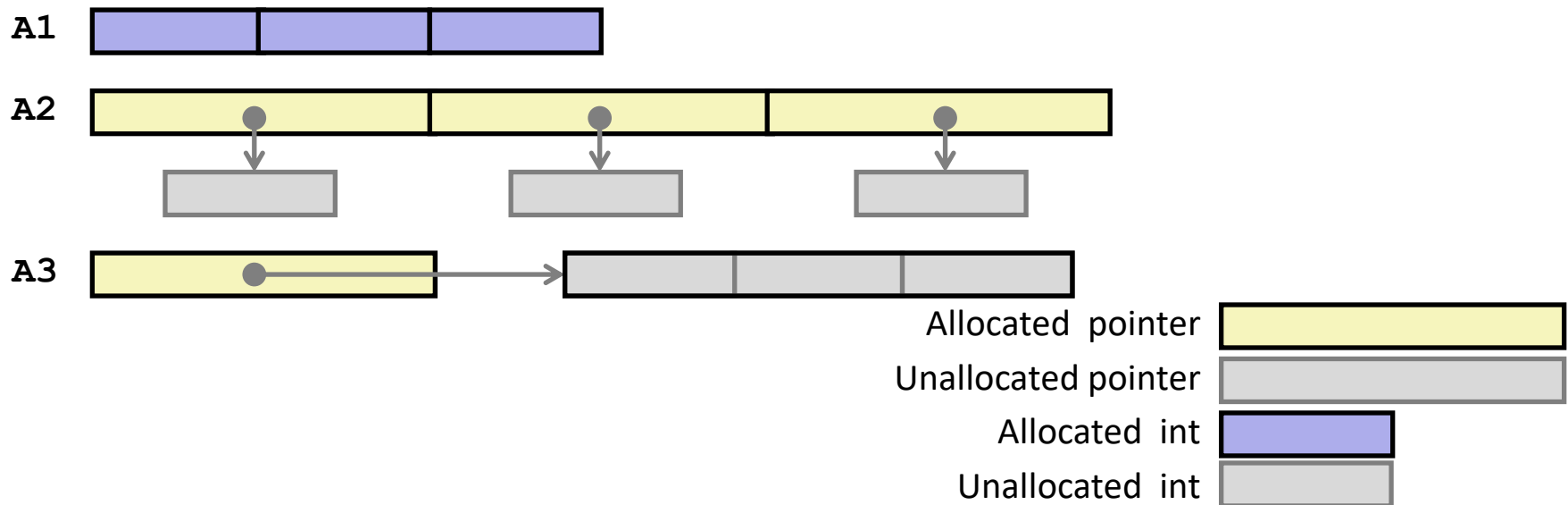
# Understanding Pointers & Arrays #2

Decl	<i>A<sub>n</sub></i>			<i>*A<sub>n</sub></i>			<i>**A<sub>n</sub></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>									



# Understanding Pointers & Arrays #2

Decl	<i>A<sub>n</sub></i>			<i>*A<sub>n</sub></i>			<i>**A<sub>n</sub></i>		
	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



# Multidimensional (Nested) Arrays

## ■ Declaration

`T A[R][C];`

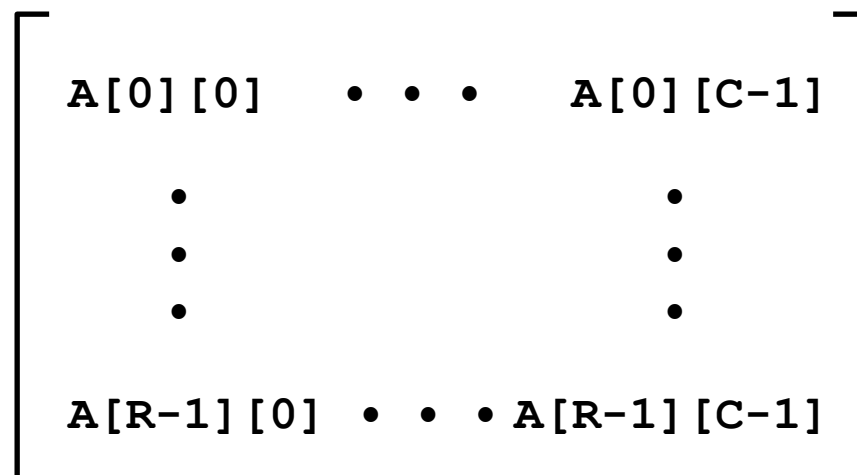
- 2D array of data type  $T$
- $R$  rows,  $C$  columns

