



**CS 240**

Foundations of Computer Systems



# Representing Data Structures

Multidimensional arrays

C structs

# Outline

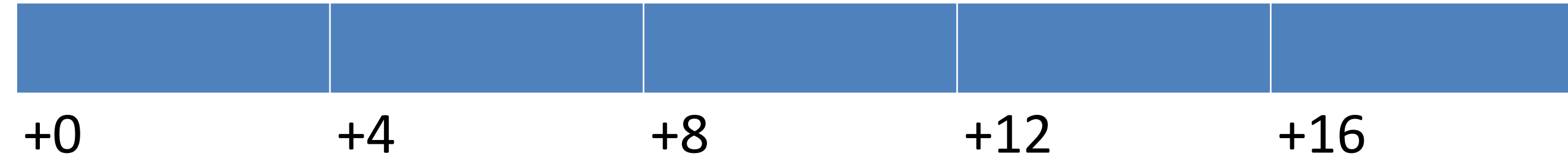
*Goal:* understand how we represented structured data in C and x86

- Arrays in x86
  - Array indexing
  - Arrays of pointers to arrays
  - 2-dimensional arrays (defer details to video)
- C structs (simpler version of objects)
  - Overview and accessing fields
  - Alignment
  - LinkedList example

# C: Array layout and indexing

ex

```
int val[5];
```



Recall:

- Array layout will be contiguous block of memory
- The base address will be aligned based on the element type: here, a multiple of 4

For:  $T$   $a[N]$

Address of  $a[i]$  is:

$a + i * \text{sizeof}(T)$

Write x86 code to load  $val[i]$  into  $\%eax$ .

1. Assume:

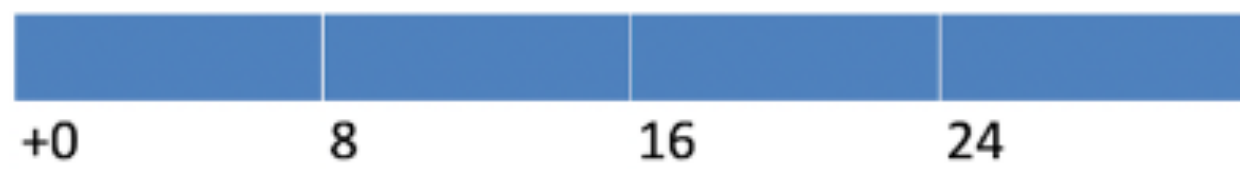
- Base address of  $val$  is in  $\%rdi$
- $i$  is in  $\%rsi$

2. Assume:

- Base address of  $val$  is 28 ( $\%rsp$ )
- $i$  is in  $\%rcx$

Which expression correctly loads val[i] into %rax? Assume val is in %rdi and i is in %rsi.

```
long val[4];
```



`movq (%rsi,%rdi,4), %rax`

`movq (%rdi,%rsi,4), %rax`

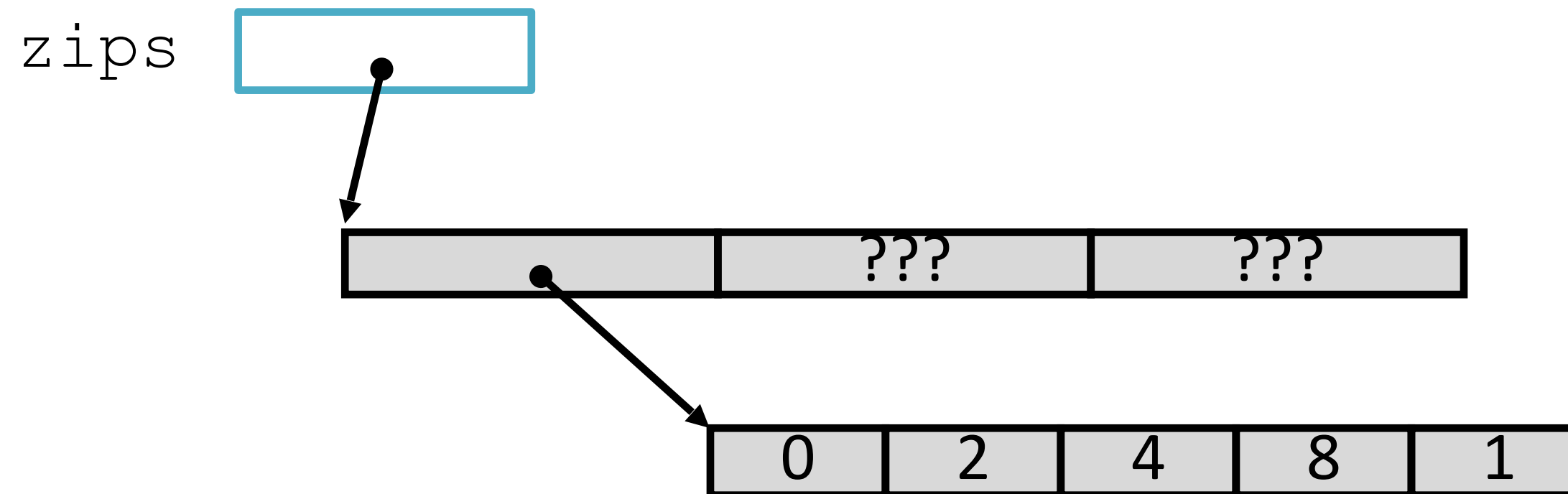
`movq (%rdi,%rsi,8), %rax`

`movq (%rsi,%rdi,8), %rax`

None of the above

# C: Arrays of pointers to arrays of ...

```
int** zips = (int**)malloc(sizeof(int*)*3);  
...  
zips[0] = (int*)malloc(sizeof(int)*5);  
...  
int* zip0 = zips[0];  
zip0[0] = 0;  
zips[0][1] = 2;  
zips[0][2] = 4;  
zips[0][3] = 8;  
zips[0][4] = 1;
```

**C**

```
int[][] zips = new int[3][];  
zips[0] = new int[5] {0, 2, 4, 8, 1};
```

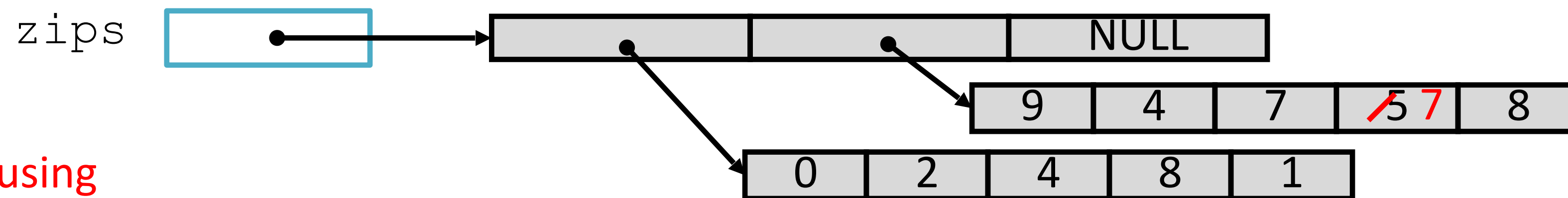
**Java**

# C: Arrays of pointers to arrays in x86

ex

```
void copyfromleft(int** zips, long i, long j) {  
    zipCodes[i][j] = zipCodes[i][j - 1];  
}
```

copyleft(zips, 1, 3)

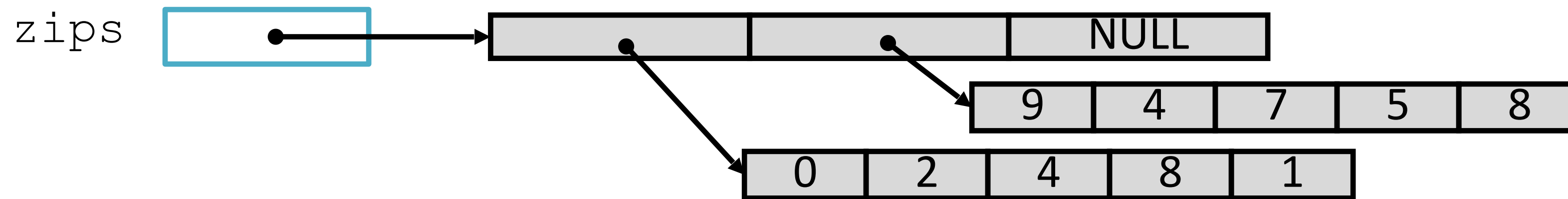


Goal: translate to x86, using  
two scratch registers

`%rax, %ecx` (why 32 bits?)

