

# Structs & Alignment

CSE 351 Summer 2022

## Instructor:

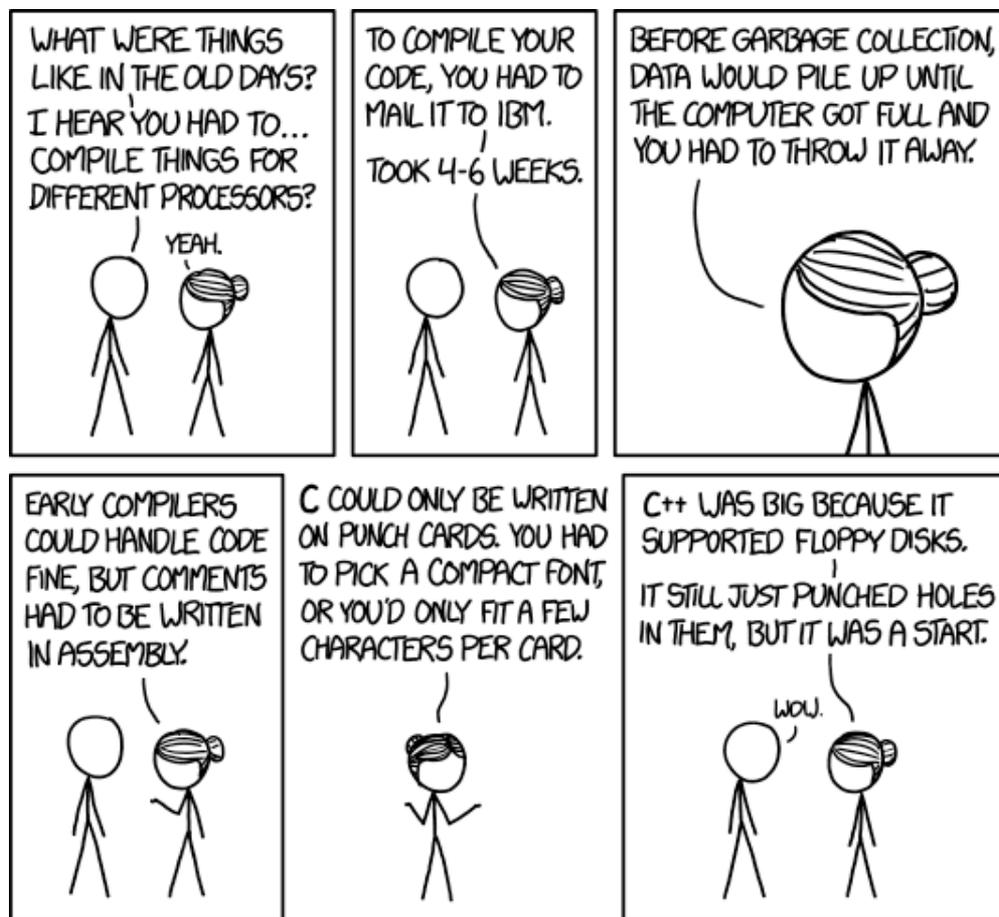
Kyrie Dowling

## Teaching Assistants:

Aakash Srazali

Allie Pflieger

Ellis Haker



<https://xkcd.com/1755/>

# Relevant Course Information

- ❖ hw13 due Friday (7/29)
  - Can finish after today's lecture
  
- ❖ Lab 3 due Friday (7/29)
  
- ❖ Unit Portfolio 2 released today, due 8/3
  - Based on the topics from lecture 7 through 14 (today)
  - Feedback from the first unit portfolio will be released before the second one is due

# Reading Review

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

# Review Questions

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

- ❖ How much space does (in bytes) does an instance of `struct ll_node` take?
- ❖ Which of the following statements are syntactically valid (possibly more than one)?
  - A. `n1.next = &n2;`
  - B. `n2->data = 351;`
  - C. `n1.next->data = 333;`
  - D. `(&n2)->next->next.data = 451;`

