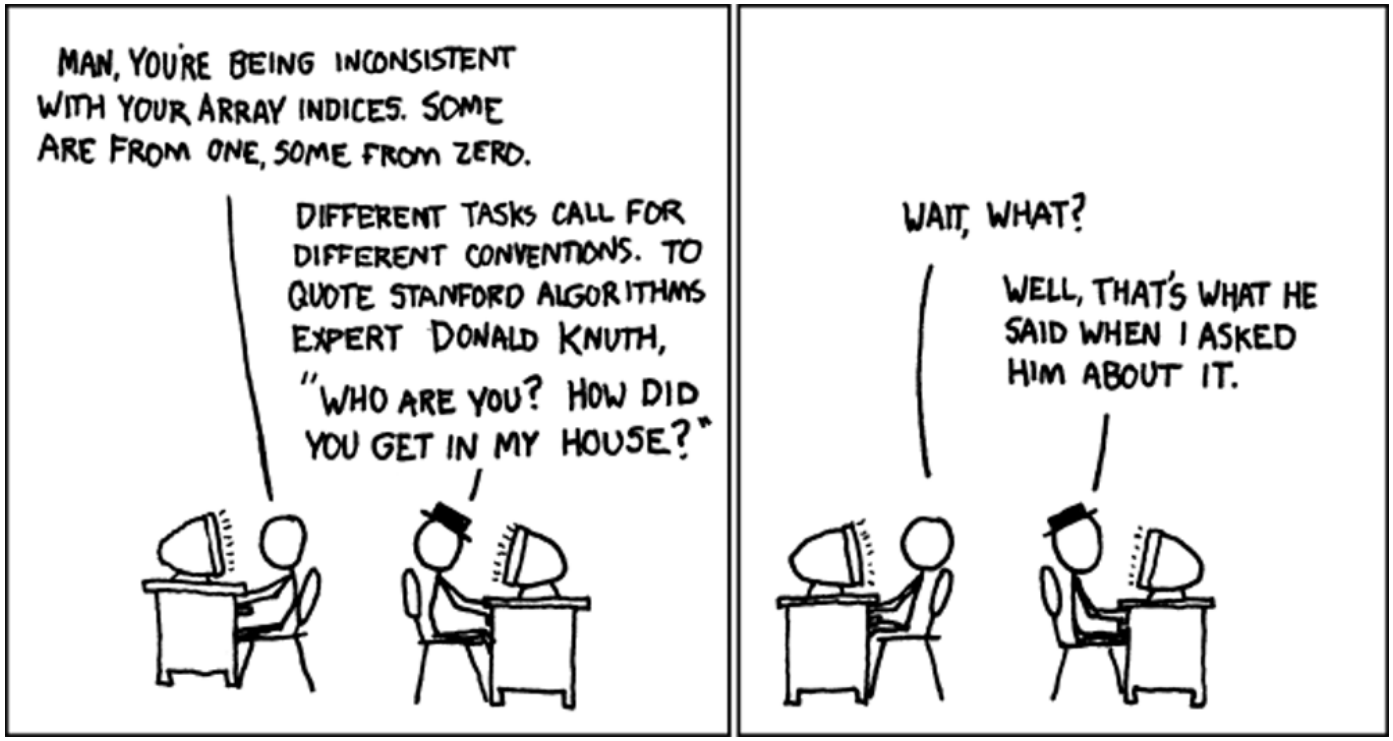
Structs & Alignment

CSE 351 Autumn 2024

Instructor:
Ruth Anderson

Teaching Assistants:
Alexandra Michael
Connie Chen
Chloe Fong
Chendur Jayavelu
Joshua Tan
Nikolas McNamee
Nahush Shrivatsa
Naama Amiel
Neela Kausik
Renee Ruan
Rubee Zhao
Samantha Dreussi
Sean Siddens
Waleed Yagoub



http://xkcd.com/163/

Relevant Course Information

- Lab 2 (x86-64) due TONIGHT, Friday (10/25)
  - Since you are submitting a text file (`defuser.txt`), there won't be any Gradescope autograder output this time
- HW12 due TONIGHT, Friday (10/25) @ 11:59 pm
- HW13 due Monday (10/28) @ 11:59 pm
- HW14 due Wednesday (10/30) @ 11:59 pm
- No Lecture on Fri 11/01 (No HW/Reading due)
- **Midterm Exam**: https://cs.uw.edu/cse351/exams/
  - Take home, on Gradescope
  - Open: Thursday 10/31 at 5pm; Due: Saturday 11/02 at 11:59pm
  - Review in section next week (10/31)