1. Put `zips[i]` in a reg
2. Access element `[j-1]`
3. Set element `[j]`
4. Return (nothing)

# C: Arrays of pointers to arrays: Pros/Cons



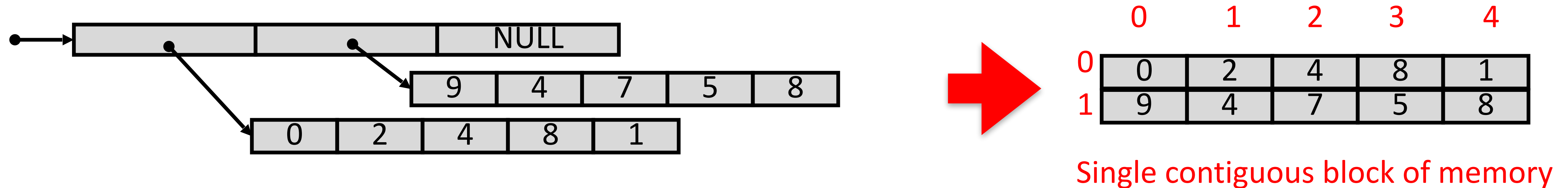
Pros:

- Flexible array lengths
- Different elements can be different lengths
- Lengths can change as the program runs
- Representation of empty elements saves space

Cons:

- Accessing a nested element requires multiple memory operations

# Alternative: row-major nested arrays



Pros:

- Accessing nested elements now a single memory operation!
- Calculations can be done ahead of time, via arithmetic

Cons:

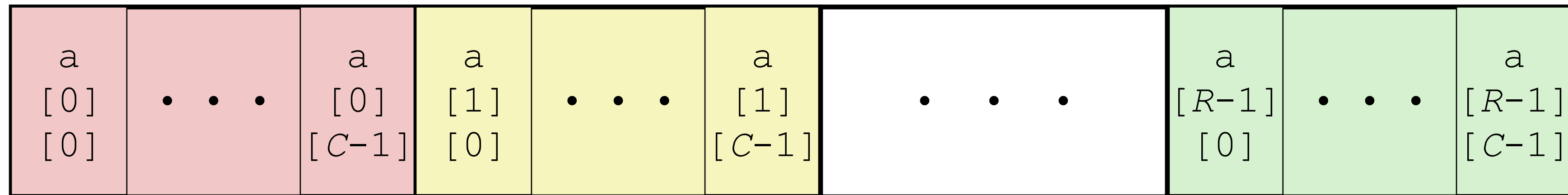
- Less space efficient depending on the shape of the data
- Need to be careful with our order of indexing!



# C: Row-major nested arrays

$$\begin{bmatrix} a[0][0] & \cdot & \cdot & \cdot & a[0][C-1] \\ \vdots & & & & \vdots \\ a[R-1][0] & \cdot & \cdot & \cdot & a[R-1][C-1] \end{bmatrix}$$

```
int a[R][C];
```



Suppose `a`'s base address is `A`.

`&a[i][j]` is  $\underline{A + C \times \text{sizeof}(\text{int}) \times i + \text{sizeof}(\text{int}) \times j}$   
*(regular unscaled arithmetic)*

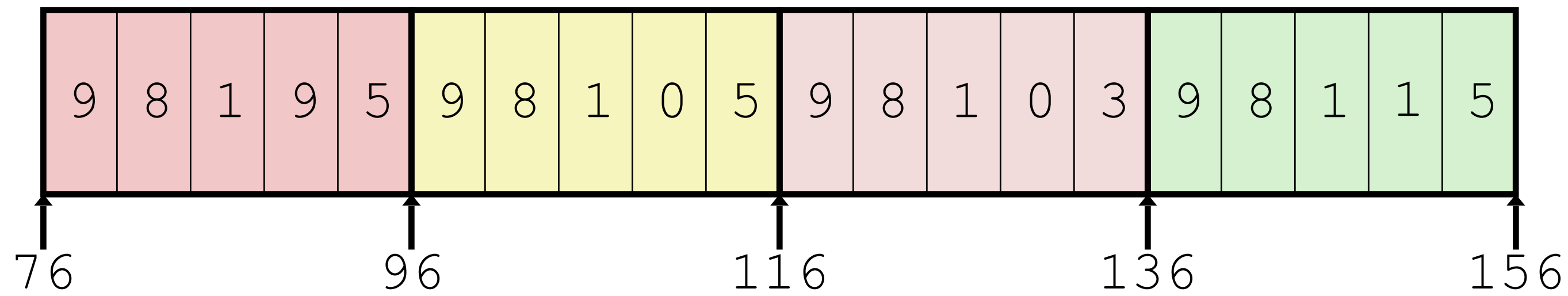
```
int* b = (int*)a; // Treat as larger 1D array
```

```
&a[i][j] == &b[ C*i + j ]
```

# C: Strange array indexing examples



```
int sea[4][5];
```



Reference	Address	Value
sea[3][3]		
sea[2][5]		
sea[2][-1]		
sea[4][-1]		
sea[0][19]		
sea[0][-1]		

C does not do any bounds checking.

Row-major array layout is guaranteed.

# C structs

Like Java class/object, without methods.

Models structured, but not necessarily list-like, data.

Combines other, simpler types.

```
struct point {  
    int xcoordinate;  
    int ycoordinate;  
};
```

```
struct student {  
    int classyear;  
    int id;  
    char* name;  
};
```

# C structs

Like Java class/object without methods.

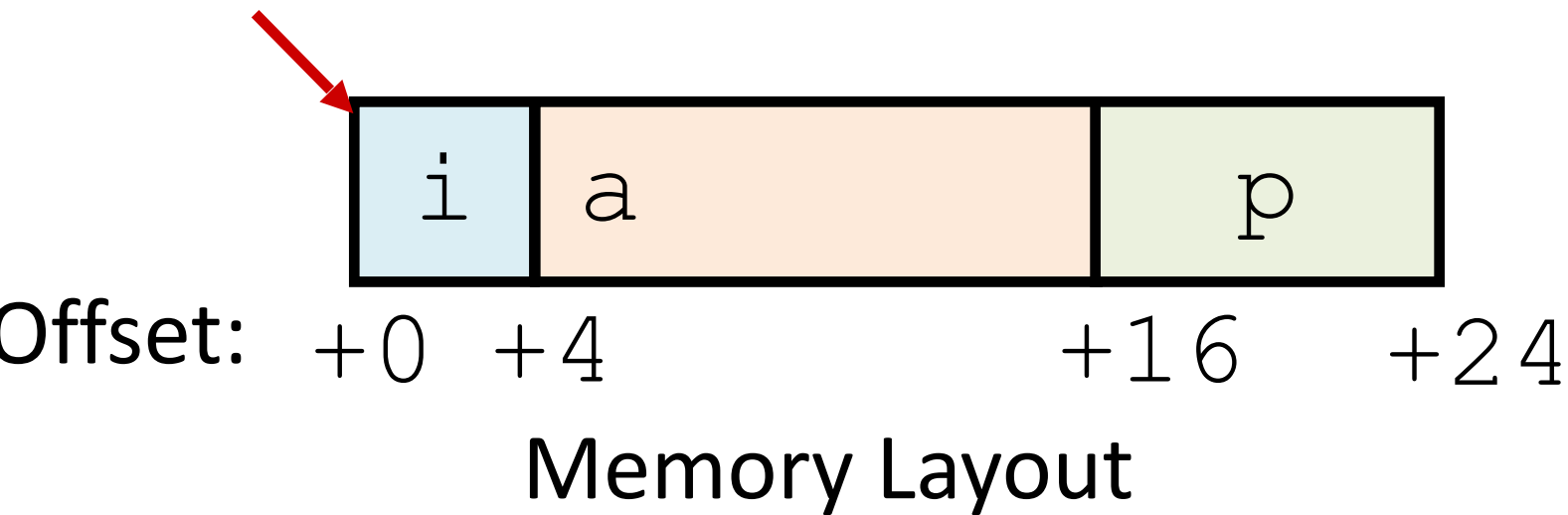
Compiler determines:

- Total size
- Offset of each field

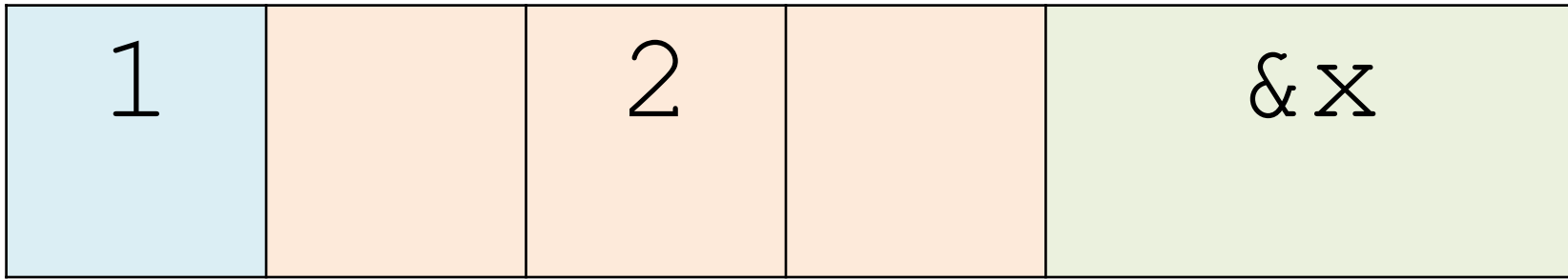
```
struct rec {  
    int i;  
    int a[3];  
    int* p;  
};
```

```
struct rec x;  
struct rec y;  
x.i = 1;  
x.a[1] = 2;  
x.p = &(x.i);
```