# Data Structures in C

## ❖ Arrays

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

## ❖ **Structs**

- **Alignment**

## ~~❖ Unions~~

# Structs in C (Review)

- ❖ 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 = "Rose-Colored Body";
song1.lengthInSeconds = 212;
song1.yearReleased = 2017;

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

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

```
song1
title: "Rose-Colored Boy"
lengthInSeconds:    212
yearReleased:      2017
```

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

# Struct Definitions (Review)

## ❖ Structure definition:

- Does NOT declare a variable
- Variable type is “**struct name**”

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

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 read, though

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

# Typedef in C (Review)

- ❖ 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;`

- ❖ Joint struct definition and typedef

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

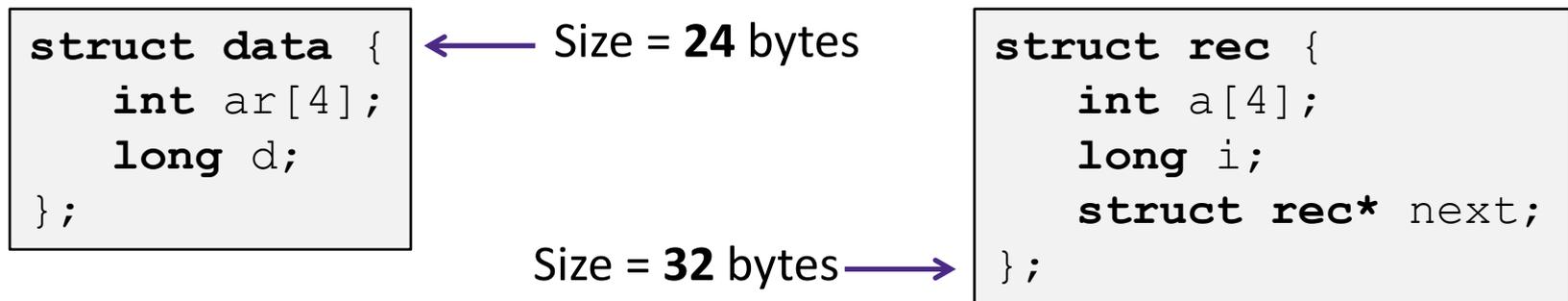
```
struct nm {  
    /* fields */  
};  
typedef struct nm name;  
name n1;
```



```
typedef struct {  
    /* fields */  
} name;  
name n1;
```

# Scope of Struct Definition (Review)

- ❖ 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 (Review)

- ❖ 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 (shorter): `r->i = val;`

- ❖ **In assembly:** register holds address of the first byte
  - Access members with offsets