Reading Review

- Terminology:
  - Structs: tags and fields, . and -> operators
  - Typedef
  - Alignment, internal fragmentation, external fragmentation

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?

16 Bytes

- for struct instances (inst), -> for struct pointers (ptr)

- Which of the following statements are syntactically valid?

ptr -> field <-> equiv (*ptr).field

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

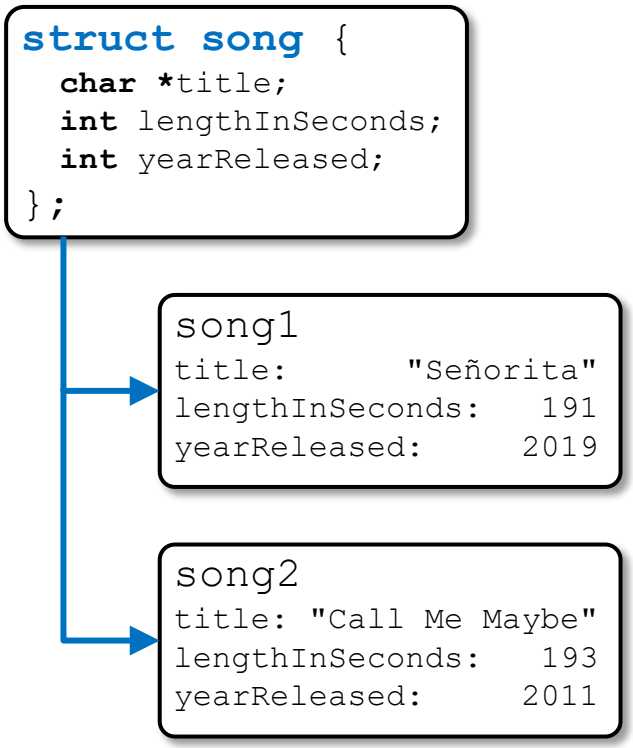
Data Structures in C

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structs**
  - **Alignment**

Structs in C (Review)

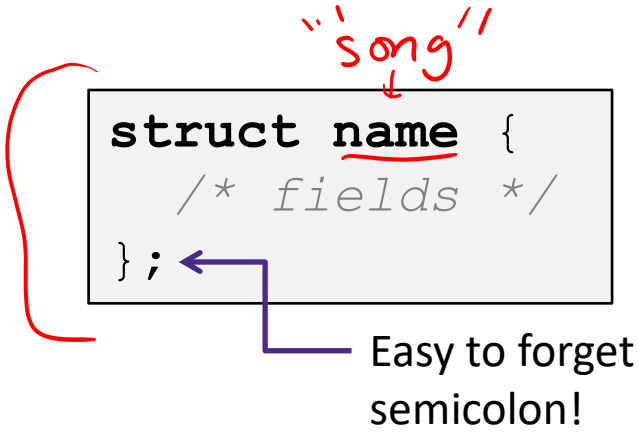
- User-defined structured group of variables, possibly including other structs
  - Similar to Java object, but no methods nor inheritance; just fields
  - Way of defining compound data types

```
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 Definitions (Review)

- Structure definition:
  - Does NOT declare a variable
  - Tells compiler we're defining it and will be using instances of it
  - Variable type is "**`struct name`**"



- Variable declarations like any other data type:

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

instance
pointer
array

- Can also combine struct and instance definitions:

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

Used in review question—this syntax can be difficult to read and do not recommend!

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