## ■ Array Size

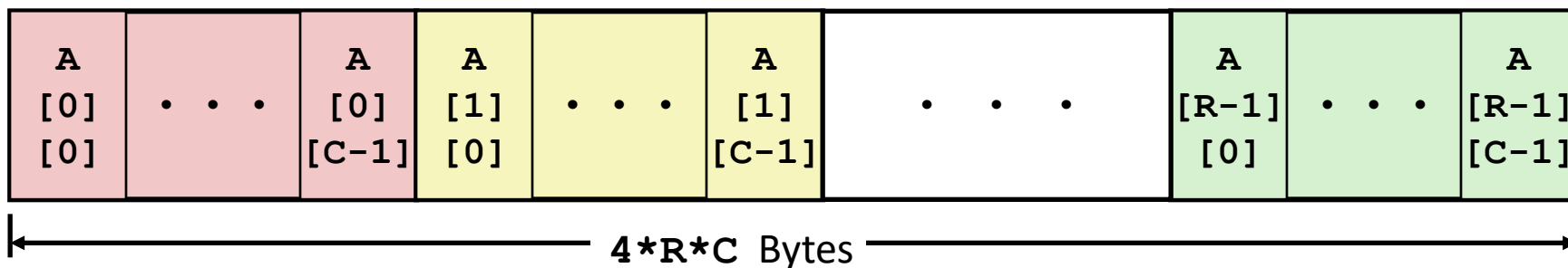
- $R * C * \text{sizeof}(T)$  bytes

## ■ Arrangement

- Row-Major Ordering



`int A[R][C];`

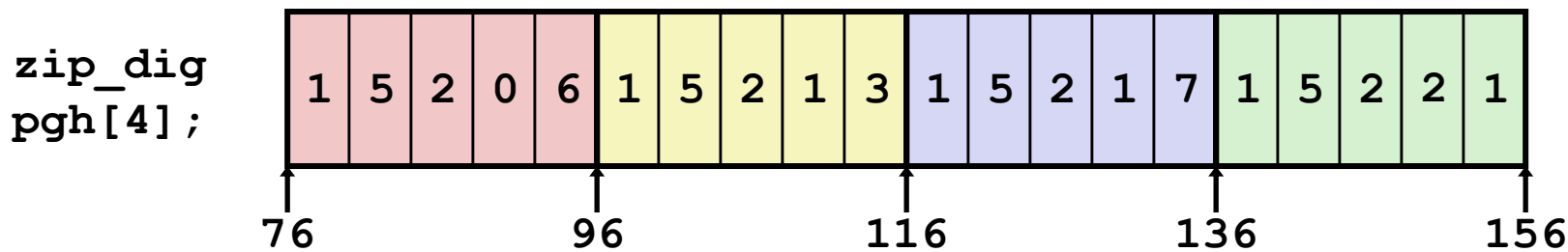




# Nested Array Example

```
#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 }};
```



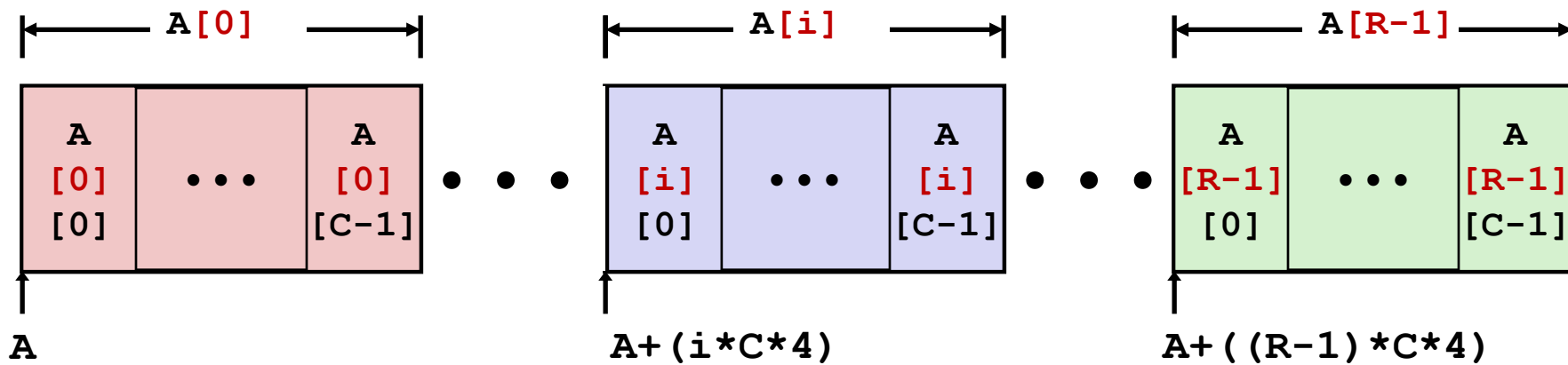
- **“zip\_dig pgh[4]” equivalent to “int pgh[4][5]”**
  - Variable **pgh**: array of 4 elements, allocated contiguously
  - Each element is an array of 5 **int**'s, allocated contiguously
- **“Row-Major” ordering of all elements in memory**