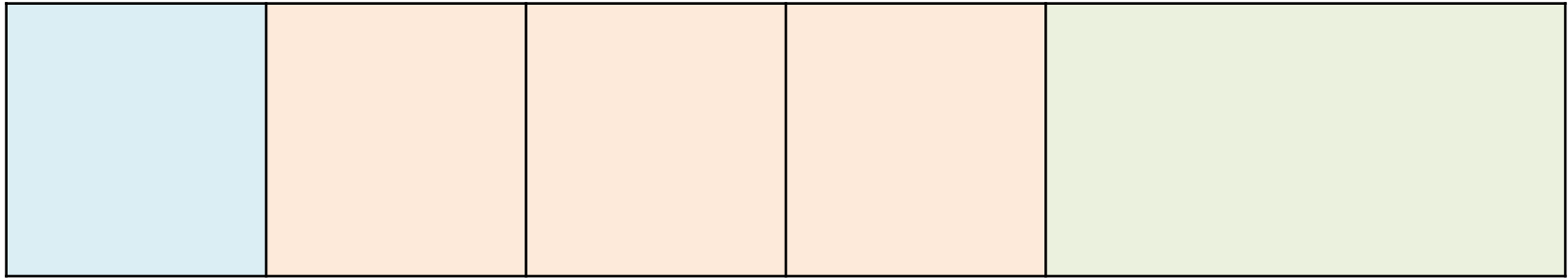
Base address



x



y



# C structs

Like Java class/object without methods.

Compiler determines:

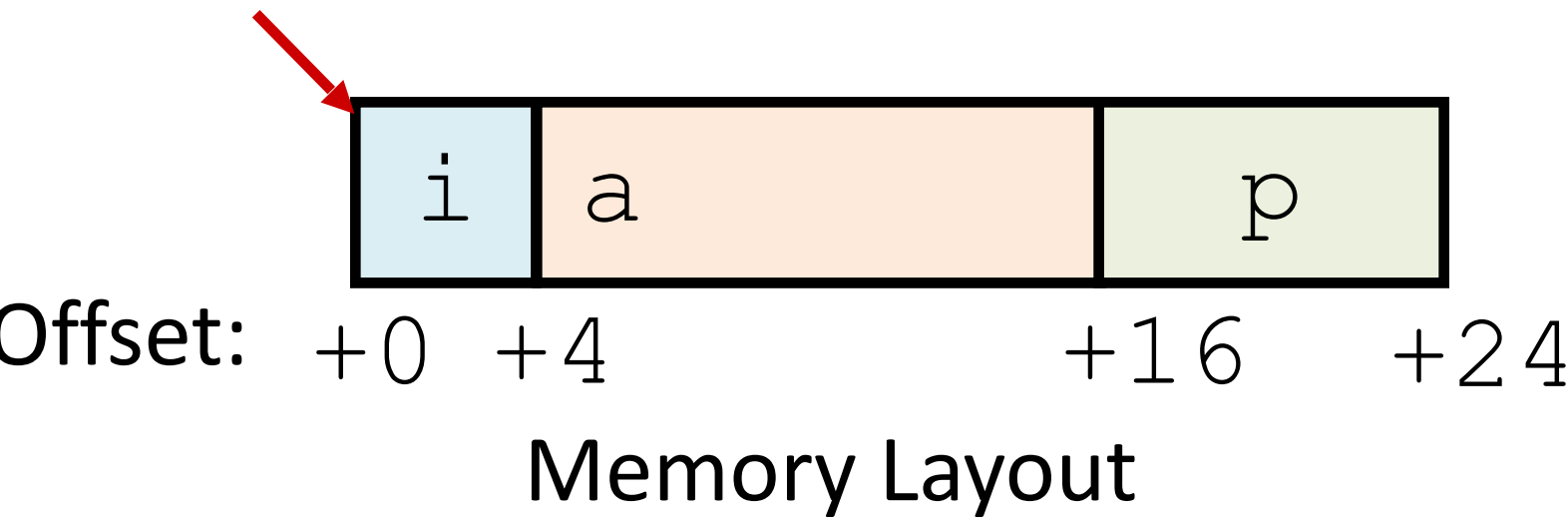
- Total size
- Offset of each field

```
struct rec {  
    int i;  
    int a[3];  
    int* p;  
};
```

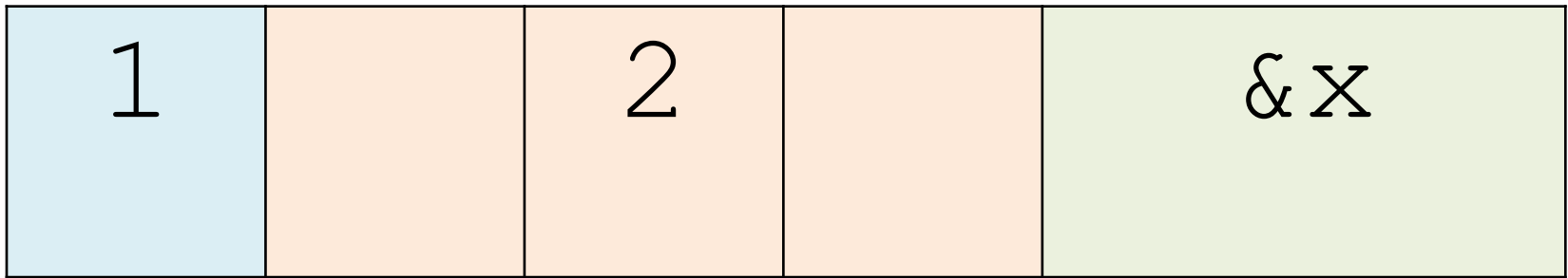
```
struct rec x;  
struct rec y;  
x.i = 1;  
x.a[1] = 2;  
x.p = &(x.i);
```

```
// copy full struct  
y = x;
```

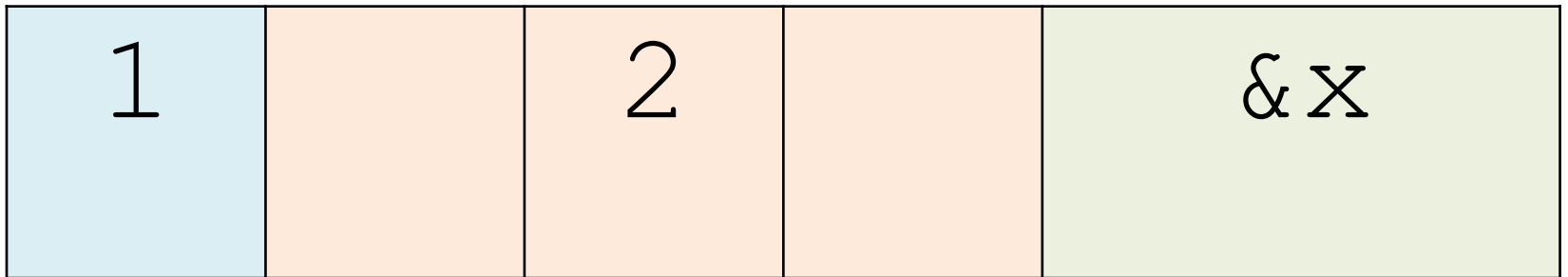
Base address



x



y



# C structs

Like Java class/object without methods.

