

# Structs & Alignment

CSE 351 Autumn 2020

## Instructor:

Justin Hsia

## Teaching Assistants:

Aman Mohammed

Cosmo Wang

Joy Dang

Kyrie Dowling

Yan Zhe Ong

Ami Oka

Hang Do

Julia Wang

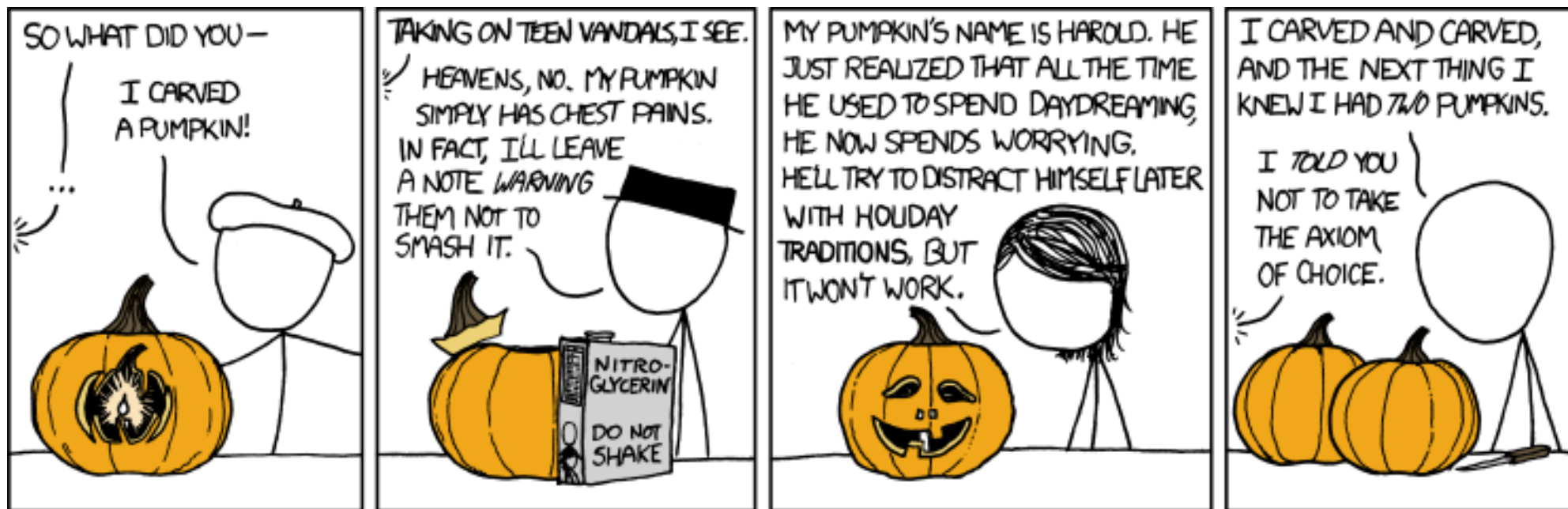
Mariam Mayanja

Callum Walker

Jim Limprasert

Kaelin Laundry

Shawn Stanley



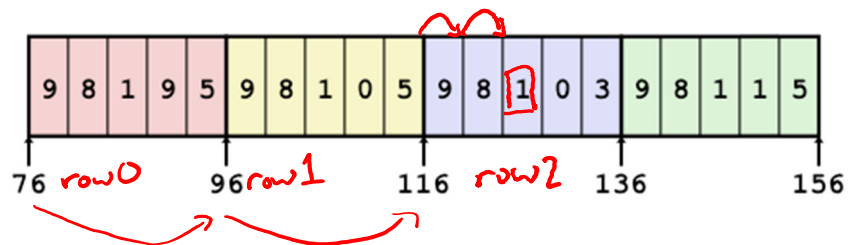
# Administrivia

- ❖ Lab 2 due tonight
- ❖ Lab 3 released next Wednesday (11/4)
- ❖ hw13 due next Wednesday (11/4)
  
- ❖ **Midterm, Group Stage (10/31 – 11/2)**
  - Instructions will be posted on Ed Discussion
  - **Communicate with your group**: work together, help each other out, and prep for individual stage
  - We will be available on Ed Discussion (private posts, please) and office hours to answer clarifying questions