# Nested Array Row Access

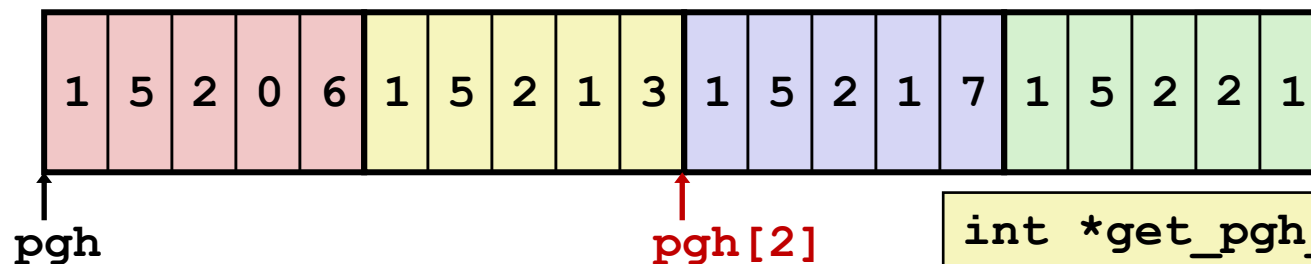
## ■ Row Vectors

- $A[i]$  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

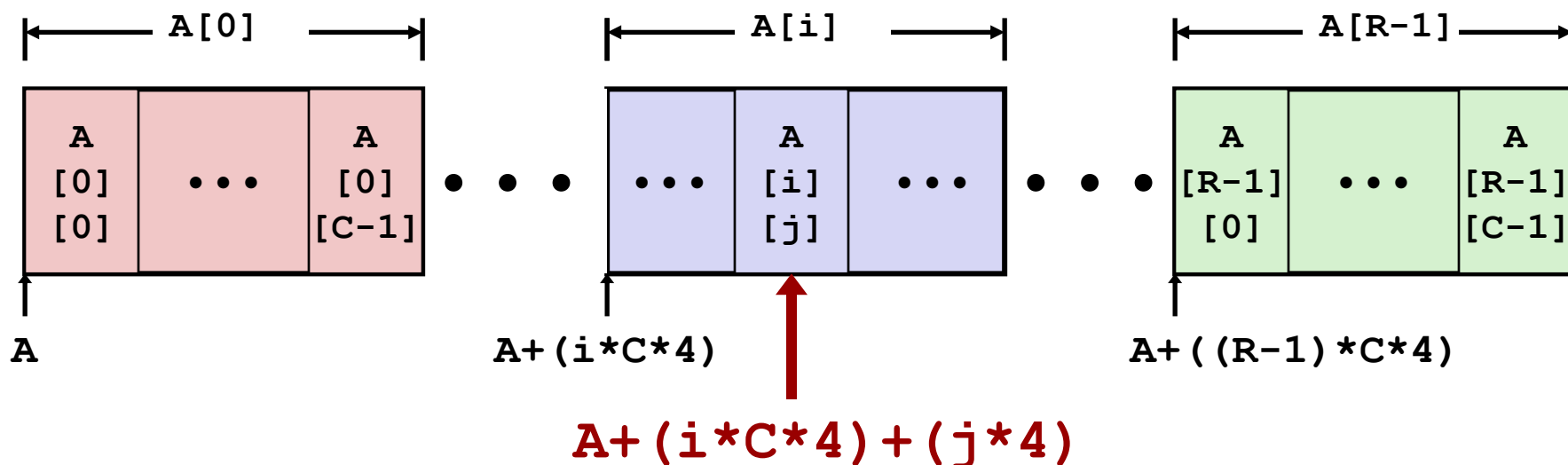
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`

# Nested Array Element Access

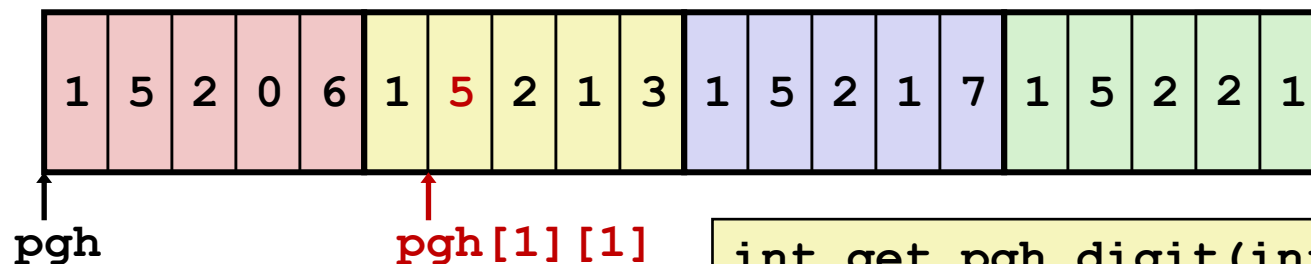
## ■ Array Elements

- $A[i][j]$  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

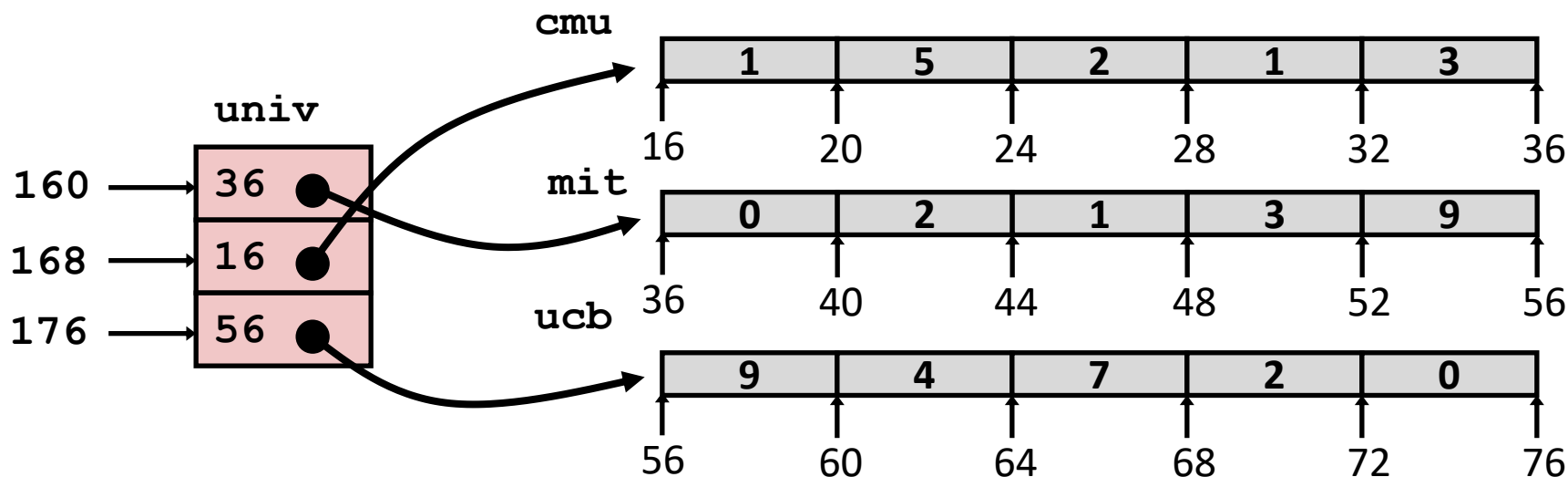
- `pgh[index][dig]` is `int`
- Address:  $\text{pgh} + 20 \cdot \text{index} + 4 \cdot \text{dig}$   
 $= \text{pgh} + 4 \cdot (5 \cdot \text{index} + \text{dig})$