```
typedef unsigned long int uli;
unsigned long int x = 12131989;
uli y = 12131989; // can now use it like this!
```

- 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

```
struct data {
    int ar[4];
    long d;
};
```

← Size = **24** bytes

Size = **32** bytes →

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

- Almost always define structs in global scope near the top of your C file
  - Struct definitions follow normal rules of scope
  - Top of singular C files, or if using a header file, place there!

Accessing Structure Members (Review)

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

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

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

32 Bytes

- Given a *pointer* to a struct:

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

8 bytes

We have two options:

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

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`

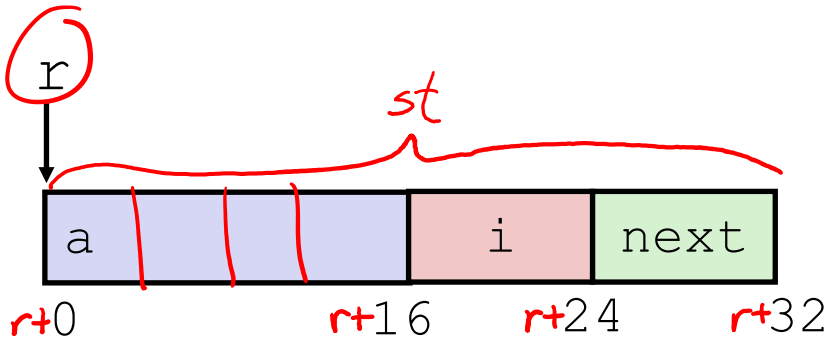
- In Java, almost everything is a pointer ("*reference*") to an object
  - Cannot declare variables or fields that are structs or arrays
  - Always a *pointer* to a struct or array
  - So every Java variable or field is ≤ 8 bytes (but can point to lots of data)

Structure Representation (Review)

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

instance

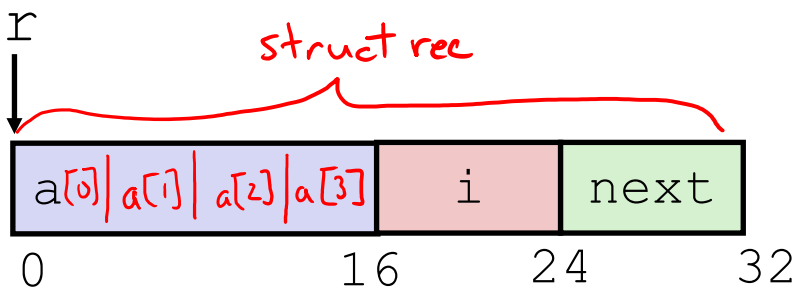
pointer



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

Structure Representation (Review)

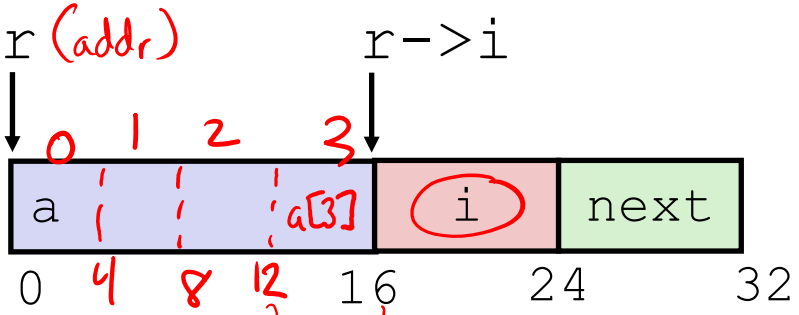
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} st, *r = &st;
```