# Array Element Accesses

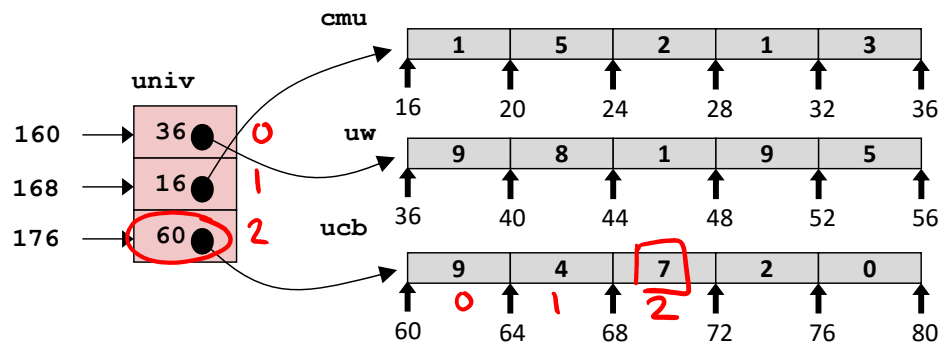
## Multidimensional array

```
int get_sea_digit
(int index, int digit)
{
    return sea[index][digit];
}
```



## Multilevel array

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



Access *looks* the same, but it isn't:

**Mem**[sea+20\*index+4\*digit]

more efficient:

- less overall memory
- faster to access

**Mem**[Mem[univ+8\*index]+4\*digit]

more flexible:

- easier to "fit" smaller arrays in memory
- can swap out rows (and resize)
- can have rows of different lengths

# Reading Review

- ❖ Terminology:
  - Structs: tags and fields, . and -> operators
  - Typedef
  - Alignment, internal fragmentation, external fragmentation
  
- ❖ Questions from the Reading?

# Review Questions

```

struct ll_node {
    8B long data;
    8B struct ll_node* next;
} n1, n2;
    
```

*Handwritten annotations:*  
 - A red arrow labeled "tag" points to the opening curly brace of the struct definition.  
 - A red bracket on the right side of the struct definition is labeled "fields".  
 - A red bracket under "n1, n2;" is labeled "two instances".  
 - A red note "K<sub>max</sub>=8" is written to the left of the struct definition.

❖ How much space does (in bytes) does an instance of struct ll\_node take? 16 B

❖ Which of the following statements are syntactically valid?

*Handwritten note:* • for struct instances, (inst) → for struct pointers (ptr)

- ✓ *inst* ✓ *ptr* ptr  
 ■ n1.next = &n2;
- ✗ *inst* ✗  
 ■ n2->data = 351;
- ✓ *inst* ✓ *ptr* ✓ *long*  
 ■ n1.next->data = 333;
- ✗ *ptr* ✓ *ptr* ✓ *ptr* ✗  
 ■ (&n2) ->next->next.data = 451;

# Data Structures in Assembly

## ❖ Arrays

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

## ❖ **Structs**

- **Alignment**

## ❖ ~~Unions~~

# Structs in C

- ❖ A structured group of variables, possibly including other structs
  - Way of defining compound data types

```
struct song {
    char* title;
    int lengthInSeconds;
    int yearReleased;
};

struct song song1;
song1.title = "Señorita";
song1.lengthInSeconds = 191;
song1.yearReleased = 2019;

struct song song2;
song2.title = "Call Me Maybe";
song2.lengthInSeconds = 193;
song2.yearReleased = 2011;
```

```
struct song {
    char* title;
    int lengthInSeconds;
    int yearReleased;
};
```

```
song1
title:      "Señorita"
lengthInSeconds: 191
yearReleased: 2019
```

```
song2
title: "Call Me Maybe"
lengthInSeconds: 193
yearReleased: 2011
```

# Struct Definitions

- ❖ Structure definition:
  - Does NOT declare a variable
  - Variable type is “struct name”

```
struct name {  
    /* fields */  
};
```

*your choice*

Easy to forget semicolon!

- ❖ Variable declarations like any other data type:

```
struct name name1, *pn, name_ar[3];
```

instance

pointer

array

- ❖ Can also combine struct and instance definitions:
  - This syntax can be difficult to parse, though

```
struct name {  
    /* fields */  
} st, *p = &st;
```

*this is the data type (like int)*



# Typedef in C

- ❖ A way to create an *alias* for another data type:

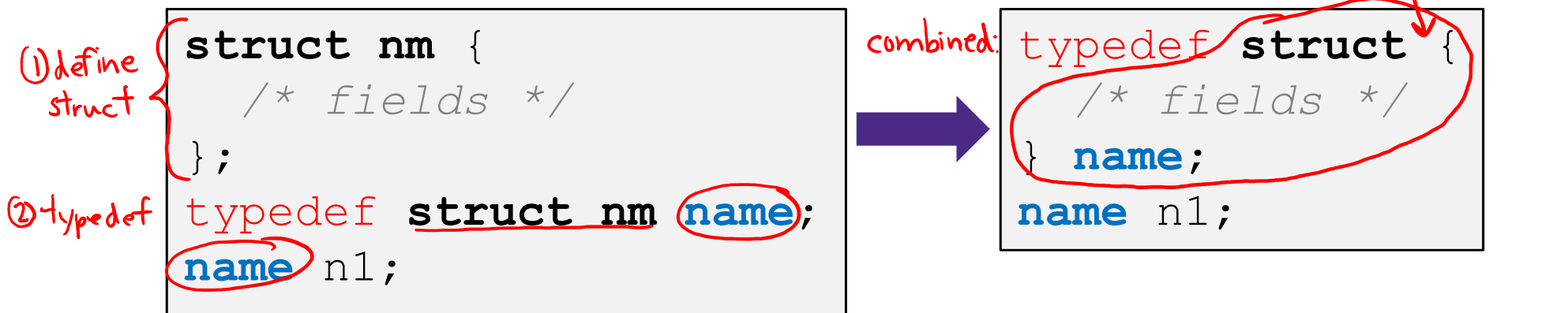
```
typedef <data type> <alias>;
```

- After typedef, the alias can be used interchangeably with the original data type

- *e.g.*, `typedef unsigned long int uli;`  
data type alias

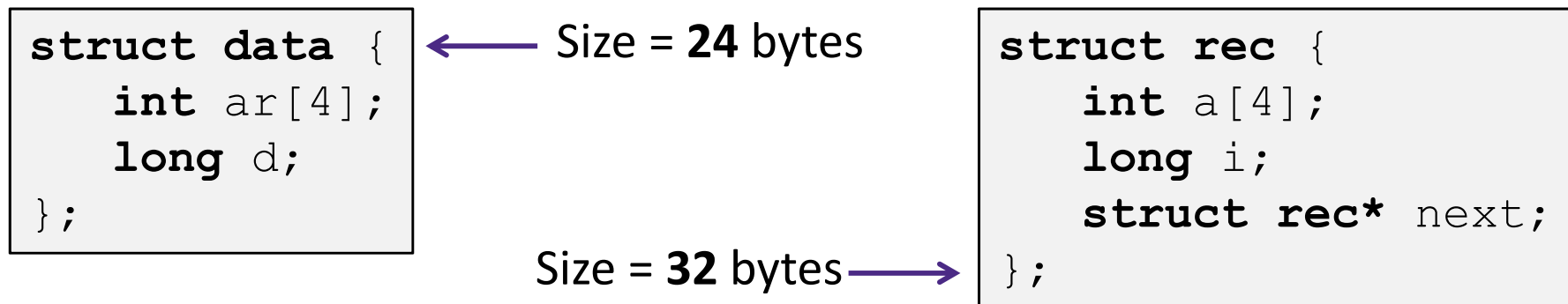
- ❖ Joint struct definition and typedef

- Don't need to give struct a name in this case



# Scope of Struct Definition

- ❖ Why is the placement of struct definition important?
  - Declaring a variable creates space for it somewhere
  - Without definition, program doesn't know how much space



- ❖ Almost always define structs in global scope near the top of your C file
  - Struct definitions follow normal rules of scope

# Accessing Structure Members

- ❖ Given a struct instance, access member using the `.` operator:

```
struct rec r1;
r1.i = val;
```

- ❖ Given a *pointer* to a struct:

```
struct rec* r;
```

```
r = &r1; // or malloc space for r to point to
```

We have two options:

- Use `*` and `.` operators: `(*r).i = val;`
- Use `->` operator for short: `r->i = val;`

① dereference (get instance)  
 ② access field  
 equivalent

- ❖ **In assembly:** register holds address of the first byte

- Access members with offsets

$D(Rb, Ri, S)$

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

# Java side-note

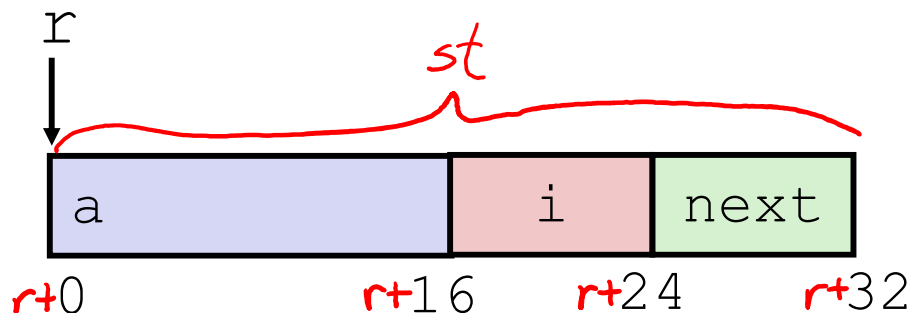
```
class Record { ... }  
Record x = new Record();
```

- ❖ An instance of a class is like a *pointer to* a struct containing the fields
  - (Ignoring methods and subclassing for now)
  - So Java's x.f is like C's x → f or (\*x).f
- ❖ In Java, almost everything is a pointer (“*reference*”) to an object
  - Cannot declare variables or fields that are structs or arrays
  - Always a *pointer* to a struct or array
  - So every Java variable or field is ≤ 8 bytes (but can point to lots of data)