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

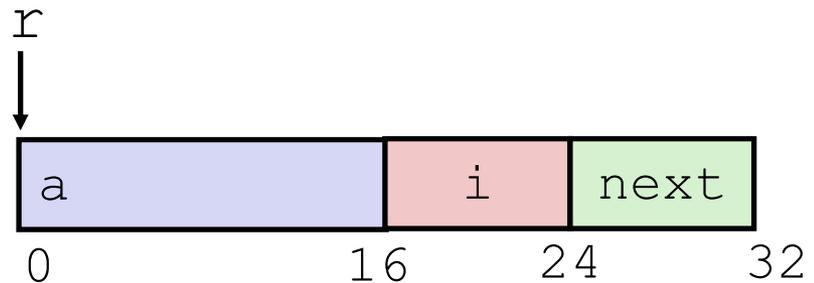
# Java side-note

```
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  $x \rightarrow f$  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  $\leq 8$  bytes (but can point to lots of data)

# Structure Representation (Review)

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

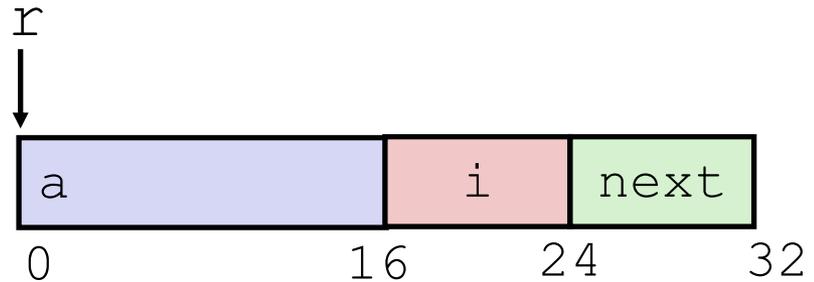


## ❖ Characteristics

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

# Structure Representation (Review)

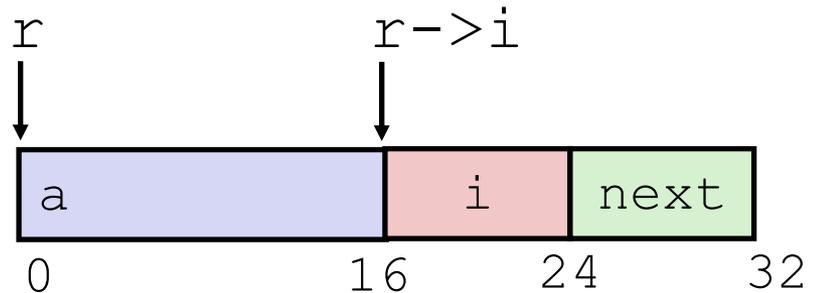
```
struct rec {  
    int a[4];  
    long i;  
    struct rec* next;  
} st, *r = &st;
```



- ❖ Structure represented as block of memory
  - Big enough to hold all 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  $*(r + \text{offset})$

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

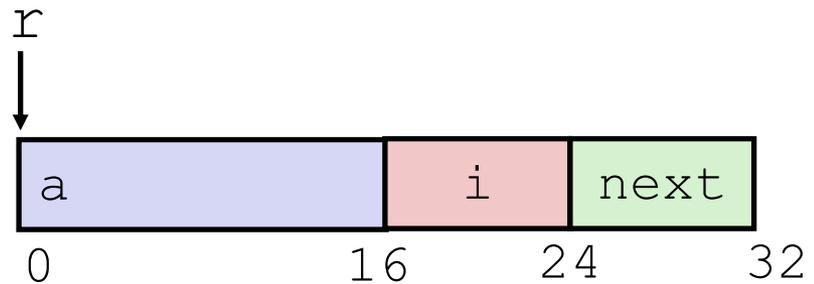
```
# r in %rdi  
movq 16(%rdi), %rax  
ret
```

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

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

# 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);  
}
```

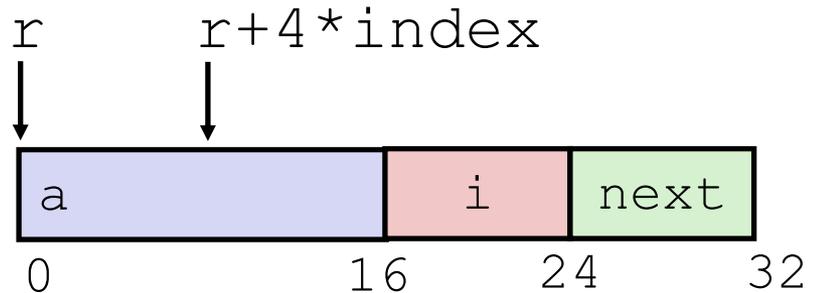
```
# r in %rdi  
leaq 24(%rdi), %rax  
ret
```