Compiler determines:

- Total size
- Offset of each field

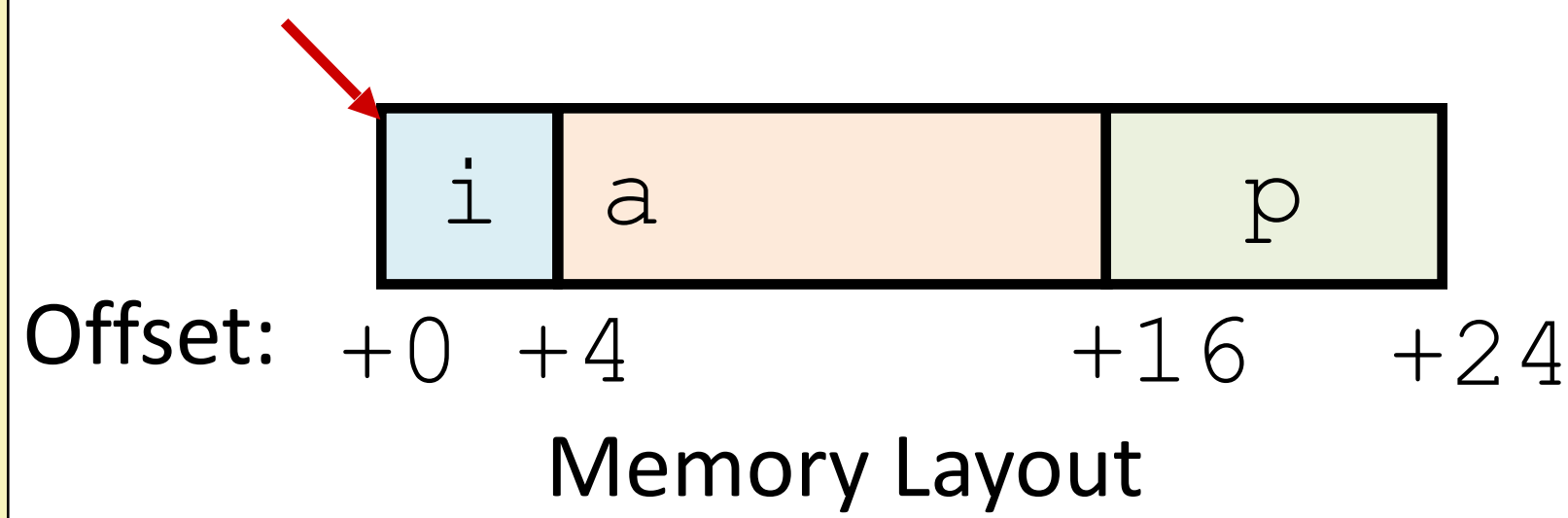
```
struct rec {  
    int i;  
    int a[3];  
    int* p;  
};
```

```
struct rec x;  
struct rec y;  
x.i = 1;  
x.a[1] = 2;  
x.p = &(x.i);
```

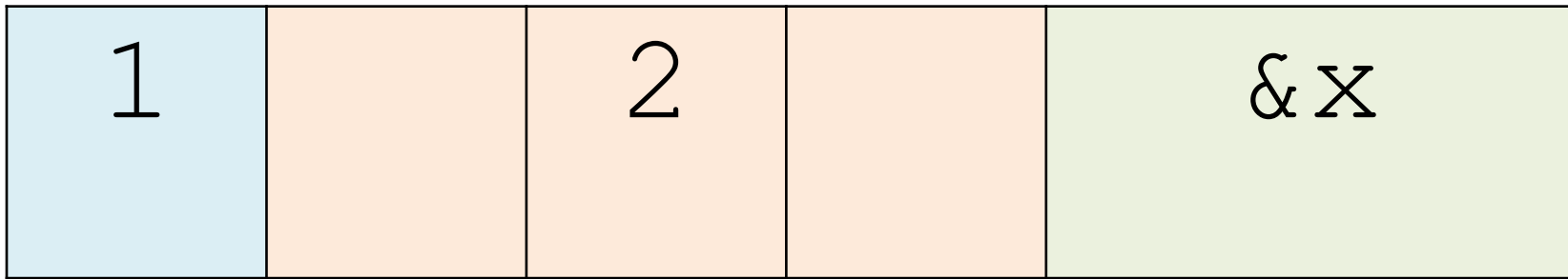
```
// copy full struct  
y = x;
```

```
struct rec* z;  
z = &y;
```

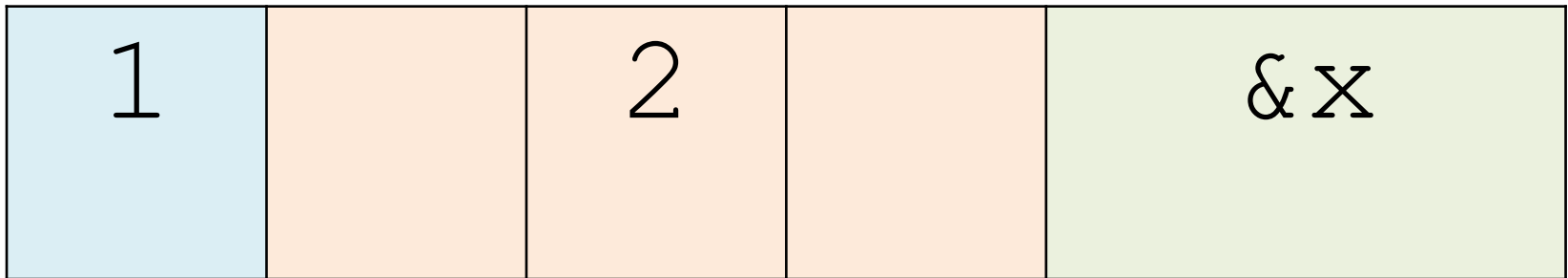
Base address



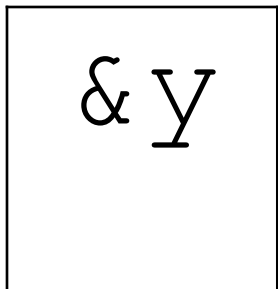
x



y



z



# C structs

Like Java class/object without methods.

Compiler determines:

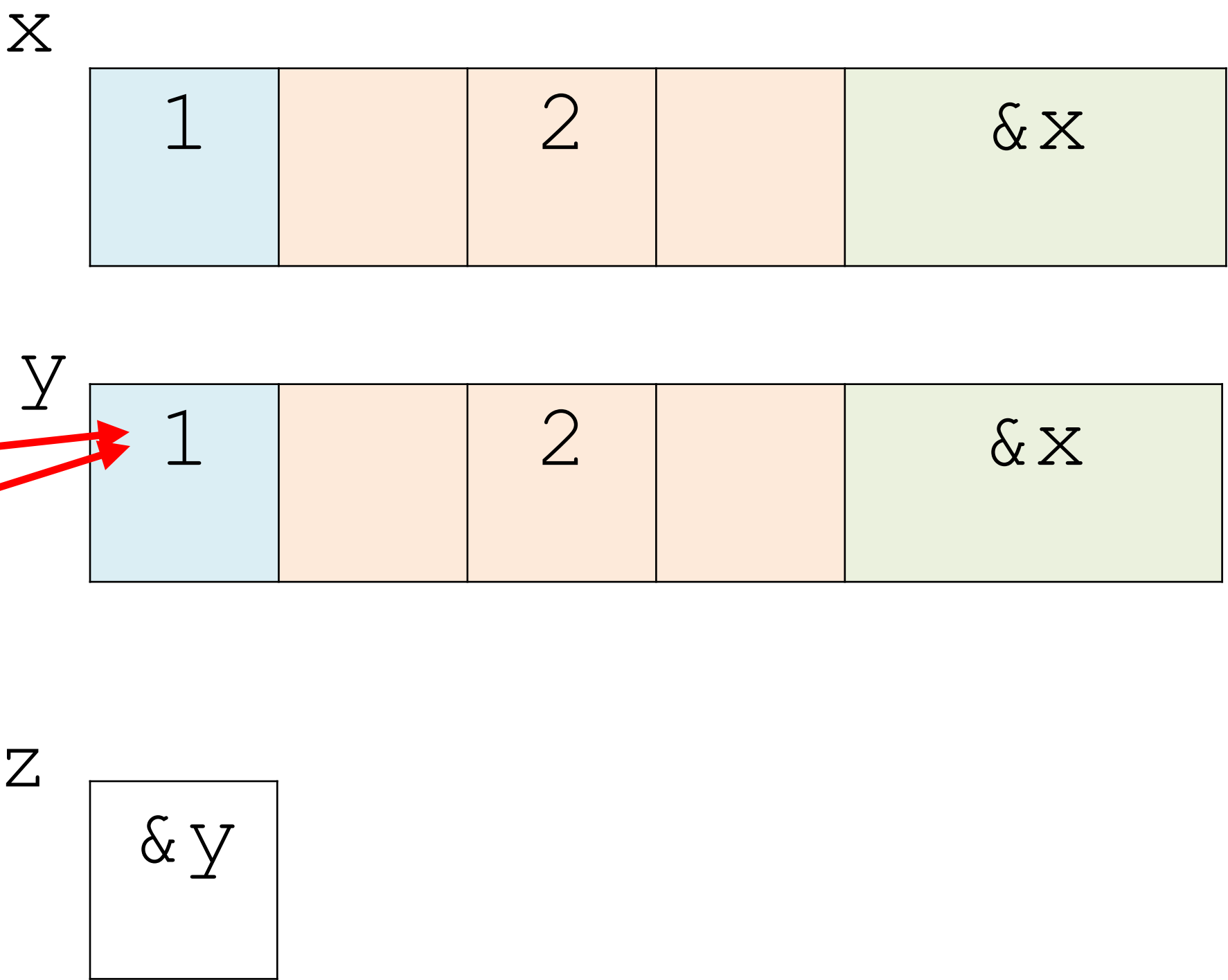
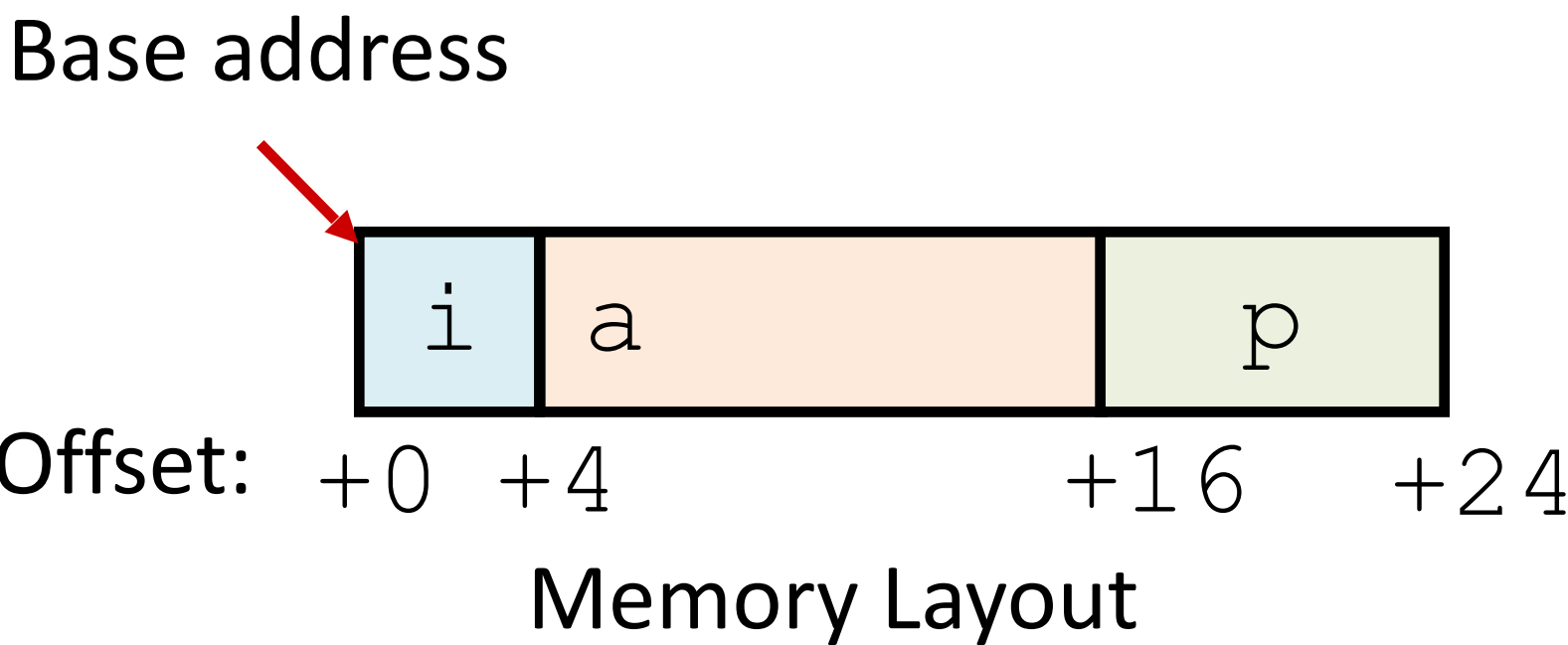
- Total size
- Offset of each field

```
struct rec {
    int i;
    int a[3];
    int* p;
};

struct rec x;
struct rec y;
x.i = 1;
x.a[1] = 2;
x.p = &(x.i);

// copy full struct
y = x;

struct rec* z;
z = &y;
(*z).i++;
// same as:
// z->i++
```



# C structs

Like Java class/object without methods.

Compiler determines:

- Total size
- Offset of each field

```
struct rec {
    int i;
    int a[3];
    int* p;
};
```

```
struct rec x;
struct rec y;
x.i = 1;
x.a[1] = 2;
x.p = &(x.i);
```

```
// copy full struct
y = x;
```

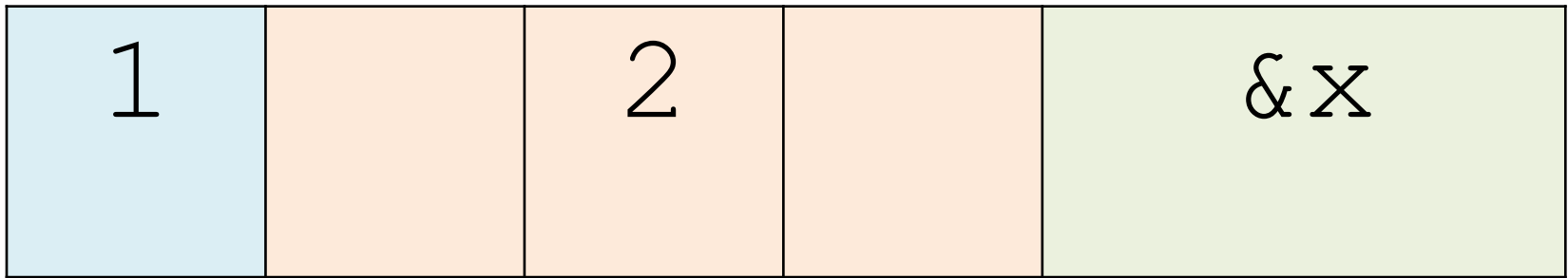
```
struct rec* z;
z = &y;
(*z).i++;
// same as:
// z->i++
```

Base address

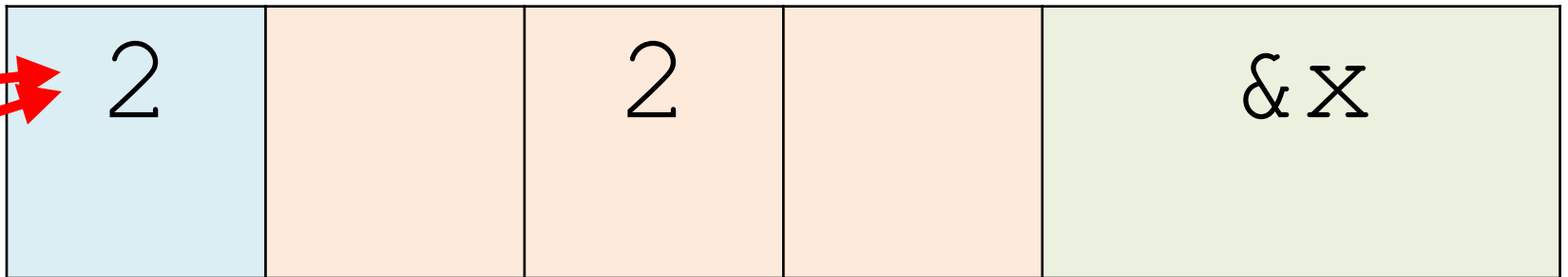
Offset: +0 +4 +16 +24

Memory Layout

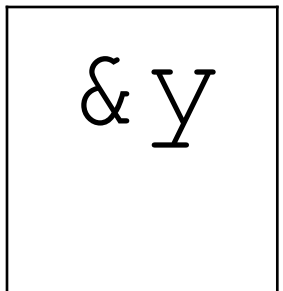
x



y



z







# C: Accessing struct fields

ex

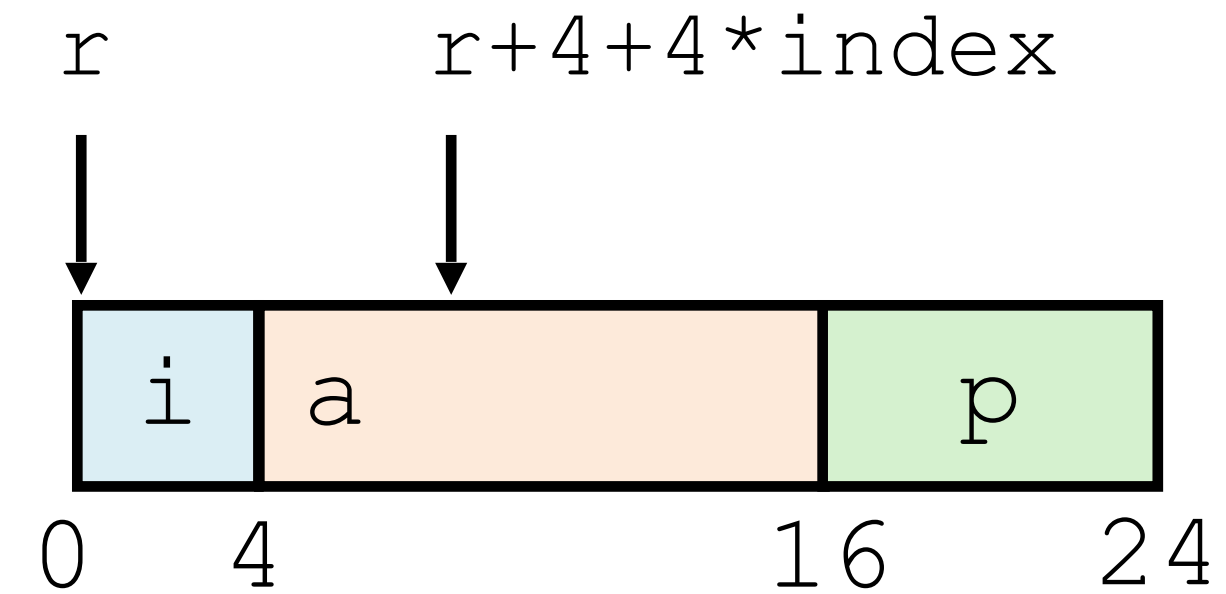
```
struct student {  
    int classyear;  
    int id;  
    char* name;  
};
```