- Structure represented as block of memory
  - Big enough to hold all 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;
}
```

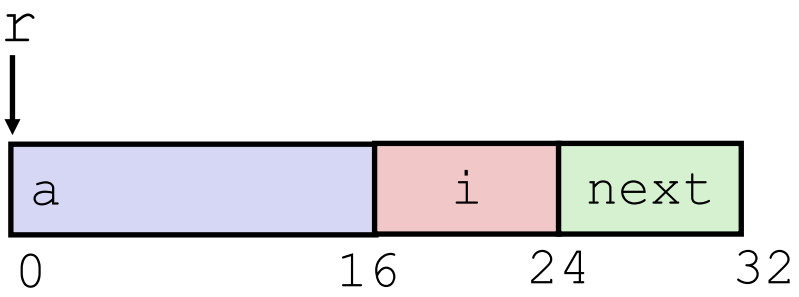
```
# r in %rdi
movq  16(%rdi), %rax
ret
```

```
long 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);
}
```

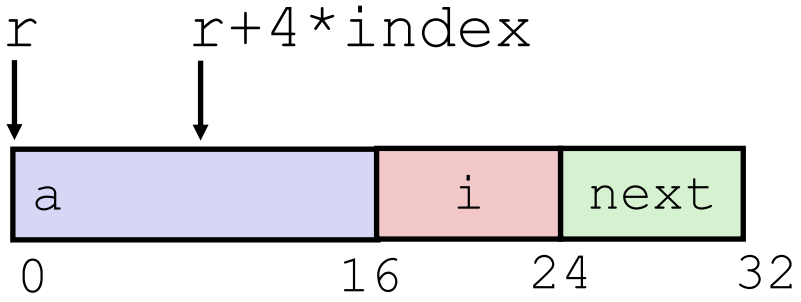
```
# r in %rdi
leaq  16(%rdi), %rax
ret
```

```
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->a[index])`

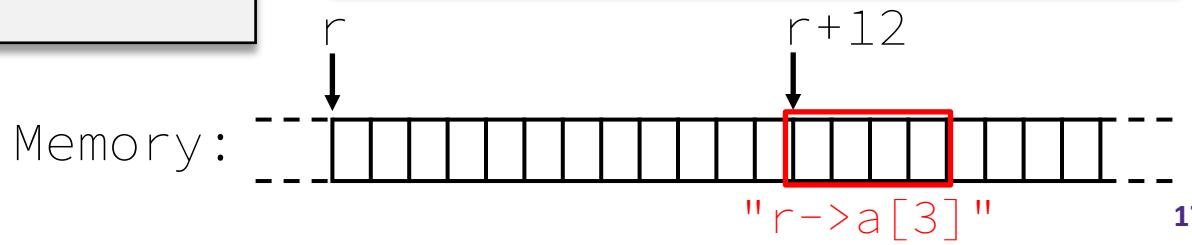
```
  # r in %rdi, index in %rsi
  leaq   (%rdi,%rsi,4), %rax
  ret
```

Struct Pointers

- Pointers store addresses, which all "look" the same
  - Lab 0 Example: struct instance `Scores` could be treated as array of `int`s 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)
    - Important for caching and paging, virtual memory
    - 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

int k = 4

Memory Alignment in x86-64

int

K = 4 Bytes

- *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: ...0_2
4	`int, float`	Lowest 2 bits zero: ...00_2
8	`long, double, *`	Lowest 3 bits zero: ...000_2
16	`long double`	Lowest 4 bits zero: ...0000_2

K=4

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

1 0 0

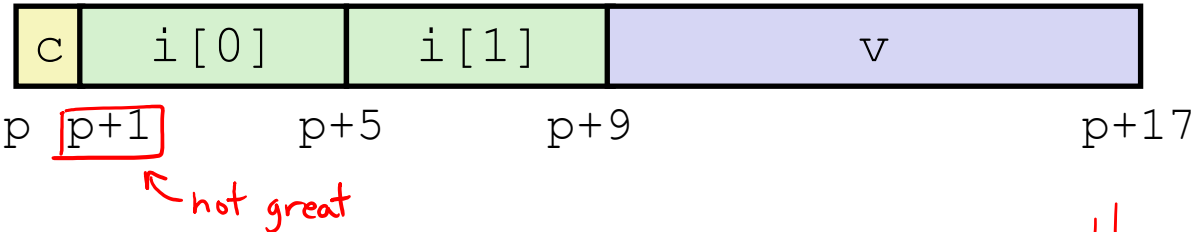
2^3 2^2 2^1 2^0

8 4 2 1

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

Structures & Alignment (Review)

- **Unaligned Data:** just pack all together!



