Structs & Alignment

CSE 351 Spring 2022

Instructor:

Ruth Anderson

Teaching Assistants:

Melissa Birchfield

Jacob Christy

Alena Dickmann

Kyrie Dowling

Ellis Haker

Maggie Jiang

Diya Joy

Anirudh Kumar

Jim Limprasert

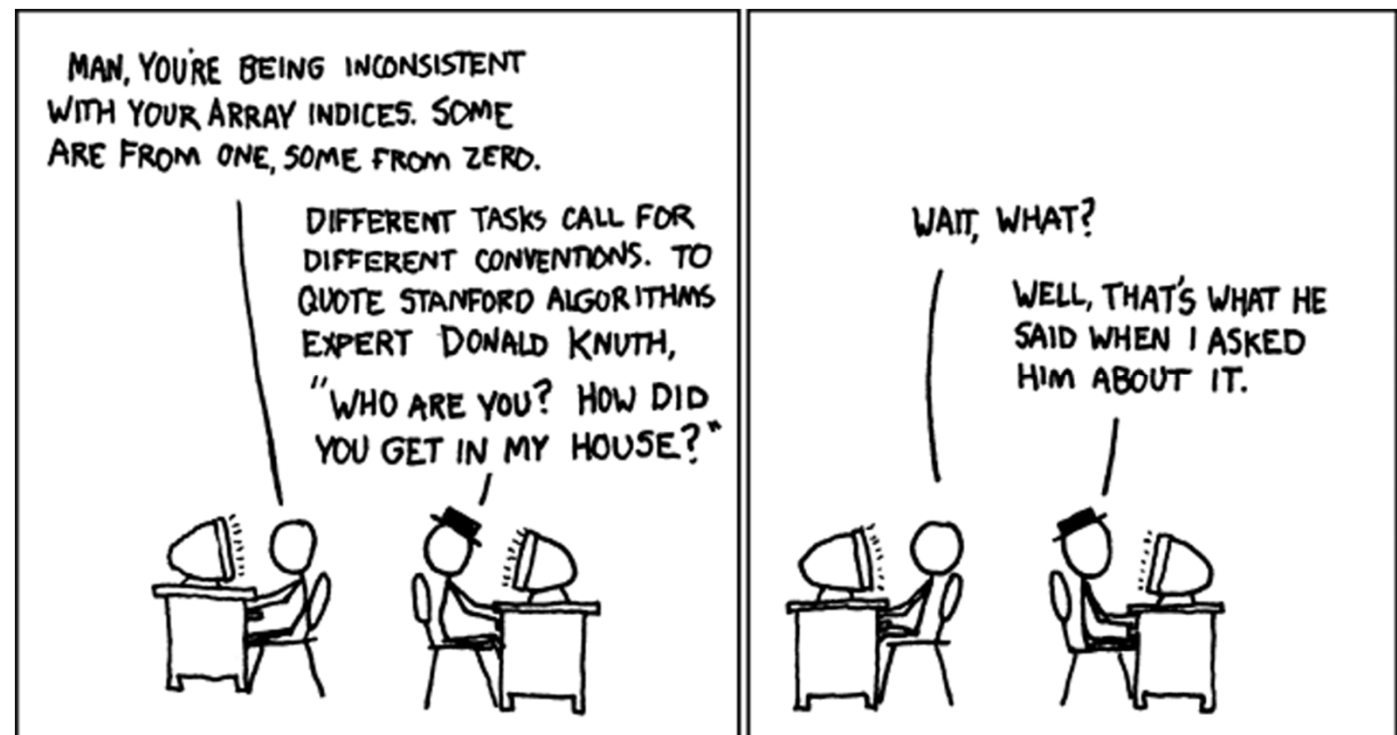Armin Magness

Hamsa Shankar

Dara Stotland

Jeffery Tian

Assaf Vayner

Tom Wu

Angela Xu

Effie Zheng

http://xkcd.com/163/

Relevant Course Information

- ❖ Lab 2 (x86-64) due Friday (4/29)
    - ▪ Learn to read x86-64 assembly and use GDB
    - ▪ Optional GDB Tutorial on Ed Lessons
    - ▪ Since you are submitting a text file (`defuser.txt`), there won't be any Gradescope autograder output this time