# Structure Representation

```
struct rec {  
    int a[4];  
    long i;  
    struct rec* next;  
} st, *r = &st;
```

*instance* (pointing to `st`)  
*pointer* (pointing to `*r`)

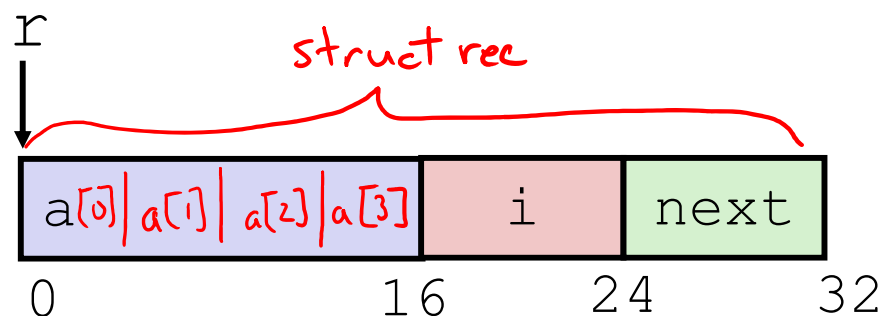


## ❖ Characteristics

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Fields may be of different types

# Structure Representation

```
struct rec {  
  ① int a[4];  
  ② long i;  
  ③ struct rec* next;  
} st, *r = &st;
```

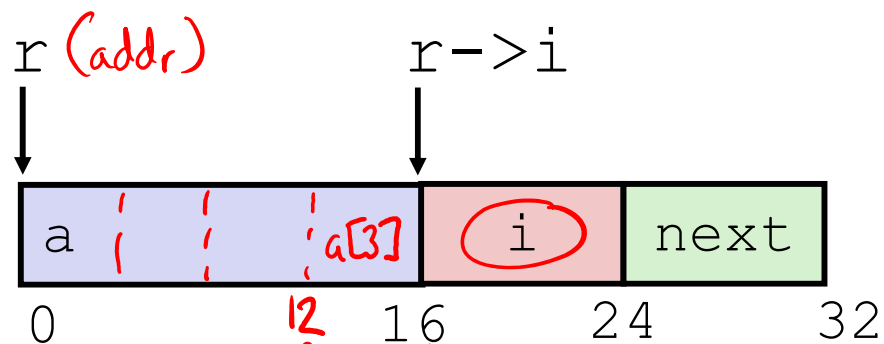


- ❖ Structure represented as block of memory
  - Big enough to hold all of the fields
- ❖ Fields ordered according to declaration order
  - Even if another ordering would be more compact
- ❖ Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures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

# Accessing a Structure Member

```

struct rec {
    int a[4];
    long i;
    struct rec* next;
} st, *r = &st;
    
```



- ❖ Compiler knows the *offset* of each member
  - No pointer arithmetic; compute as `*(r+offset)`

```

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

```

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

```

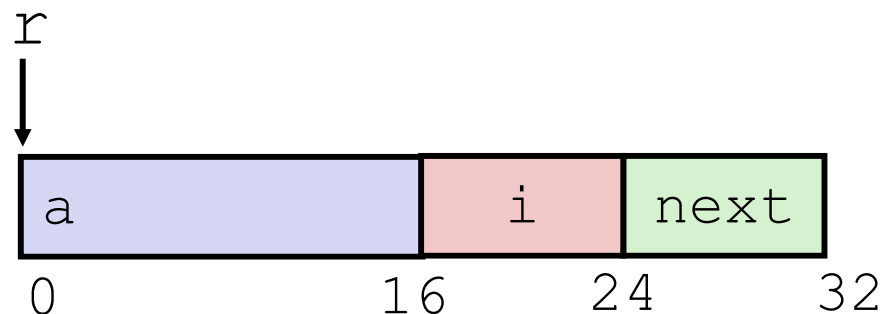
long get_a3(struct rec* r) {
    return r->a[3];
}
    
```

```

# r in %rdi
movl 12(%rdi), %rax
ret int
    
```

# Pointer to Structure Member

```
struct rec {  
    int a[4];  
    long i;  
    struct rec* next;  
} st, *r = &st;
```



```
long* addr_of_i(struct rec* r)  
{  
    return &(r->i);  
}
```

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

```
struct rec** addr_of_next(struct rec* r)  
{  
    return &(r->next);  
}
```