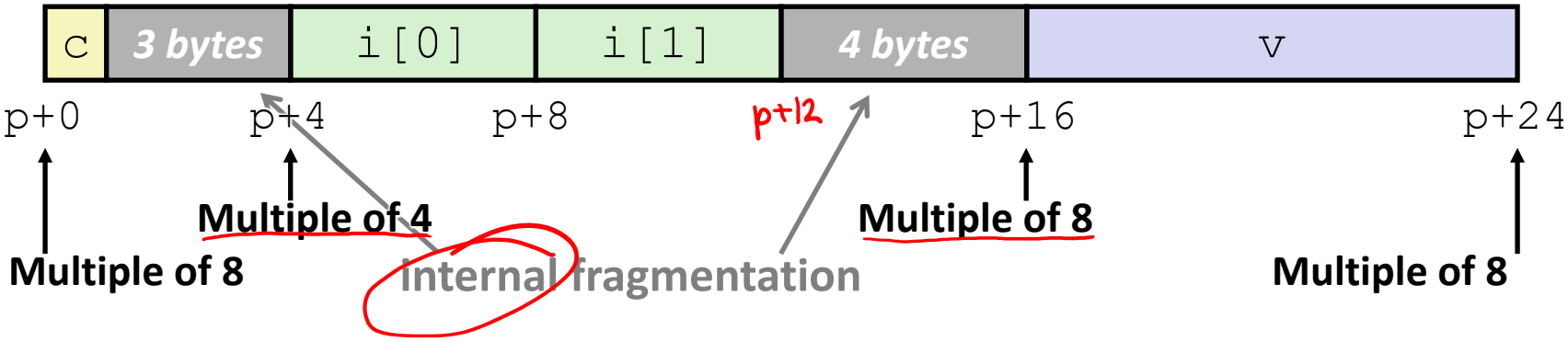

```
struct S1 {          K
① char c;      ← 1
② int i[2];    ← 4
③ double v;    ← 8
} st, *p = &st;
```

$K_{max} = 8$

Compiler will do this

- **Aligned Data:** unused space, but benefits later on.
  - 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