Example: traversing a list of struct pointers

```
// Given a null-terminated list of pointers to students,  
// return the name of the student with a given ID, or null  
// if there is no student with that ID.  
char* getStudentNameWithId(struct student *s[], int id) {  
    struct student **curr = s;  
    while (*curr) {  
        if ((*curr)->id == id)  
            return (*curr)->name;  
        curr++;  
    }  
    return NULL;  
}
```

# C: Accessing struct field

```
struct rec {  
    int i;  
    int a[3];  
    int* p;  
};
```



```
int get_i_plus_elem(struct rec* r, int index) {  
    return r->i + r->a[index];  
}
```

```
movl 0(%rdi),%eax          # Mem[r+0]  
addl 4(%rdi,%rsi,4),%eax.   # Mem[r+4*index+4]  
retq
```

# C: Struct field alignment

Alignment is especially important for structs

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

Defines new struct type  
and declares variable `p`  
of type `struct S1*`

Unaligned Data (not what C does)

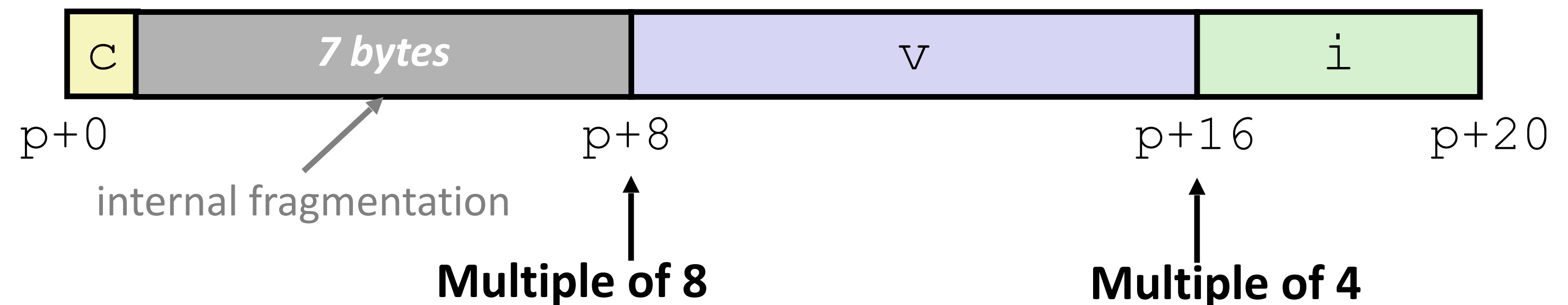


Aligned Data (what C does)

Primitive data type requires ***K*** bytes

Address must be multiple of ***K***

**C:** align every struct field accordingly.



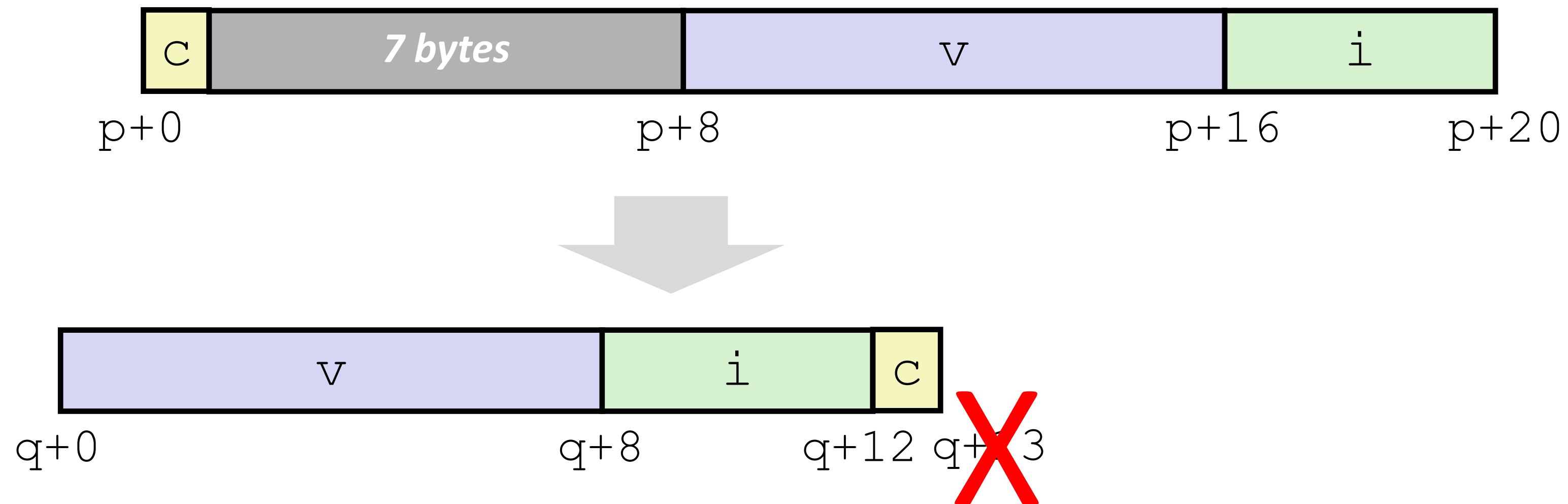
# C: Struct packing

Put large data types first:

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

→  
programmer

```
struct S2 {  
    double v;  
    int i;  
    char c;  
} * q;
```



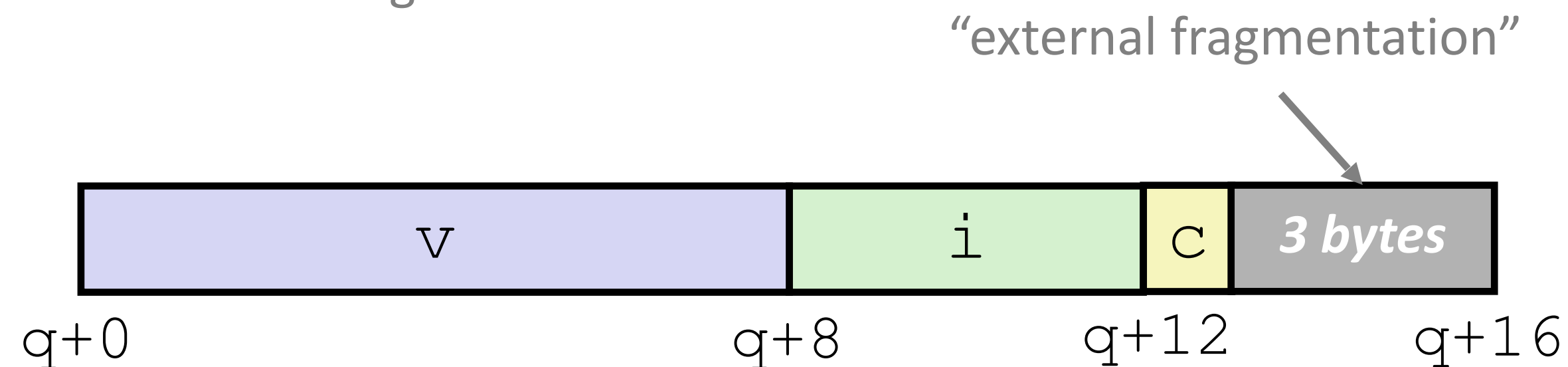
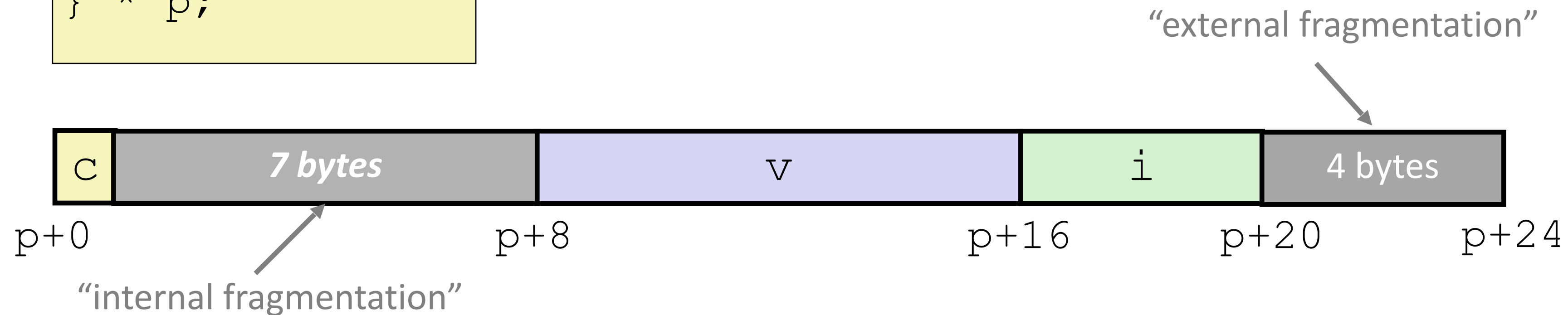
but actually...