- ❖ hw13 – due Monday 5/02
    - ▪ Based on the next two lectures, longer than normal

- ❖ Midterm (take home, 5/02-5/04)
    - ▪ Midterm review problems in section this week
    - ▪ Released 11:59pm on Mon 5/02, due 11:59pm Wed 5/04
    - ▪ See email sent to class, Ed Post, and exams page

Reading Review

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

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
      c.getMPG();
```

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
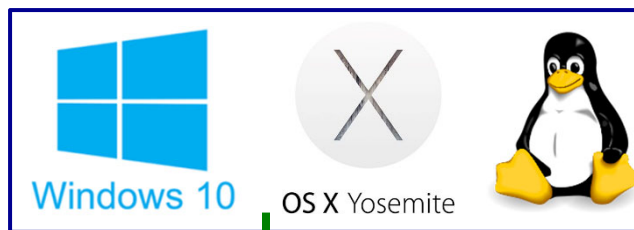
Assembly
language:

```
get_mpg:
      pushq     %rbp
      movq      %rsp, %rbp
      ...
      popq      %rbp
      ret
```
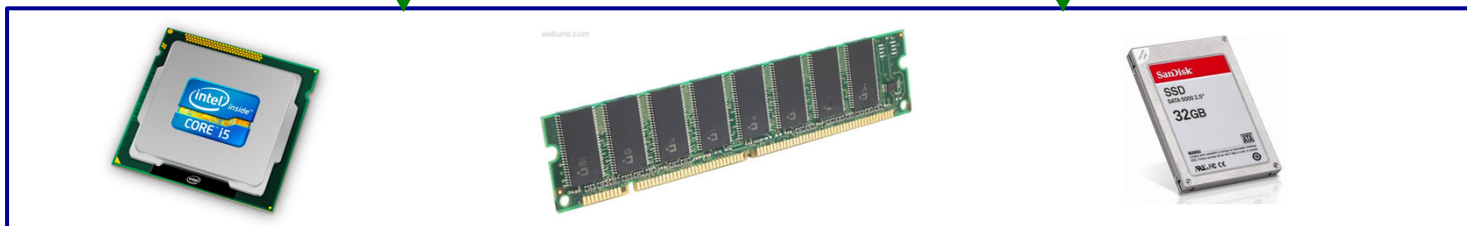
OS:

Machine
code:

```
01110100000011000
10001101000001000000010
1000100111000010
1100000111111101000011111
```


Windows 10   OS X Yosemite

Computer
system:

Review Questions

tag

```
struct ll_node {
8B  long data;
8B  struct ll_node⊛ next;          } fields
Kmax=8  } n1, n2;
```
two instances

❖ How much space does (in bytes) does an instance of
   `struct ll_node` take?   16 Bytes

• for struct instances,   → for struct pointers
         (inst)   (ptr)

❖ Which of the following statements are syntactically
   valid?   ptr→field   (*ptr).field

✓ ▪ *inst* ✓ *ptr* `n1.next = &n2;` *ptr*

✗ ▪ *inst* ✗ `n2->data = 351;`

✓ ▪ *inst* ✓ *ptr* ✓ *long* `n1.next->data = 333;`

✗ ▪ *ptr* ✓ *ptr* ✓ *ptr* ✗ `(&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 = "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
```