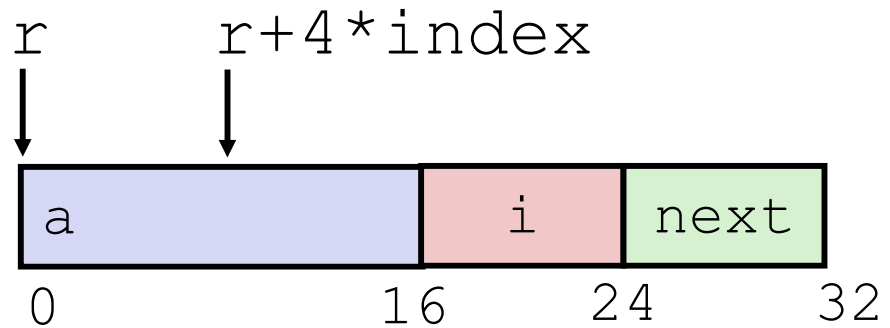
```
# r in %rdi  
leaq 24(%rdi), %rax  
ret
```



# Generating Pointer to Array Element

```

struct rec {
    int a[4];
    long i;
    struct rec* next;
} st, *r = &st;
    
```



## ❖ Generating Pointer to Array Element

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

```

int* find_addr_of_array_elem
(struct rec* r, long index)
{
    return &r->a[index];
}
    
```

`&(r->a[index])`

```

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

# Review: Memory Alignment in x86-64

- ❖ *Aligned* means that any primitive object of  $K$  bytes must have an address that is a multiple of  $K$
- ❖ Aligned addresses for data types:

$K$	Type	Addresses
1	char	No restrictions
2	short	Lowest bit must be zero: $\dots 0_2$
4	int, float	Lowest 2 bits zero: $\dots 00_2$
8	long, double, *	Lowest 3 bits zero: $\dots 000_2$
16	long double	Lowest 4 bits zero: $\dots 0000_2$

lowest  $\log_2(K)$   
bits should be 0

"multiple of" means no remainder when you divide by.  
 since  $K$  is a power of 2, dividing by  $K$  is equivalent to  $\gg \log_2(K)$ .  
 No remainder means no weight is "lost" during the shift  $\rightarrow$  all zeros in lowest  $\log_2(K)$  bits.

# Alignment Principles

## ❖ Aligned Data

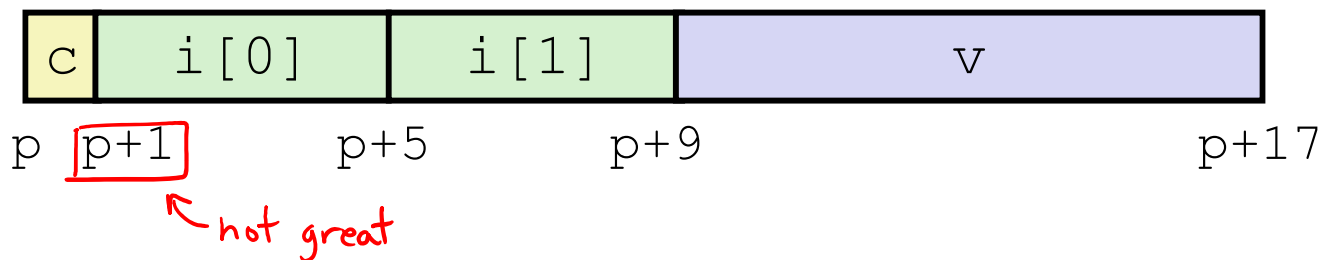
- Primitive data type requires  $K$  bytes
- Address must be multiple of  $K$
- Required on some machines; advised on x86-64

## ❖ Motivation for Aligning Data

- Memory accessed by (aligned) chunks of bytes (width is system dependent)
  - Inefficient to load or store value that spans quad word boundaries
  - Virtual memory trickier when value spans 2 pages (more on this later)
- Though x86-64 hardware will work regardless of alignment of data

# Structures & Alignment

## ❖ Unaligned Data



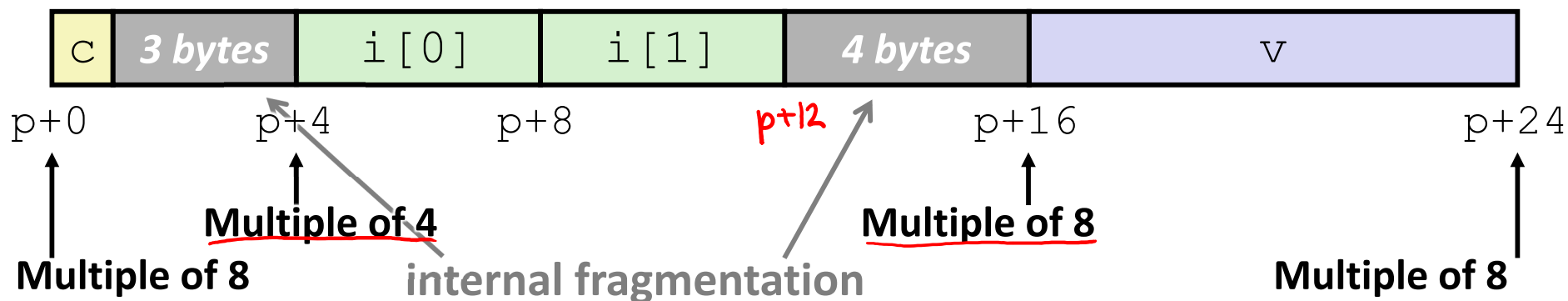
```

struct S1 {
  ① char c;
  ② int i[2];
  ③ double v;
} st, *p = &st;
    