# C: Struct alignment (full)

Base *and total size* must align largest internal primitive type.

Fields must align their type's largest alignment requirement.

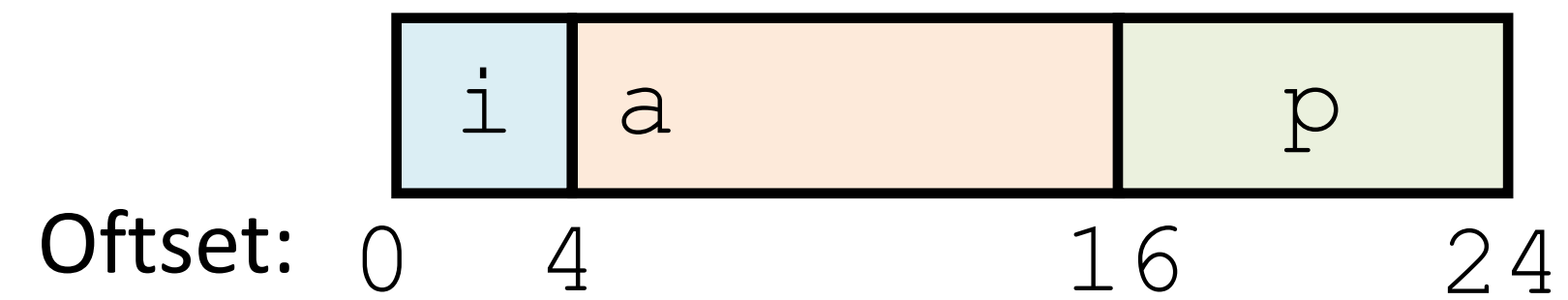
```
struct S1 {  
    char c;  
    double v;  
    int i;  
} * p;
```



```
struct S2 {  
    double v;  
    int i;  
    char c;  
} * q;
```

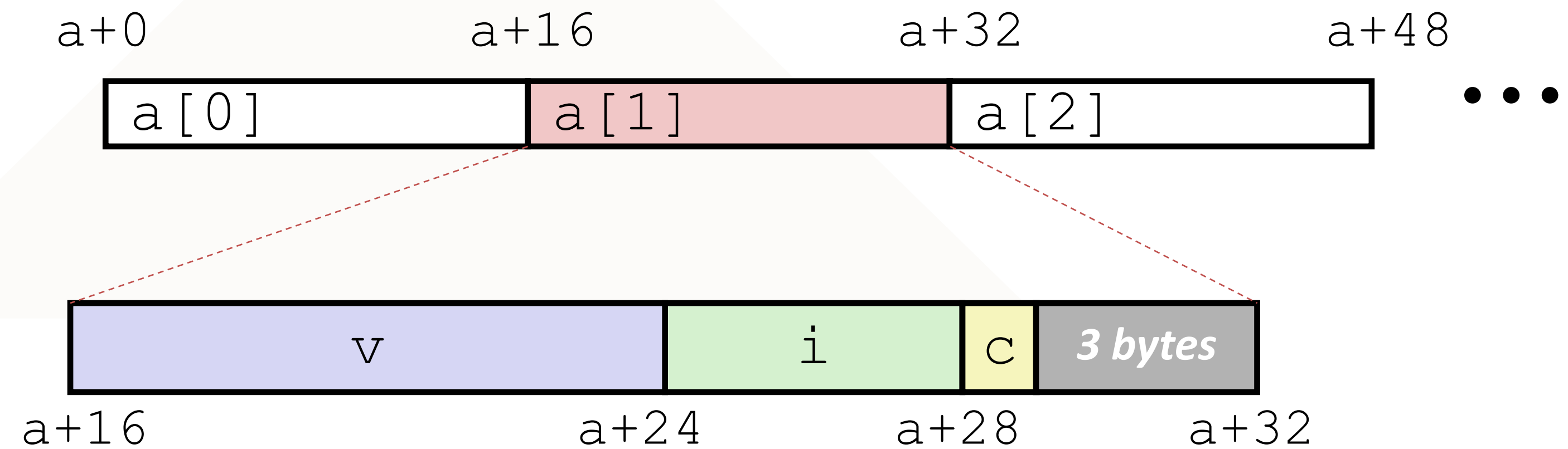
# Array in struct

```
struct rec {  
    int i;  
    int a[3];  
    int* p;  
};
```



# Struct in array

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



# C: typedef

```
// give type T another name: U  
typedef T U;
```

```
// struct types can be verbose  
struct Node { ... };
```

```
...
```

```
struct Node* n = ...;
```

```
// typedef can help  
typedef struct Node {
```

```
    ...
```

```
} Node;
```

```
...
```