7
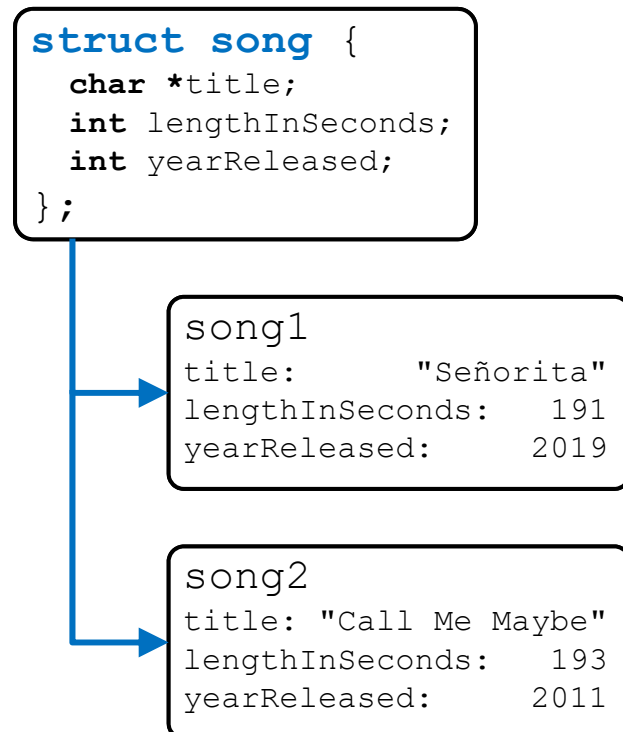
Struct Definitions (Review)

❖ Structure definition:
  - Does NOT declare a variable
  - Variable type is "**struct name**"

your choice

```
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;
```

← *this is the data type (like int)*

Typedef in C (Review)

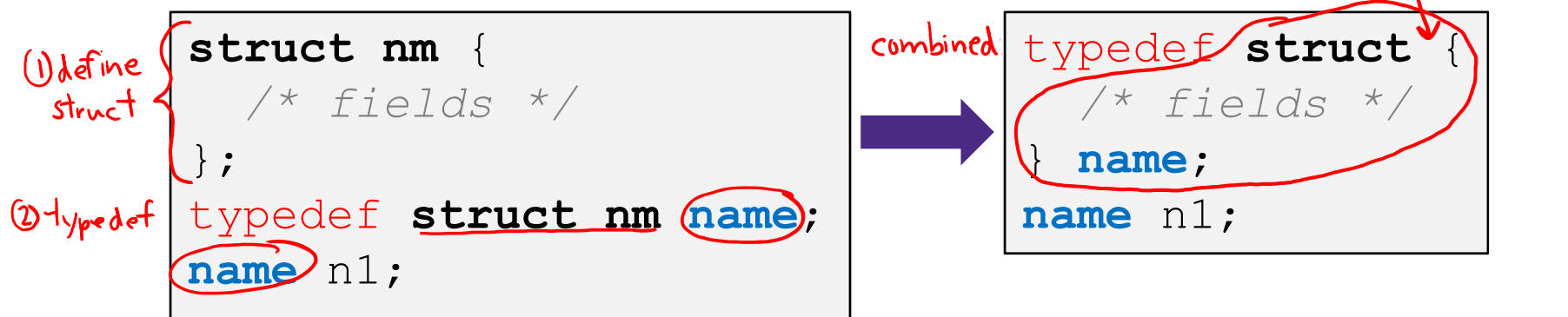❖ A way to create an *alias* for another data type:
`typedef <data type> <alias>;`

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

  ▪ *e.g.* `typedef unsigned long int uli;`

      data type   alias

❖ Joint struct definition and typedef

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

      unnamed!

      combined

```
①define   struct nm {
  struct    /* fields */
          };
②typedef  typedef struct nm name;
          name n1;
```