```

K  
← 1  
← 4  
← 8

## ❖ Aligned Data

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



24 B total

# Satisfying Alignment with Structures (1)

❖ Within structure:

- Must satisfy each element's alignment requirement

❖ Overall structure placement

- Each structure has alignment requirement  $K_{max}$

- $K_{max}$  = Largest alignment of any element
- Counts array elements individually as elements

```

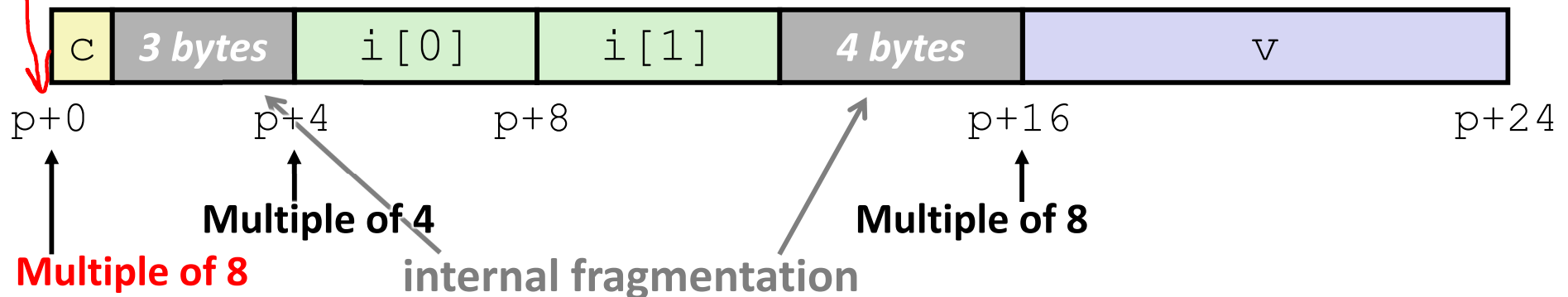
K
1
4
8
struct S1 {
    char c;
    int i[2];
    double v;
} st, *p = &st;
    
```

$K_{max} = 8$

*alignment requirement of starting addr*

❖ Example:

- $K_{max} = 8$ , due to double element



# Satisfying Alignment with Structures (2)

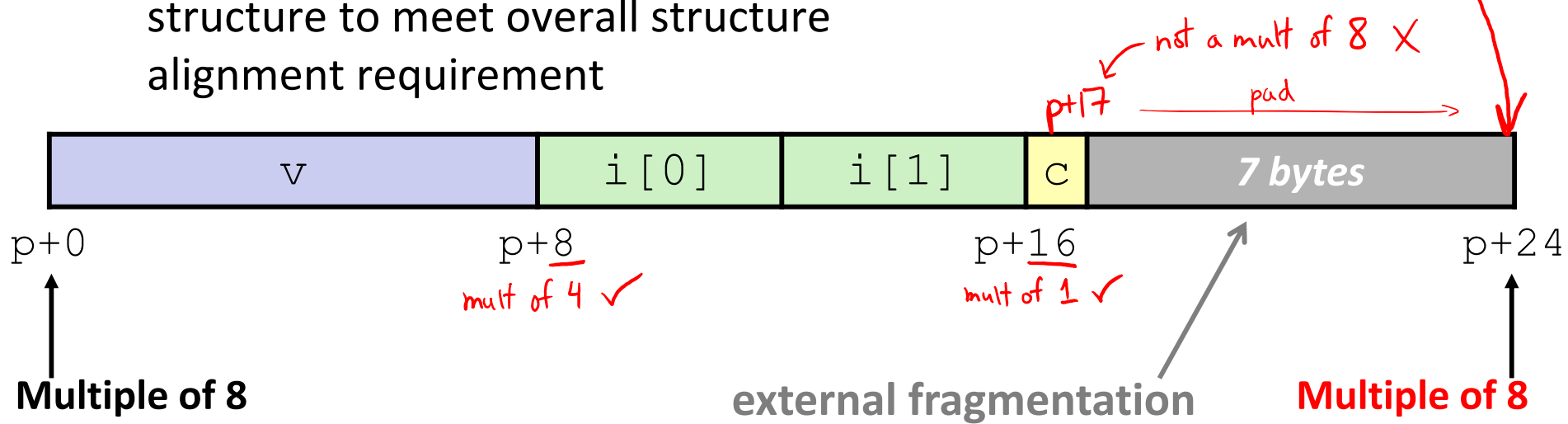
- ❖ Can find offset of individual fields using `offsetof()`
  - Need to `#include <stddef.h>`
  - Example: `offsetof(struct S2, c)` returns 16