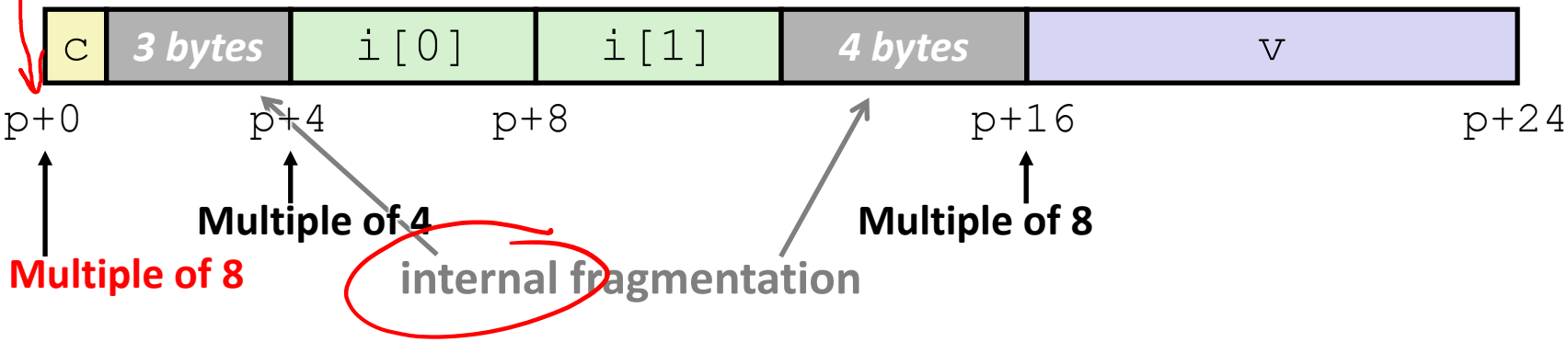
K
1
4
8

```
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

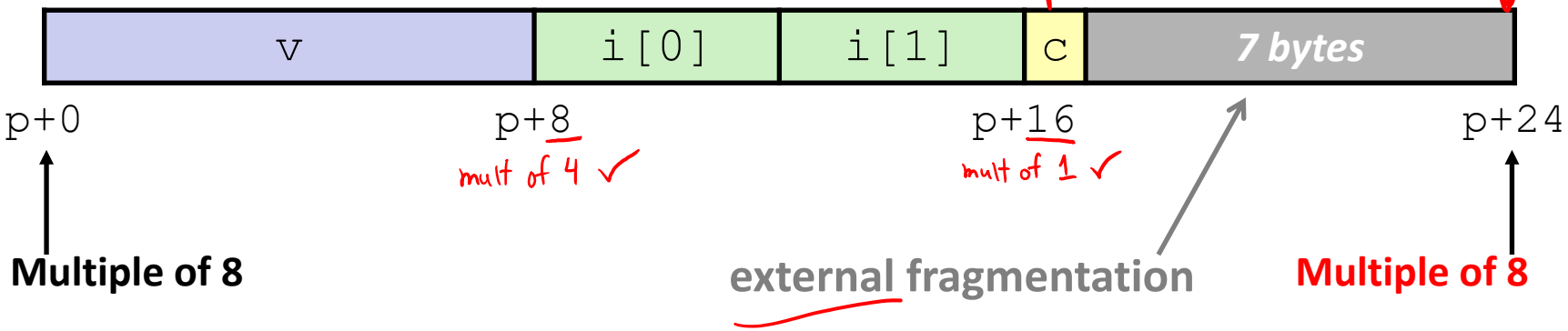


Satisfying Alignment with Structures (2)

- 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;
```

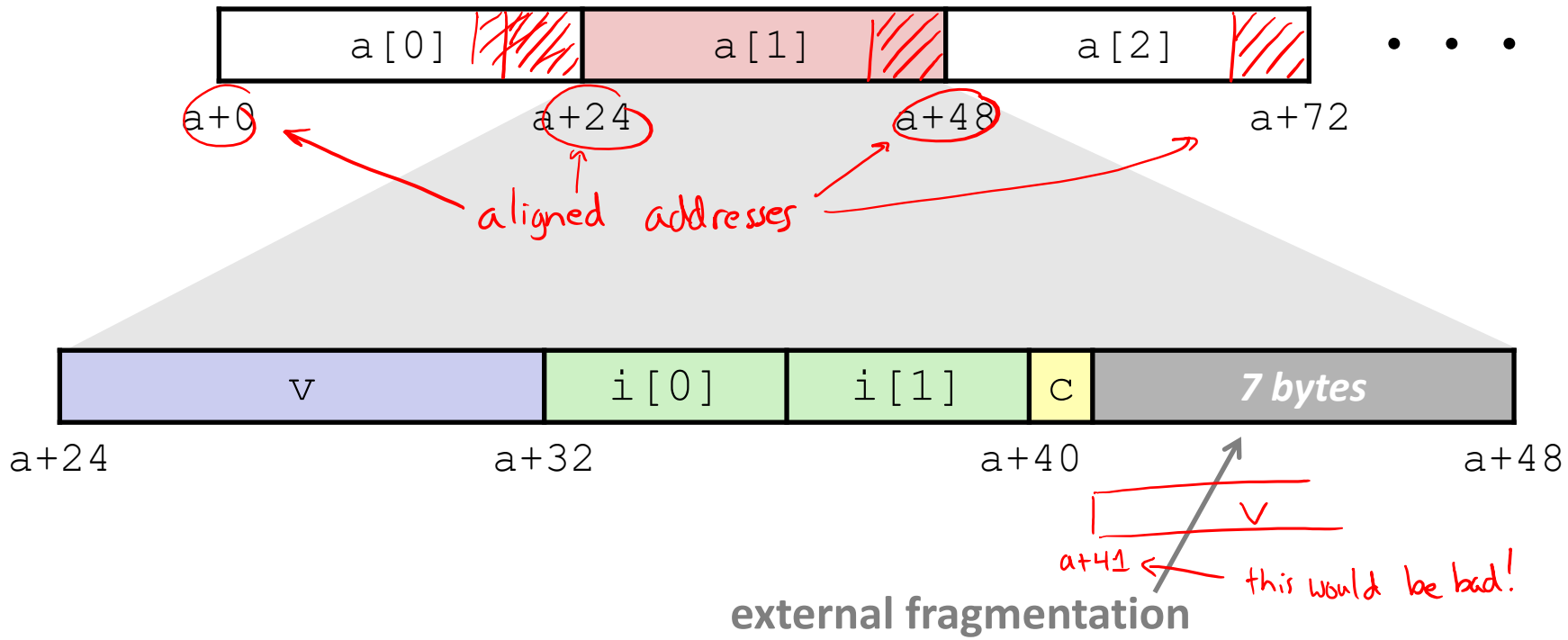
- For largest alignment requirement $K_{\max}$, overall structure size must be multiple of $K_{\max} = 8$
  - Compiler will add padding at end of structure to meet overall structure alignment requirement