# 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 };
```

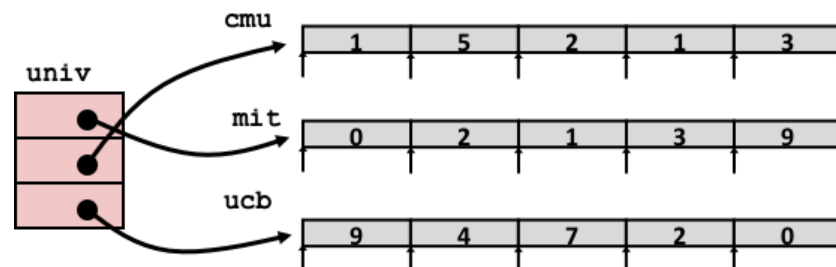
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- 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
```

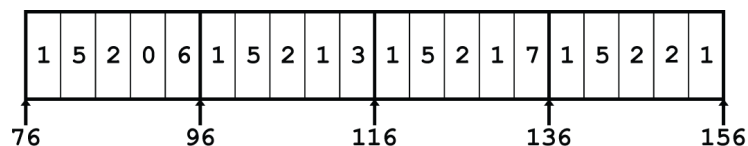
## ■ Computation

- 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

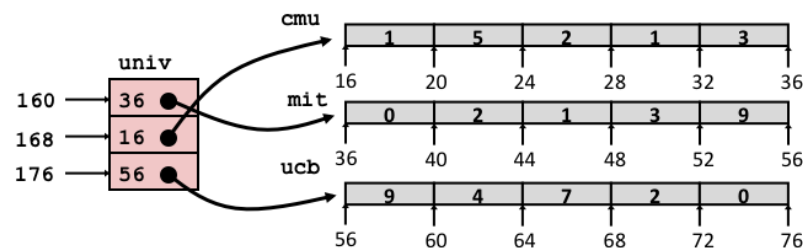
## Nested array

```
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

- Now supported by gcc

```
/* 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   # 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
result: 9
```

# 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

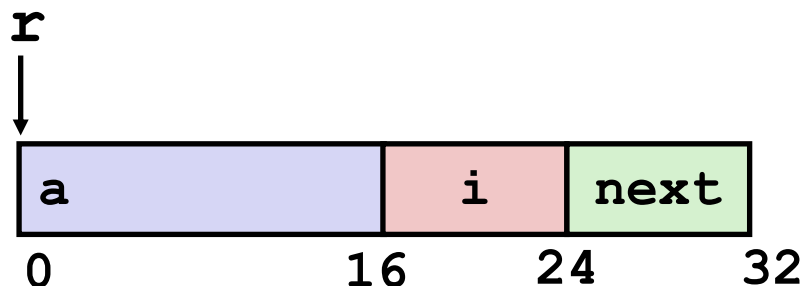
## ■ Structures

- Allocation
- Access
- Alignment

## ■ Floating Point

# Structure Representation

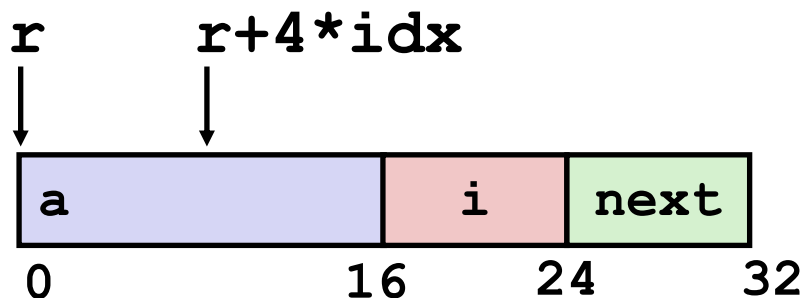
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- **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 (due to alignment rules—coming soon)
- **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
```