```

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

- ❖ For largest alignment requirement  $K_{max}$ , overall structure size must be multiple of  $K_{max} = 8$

- Compiler will add padding at end of structure to meet overall structure alignment requirement

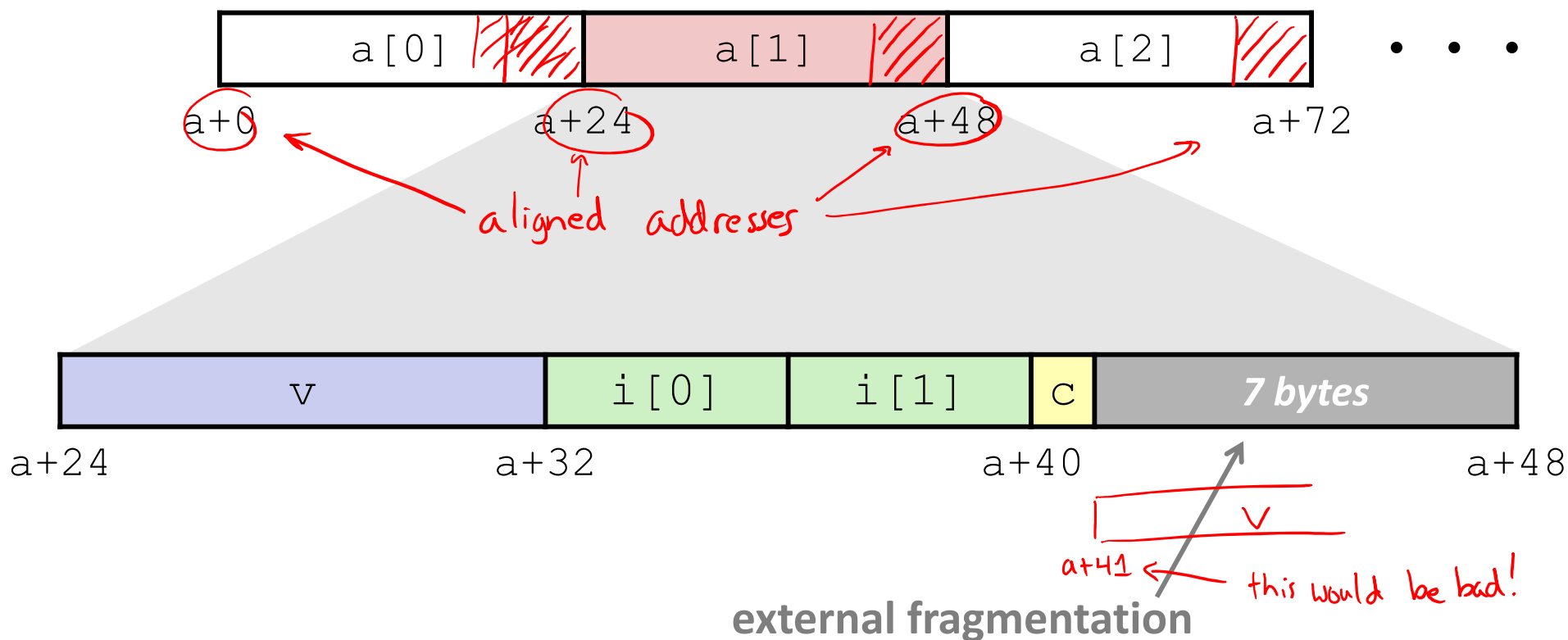


# Arrays of Structures

- ❖ Overall structure length multiple of  $K_{max}$
- ❖ Satisfy alignment requirement for every element in array

```

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



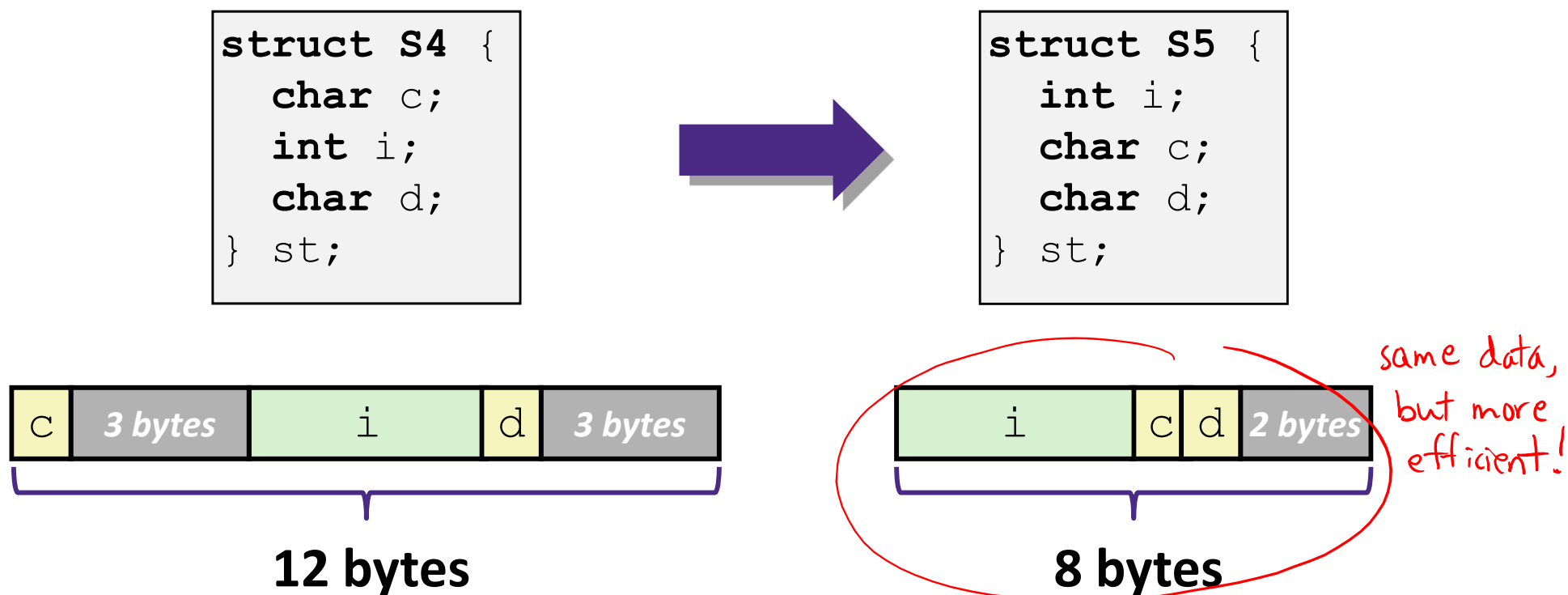
# Alignment of Structs

- ❖ Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each ***field*** must be aligned *within* the struct (*may insert padding*)
    - `offsetof` can be used to get actual field offset
  - Overall struct must be ***aligned*** according to largest field
  - Total struct ***size*** must be multiple of its alignment (*may insert padding*)
    - `sizeof` should be used to get true size of structs