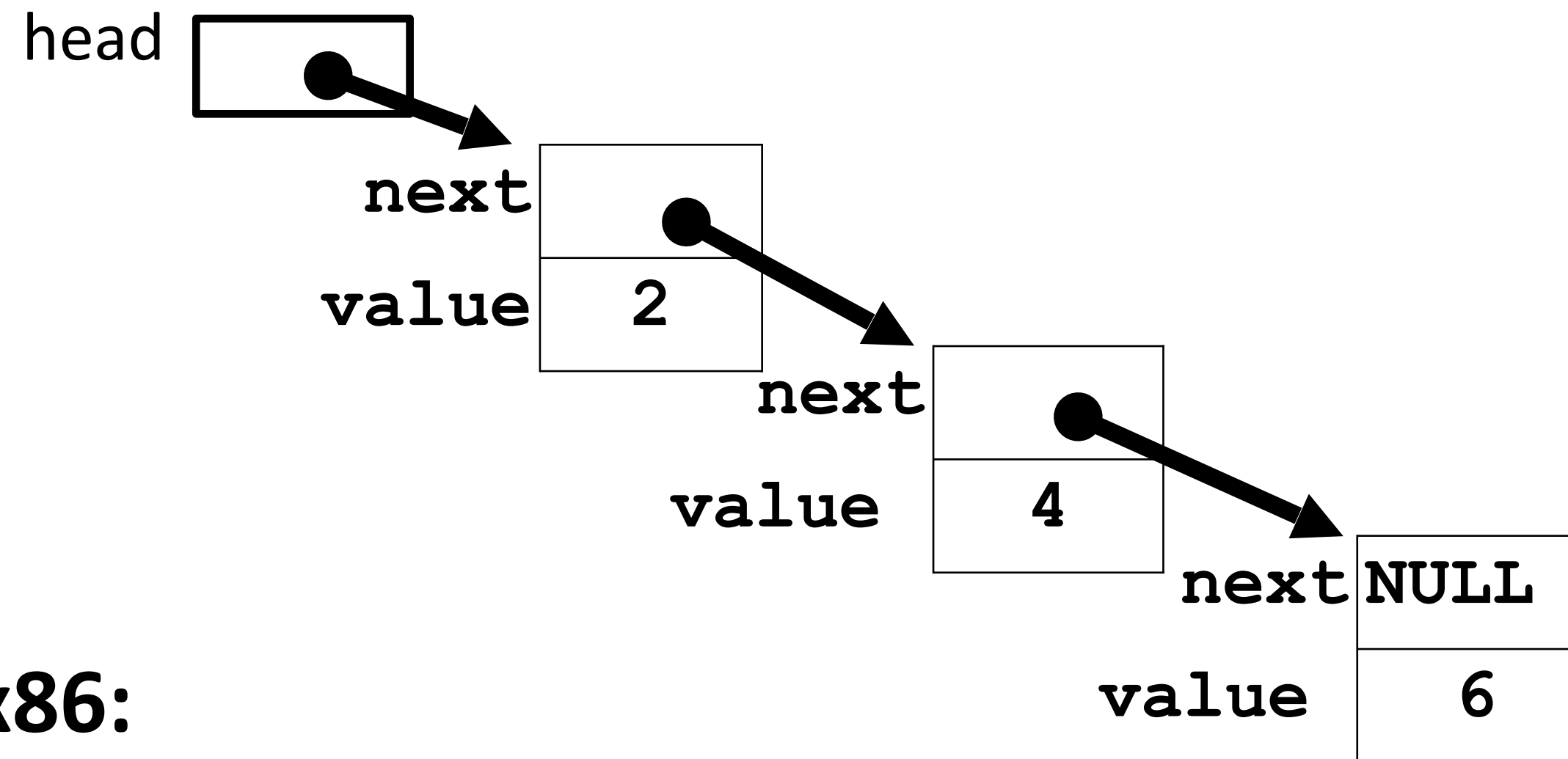
```
Node* n = ...;
```



# Linked Lists

ex

```
typedef
struct Node {
    struct Node* next;
    int value;
} Node;
```



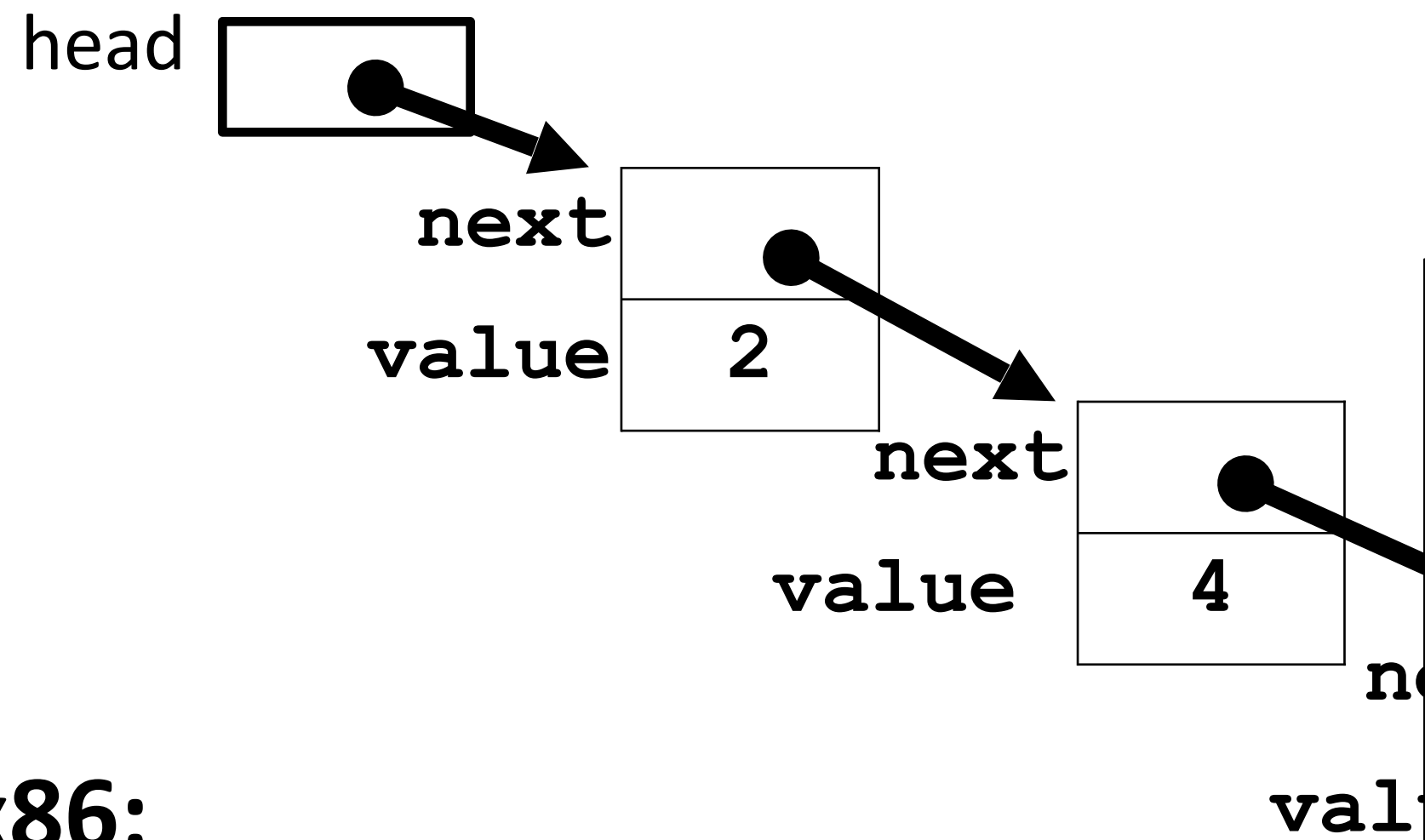
## Implement append in x86:

```
void append(Node* head, int x) {
    // assume head != NULL
    Node* cursor = head;
    // find tail
    while (cursor->next != NULL) {
        cursor = cursor->next;
    }
    Node* n = (Node*)malloc(sizeof(Node));
    // error checking omitted
    // for x86 simplicity
    cursor->next = n;
    n->next = NULL;
    n->value = x;
}
```

# Linked Lists

ex

```
typedef
struct Node {
    struct Node* next;
    int value;
} Node;
```



## Implement append in x86:

```
void append(Node* head, int x) {
    // assume head != NULL
    Node* cursor = head;
    // find tail
    while (cursor->next != NULL) {
        cursor = cursor->next;
    }
    Node* n = (Node*)malloc(sizeof(Node));
    // error checking omitted
    // for x86 simplicity
    cursor->next = n;
    n->next = NULL;
    n->value = x;
}
```

Extra fun: try a recursive version too!

### append:

```
pushq %rbp
movl %esi, %ebp
pushq %rbx
movq %rdi, %rbx
subq $8, %rsp
jmp .L3
.L6:
    movq %rax, %rbx
.L3:
    movq (%rbx), %rax
    testq %rax, %rax
    jne .L6
    movl $16, %edi
    call malloc
    movq %rax, (%rbx)
    movq $0, (%rax)
    movl %ebp, 8(%rax)
    addq $8, %rsp
    popq %rbx
    popq %rbp
    ret
```

# Struct practice problem (similar to CSAPP 3.45)

ex

```
struct s {  
    char *a;  
    short b;  
    int *c;  
    char d;  
    int e;  
    char f;  
};
```

Recall: a short is  
2 bytes in C

1. Draw a picture of how this struct is laid out in memory, labeling the byte offset of each field (starting with a at offset +0);
2. Modify your picture to show how much space a single element of this struct would take if used as an element of an array (e.g., the total size).
3. Rearrange the fields of the struct to minimize wasted space. Draw the new offsets and the total size.

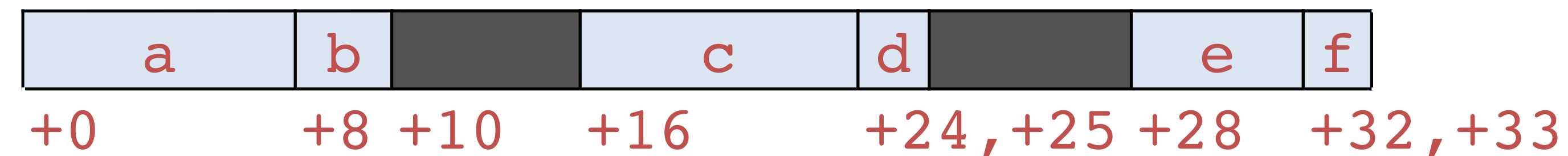
# Struct practice problem (similar to CSAPP 3.45)

ex

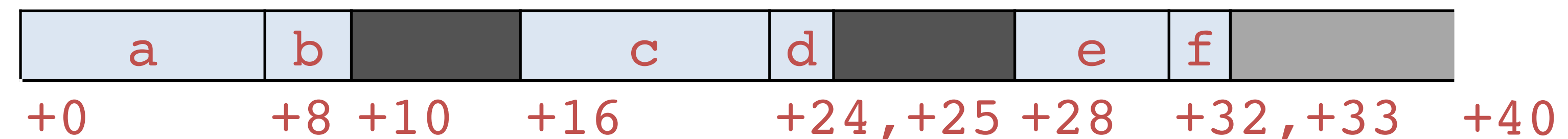
```
struct s {  
    char *a;  
    short b;  
    int *c;  
    char d;  
    int e;  
    char f;  
};
```

Recall: a short is  
2 bytes in C

1. Draw a picture of how this struct is laid out in memory, labeling the byte offset of each field (starting with a at offset +0);



2. Modify your picture to show how much space a single element of this struct would take if used as an element of an array (e.g., the total size).



3. Rearrange the fields of the struct to minimize wasted space. Draw the new offsets and the total size.