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 (Review)

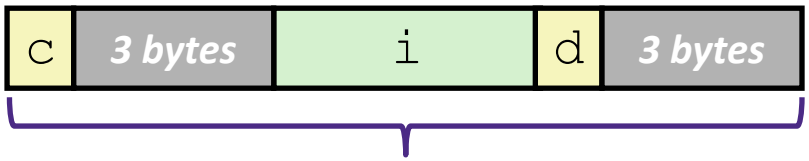
- Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each ***field*** must be aligned *within* the struct *(may insert padding)*
    - `offsetof` can be used to get actual field offset
  - Overall struct must be ***aligned*** according to largest field
  - Total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

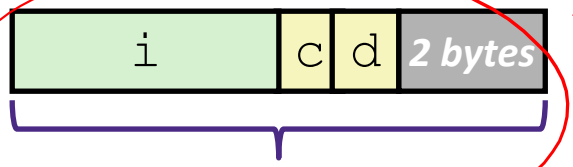
- Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first

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

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



12 bytes



8 bytes

same data, but more efficient!

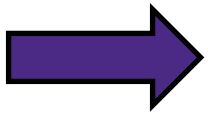
Practice Question

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

K / 4

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

2 / 8 / 4 / $K_{max} = 8$



```
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?

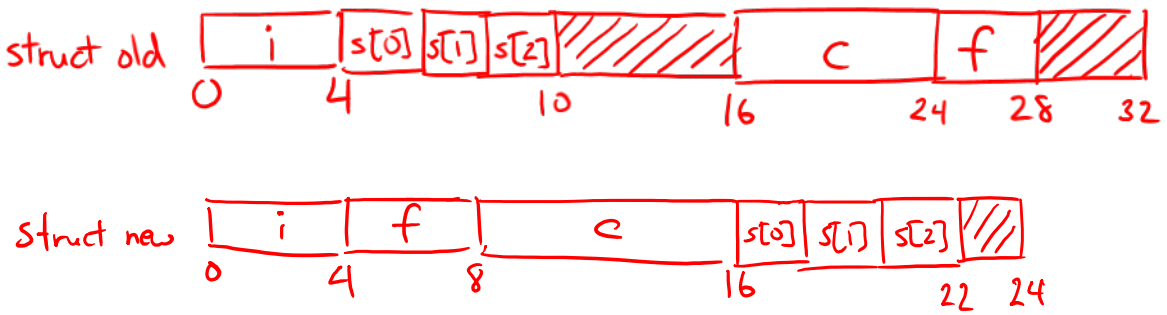
`sizeof(struct old)` = 32 B `sizeof(struct new)` = _____

A. 22 bytes

B. 24 bytes

C. 28 bytes

D. 32 bytes



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