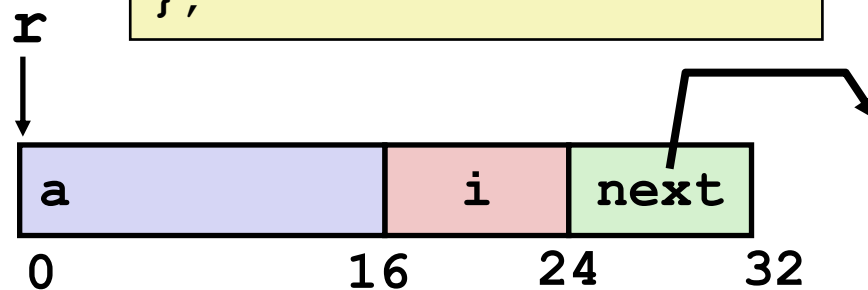


# Following Linked List #1

## ■ 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

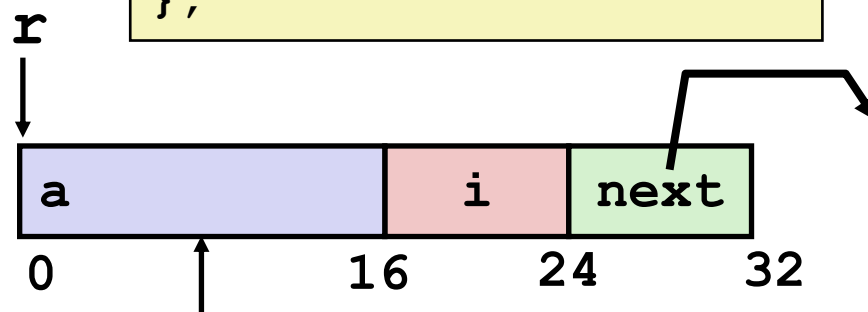
```
.L11:                # loop:
    addq    $1, %rax    # len ++
    movq    24(%rdi), %rdi # r = Mem[r+24]
    testq   %rdi, %rdi  # Test r
    jne     .L11        # If != 0, goto loop
```

# 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[i] = val;
        r = r->next;
    }
}
```

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



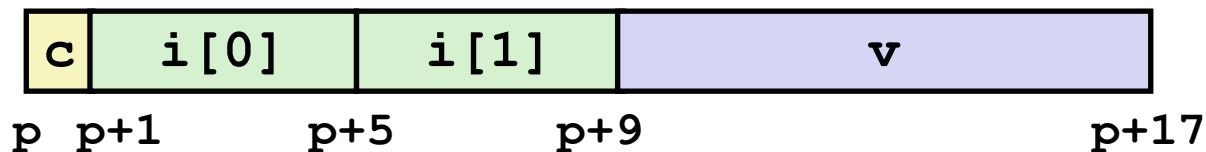
Element *i*

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

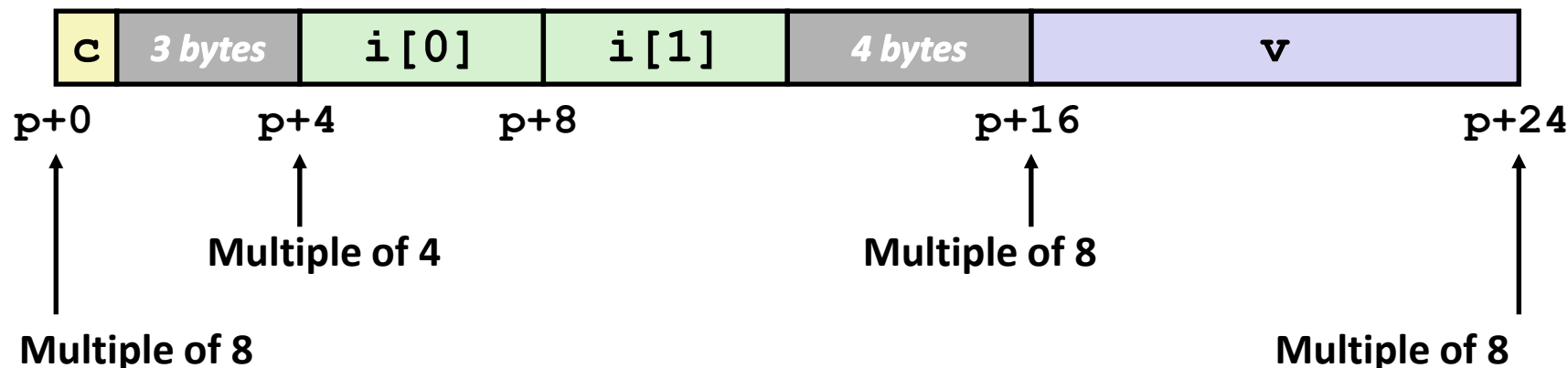
## ■ Unaligned Data