# 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];
}

```

$\&(r \rightarrow a[index])$

```

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

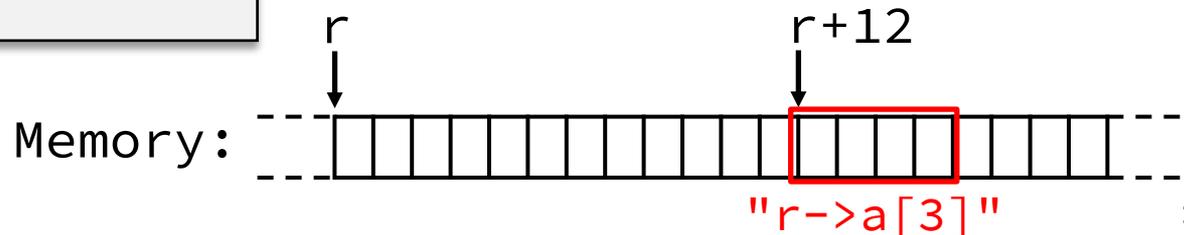
```

# Struct Pointers

- ❖ Pointers store addresses, which all “look” the same
  - Lab 0 Example: struct instance Scores could be treated as array of ints of size 4 via pointer casting
  - A struct pointer doesn't *have* to point to a declared instance of that struct type
- ❖ Different struct fields may or may not be meaningful, depending on what the pointer points to
  - This will be important for Lab 5!

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

```
movl 12(%rdi), %rax  
ret
```



# 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

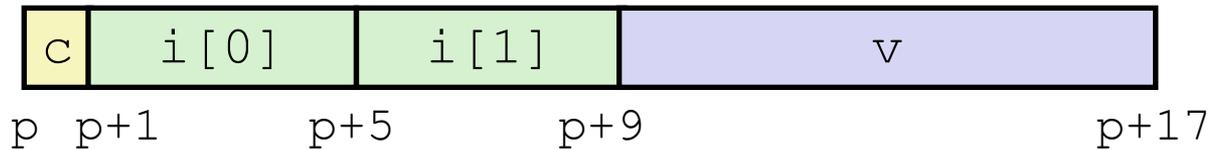
# Memory Alignment in x86-64

- ❖ *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: $\dots 0000_2$

# Structures & Alignment (Review)

## ❖ Unaligned Data

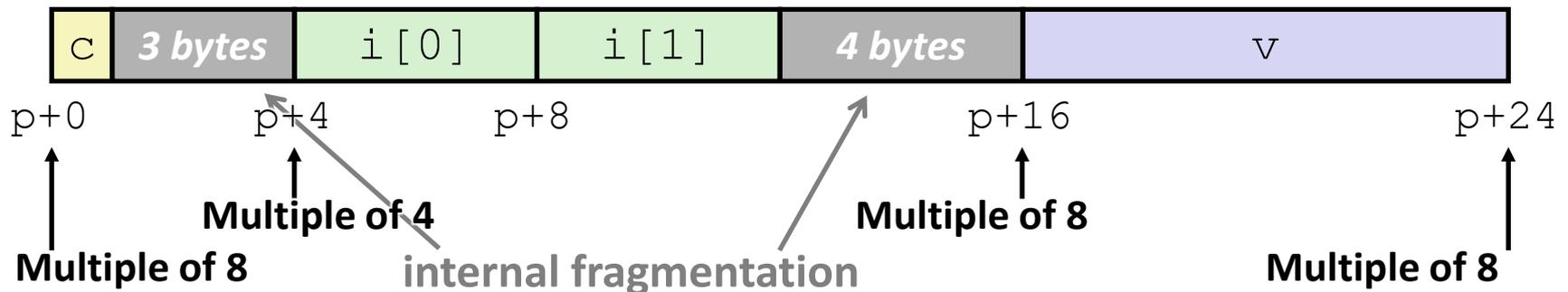