# How the Programmer Can Save Space

- ❖ Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first



# Practice Questions

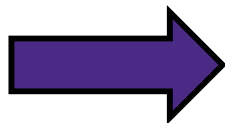
- ❖ Minimize the size of the struct by re-ordering the vars

$\frac{K}{4}$

```

struct old {
  int i;
  short s[3];
  char* c;
  float f;
};
    
```

$K_{max} = 8$



```

struct new {
  int i;
  float f;
  char* c;
  short s[3];
};
    
```

could also switch these (internal vs. external frag)

- ❖ What are the old and new sizes of the struct?

sizeof(struct old) = \_\_\_\_\_      sizeof(struct new) = \_\_\_\_\_

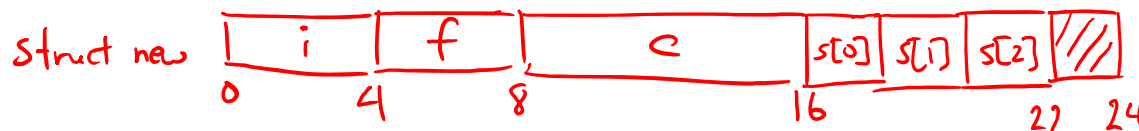
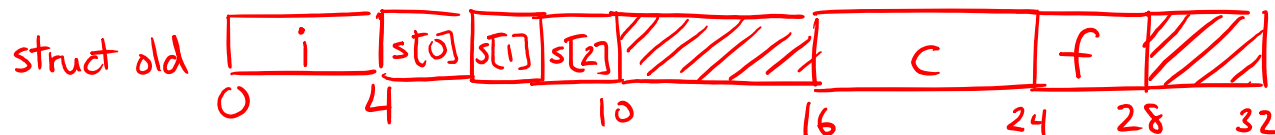
A. 16 bytes

B. 22 bytes

C. 28 bytes

**D. 32 bytes**

E. We're lost...



# Summary

- ❖ Arrays in C
  - Aligned to satisfy every element's alignment requirement
- ❖ Structures
  - Allocate bytes for fields in order declared by programmer
  - Pad in middle to satisfy individual element alignment requirements
  - Pad at end to satisfy overall struct alignment requirement