```
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 11.]*

- 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

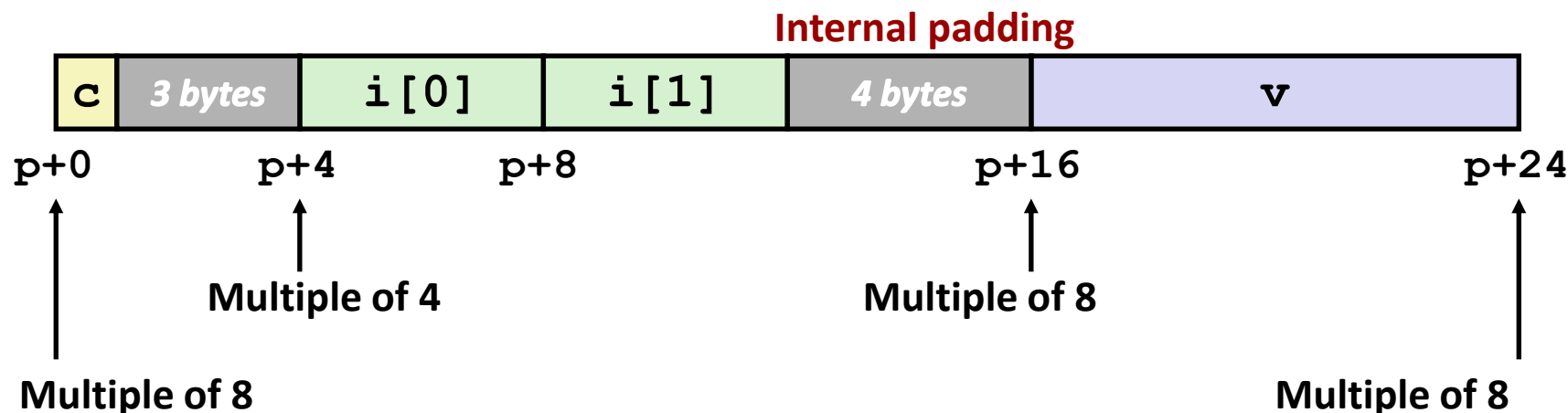
## ■ Overall structure placement

- 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:

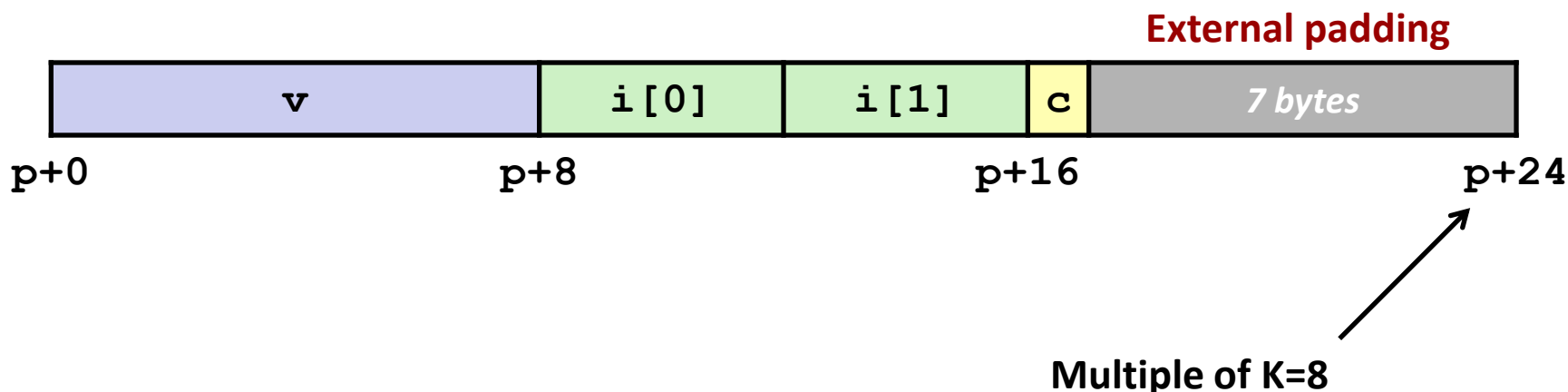
- $K = 8$ , due to `double` element



# Meeting Overall Alignment Requirement

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

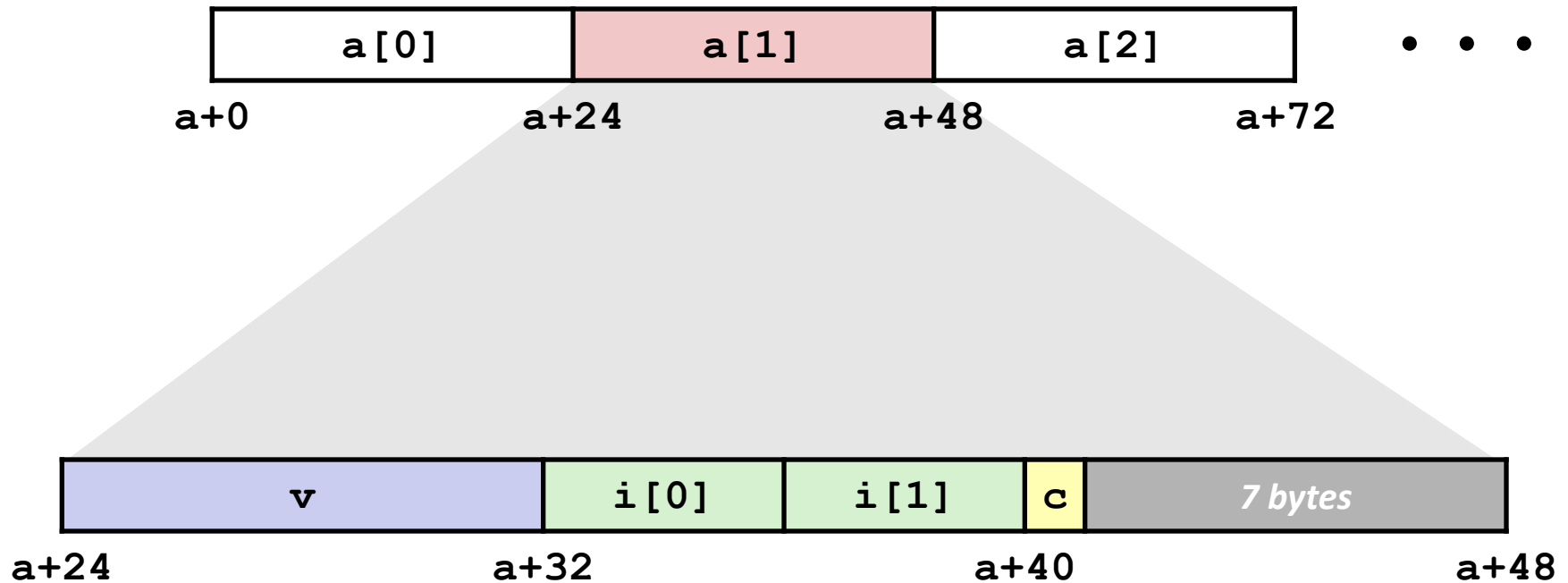
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} *p;
```



# Arrays of Structures

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

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

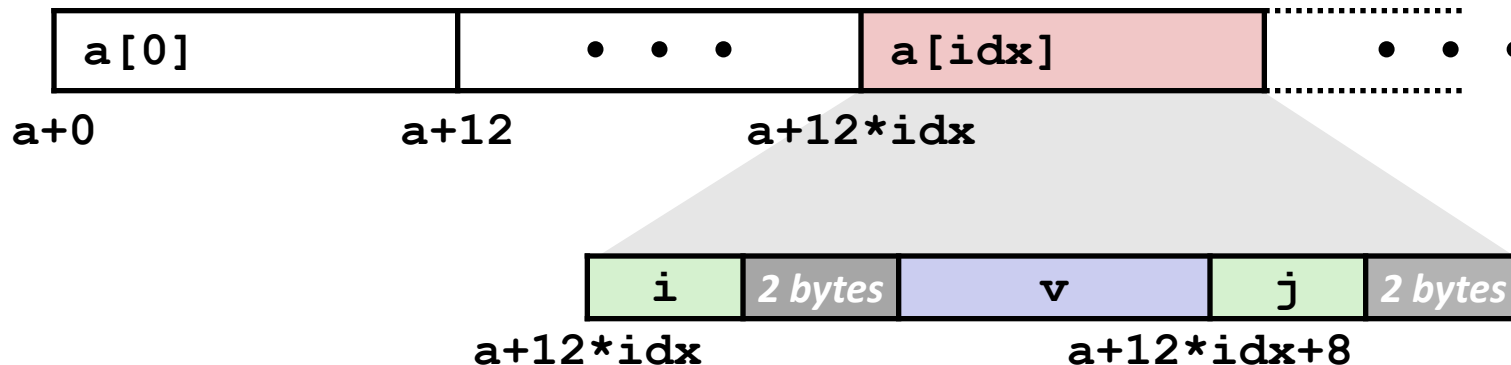




# Accessing Array Elements

- 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

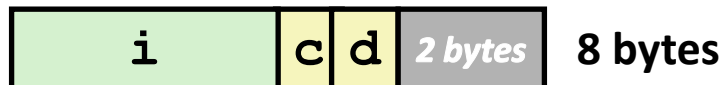
```
struct S4 {
    char c;
    int i;
    char d;
} *p;
```



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



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



# Example Struct Exam Question

## Problem 5. (8 points):

*Struct alignment.* Consider the following C struct declaration:

```
typedef struct {
    char a;
    long b;
    float c;
    char d[3];
    int *e;
    short *f;
} foo;
```

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

<http://www.cs.cmu.edu/~213/oldexams/exam1-f12.pdf>

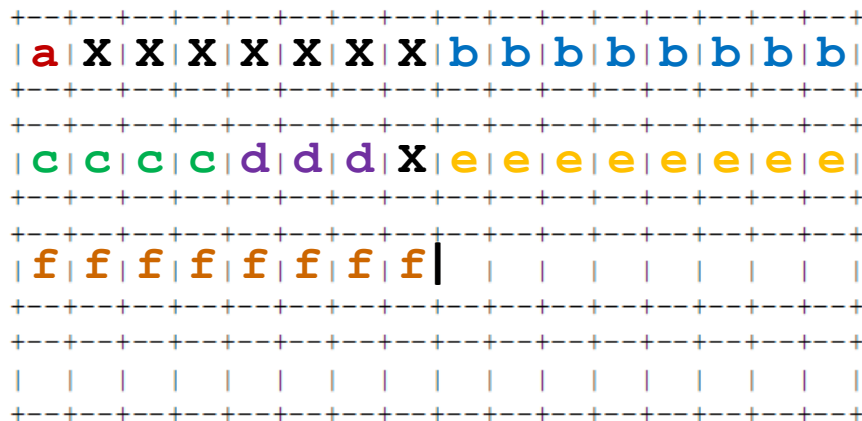
# Example Struct Exam Question

## Problem 5. (8 points):

*Struct alignment.* Consider the following C struct declaration:

```
typedef struct {
    char a;
    long b;
    float c;
    char d[3];
    int *e;
    short *f;
} foo;
```

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and clearly mark the end of the struct. Use an X to denote space that is allocated in the struct as padding.



<http://www.cs.cmu.edu/~213/oldexams/exam1-f12.pdf>

# Example Struct Exam Question (Cont'd)

## Problem 5. (8 points):

*Struct alignment.* Consider the following C struct declaration:

```
typedef struct {
    char a;
    long b;
    float c;
    char d[3];
    int *e;
    short *f;
} foo;
```

2. Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and **clearly mark the end of the struct**. Use an X to denote space that is allocated in the struct as padding.

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| | | | | | | | | | | | | | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| | | | | | | | | | | | | | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| | | | | | | | | | | | | | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| | | | | | | | | | | | | | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

<http://www.cs.cmu.edu/~213/oldexams/exam1-f12.pdf>

# Example Struct Exam Question (Cont'd)

## Problem 5. (8 points):

*Struct alignment.* Consider the following C struct declaration:

```
typedef struct {
    char a;
    long b;
    float c;
    char d[3];
    int *e;
    short *f;
} foo;
```

- Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and clearly mark the end of the struct. Use an X to denote space that is allocated in the struct as padding.

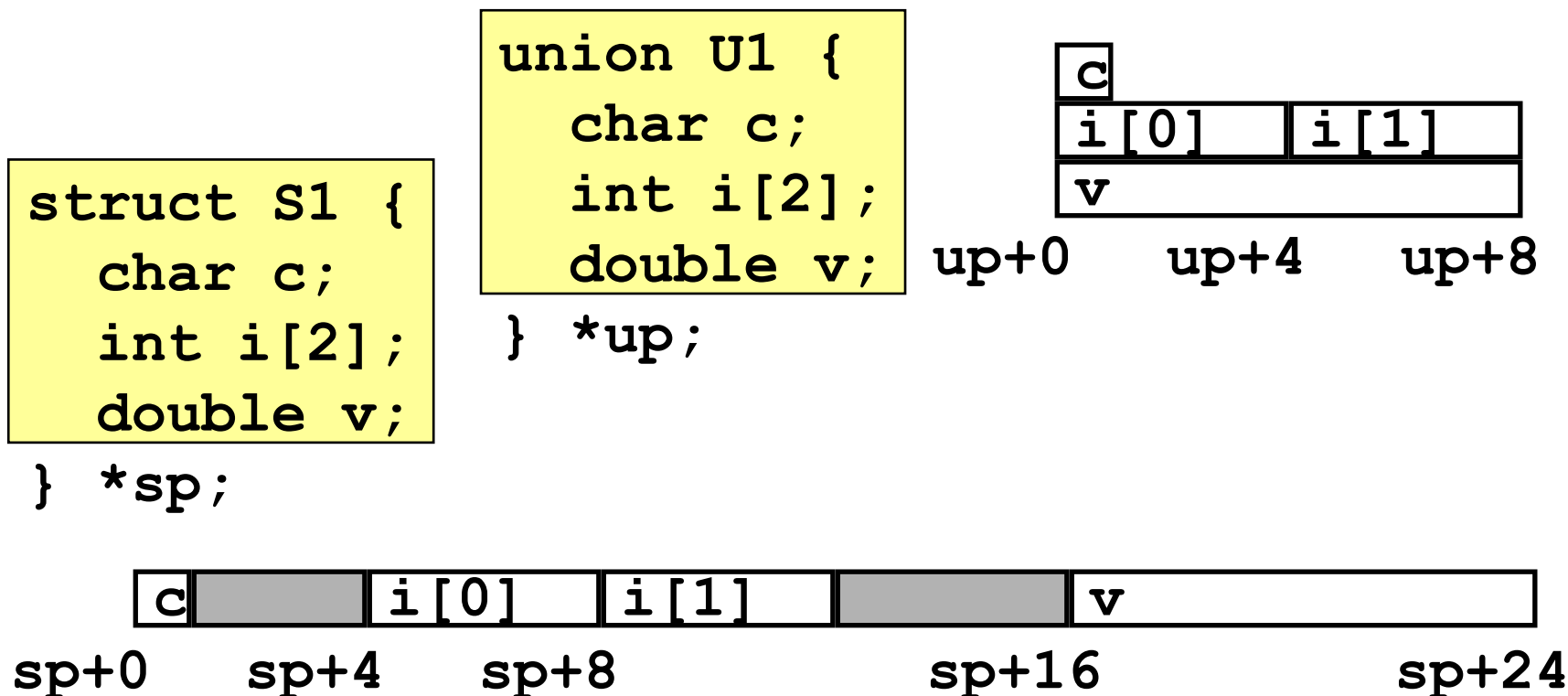
```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| a | d | d | d | c | c | c | c | b | b | b | b | b | b | b | b |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| e | e | e | e | e | e | e | e | f | f | f | f | f | f | f | f |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| | | | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| | | | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

<http://www.cs.cmu.edu/~213/oldexams/exam1-f12.pdf>

# Union Allocation

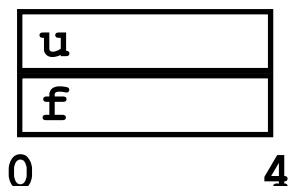
## ■ Principles

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



## Using Union to Access Bit Patterns

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



```
float bit2float(unsigned
u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}
```

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

- Get direct access to bit representation of `bit_float_t` float
- `bit2float` generates float with given bit pattern
  - NOT the same as `(float) u`
- `float2bit` generates bit pattern from float
  - NOT the same as `(unsigned) f`