```

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

## ❖ Aligned Data

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



# Satisfying Alignment with Structures (1)

## ❖ Within structure:

- Must satisfy each element's alignment requirement

## ❖ Overall structure placement

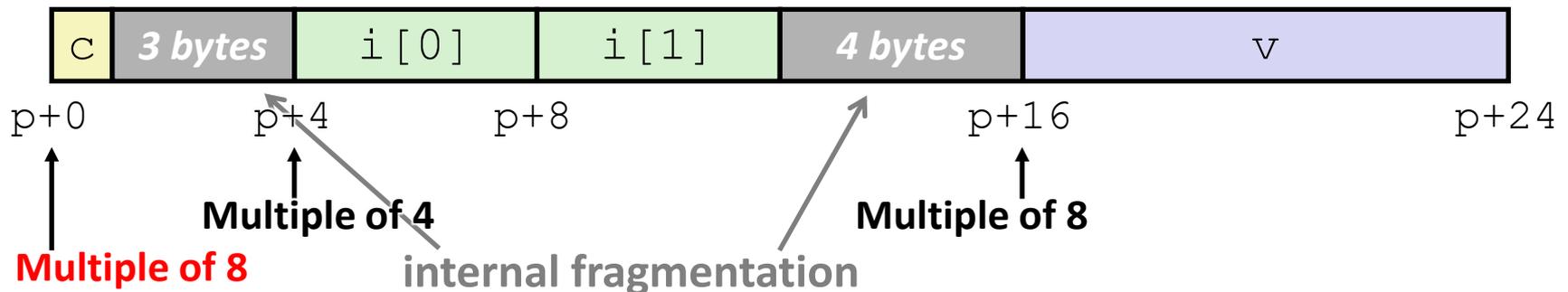
- Each structure has alignment requirement  $K_{\max}$ 
  - $K_{\max}$  = Largest alignment of any element
  - Counts array elements individually as elements

```

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

## ❖ Example:

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

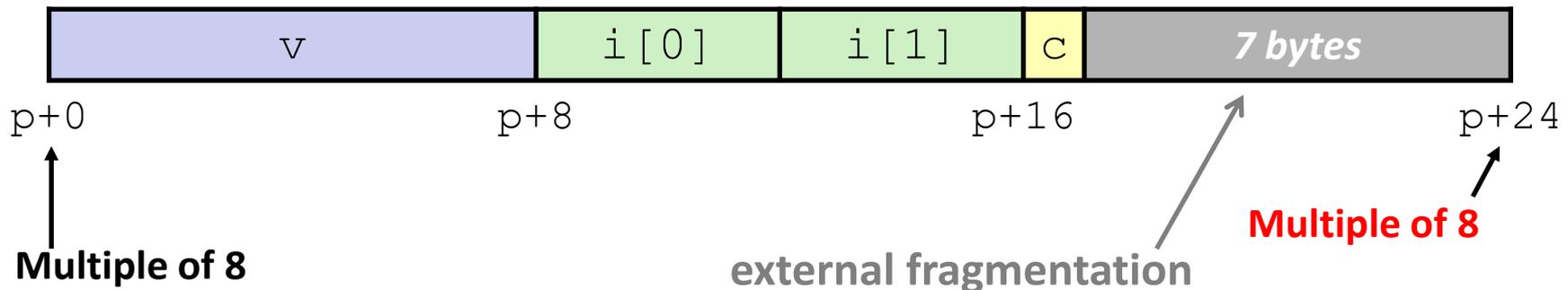


# Satisfying Alignment with Structures (2)

- ❖ 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;
```

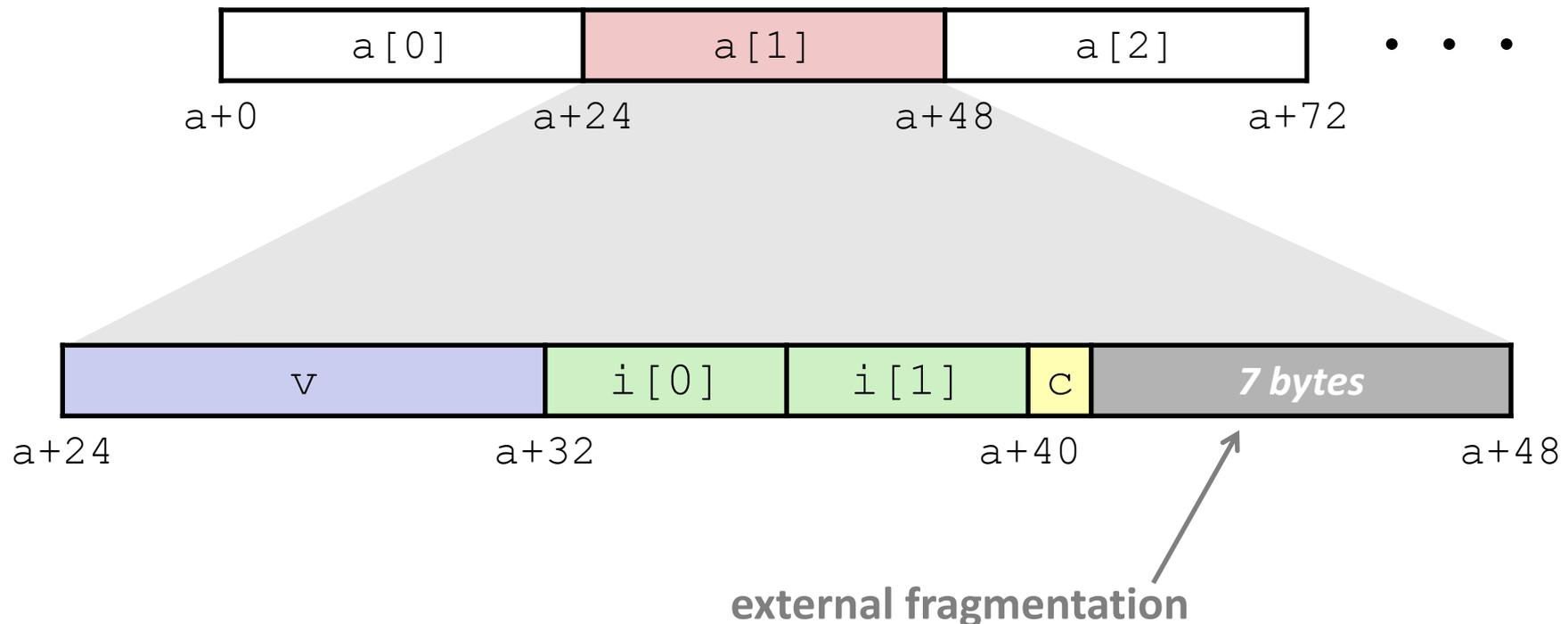
- ❖ For largest alignment requirement  $K_{\max}$ , **overall structure size must be multiple of  $K_{\max}$** 
  - Compiler will add padding **at end** of structure to meet overall structure alignment requirement



# 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 (Review)

- ❖ 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

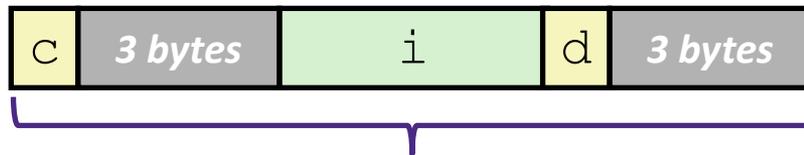
# How the Programmer Can Save Space

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

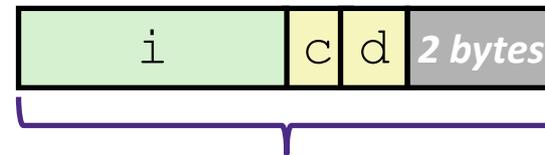
```
struct S4 {  
    char c;  
    int i;  
    char d;  
} st;
```



```
struct S5 {  
    int i;  
    char c;  
    char d;  
} st;
```



12 bytes



8 bytes

# Practice Question

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

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



```
struct new {  
    int i;  
  
    _____;  
  
    _____;  
  
    _____;  
};
```

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

`sizeof(struct old) = 32 B`

`sizeof(struct new) = _____`

- A. 22 bytes
- B. 24 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