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

9

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

```
struct rec {
    int a[4];
    long i;
    struct rec* next;
};
```
Size = **32** bytes ⟶
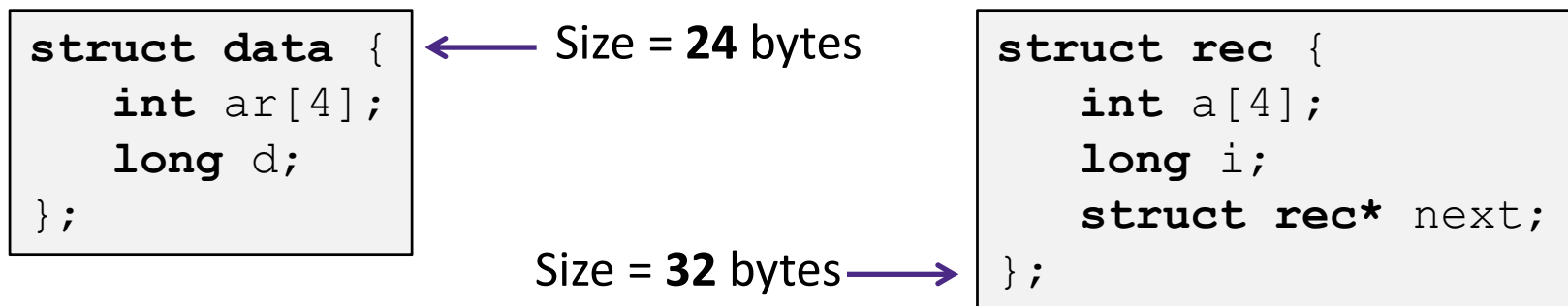
❖ Almost always define structs in global scope near the top of your C file

- 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;
```

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

❖ 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: ① dereference (get instance)
                                  ② access field
                                  `(*r).i = val;`
- Use `->` operator (shorter):   `r->i = val;`   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;
```
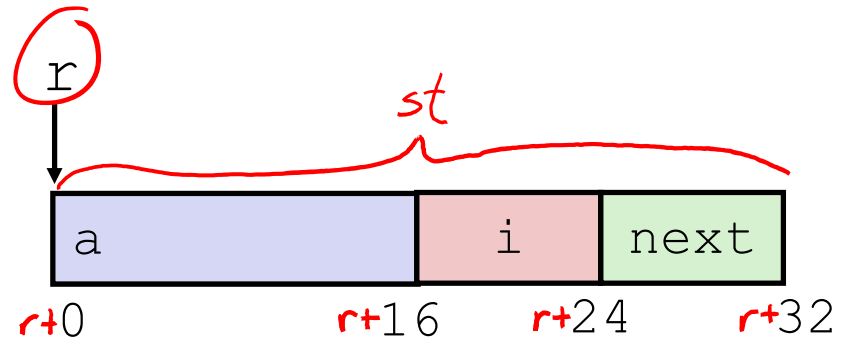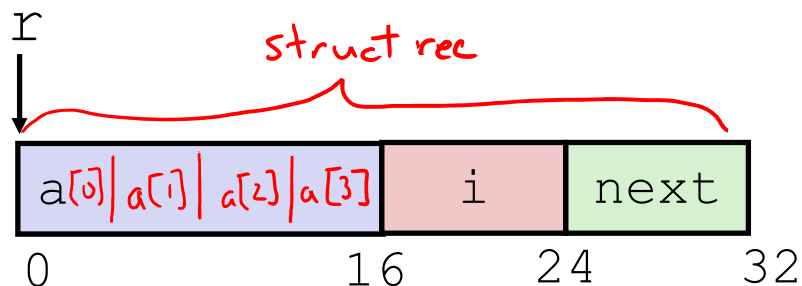

* Characteristics
  * Contiguously-allocated region of memory
  * Refer to members within structure by names
  * Fields may be of different types

Structure Representation (Review)

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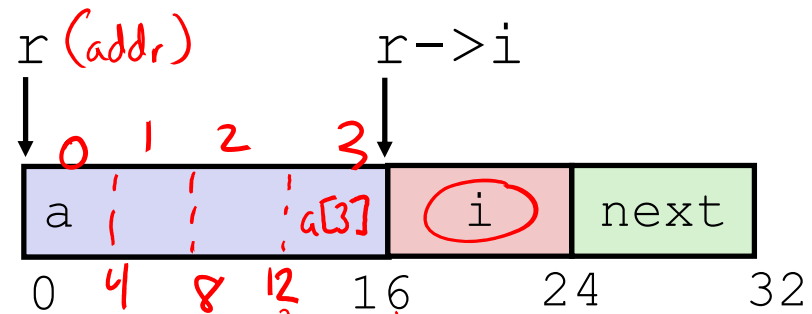

- ❖ **Structure represented as block of memory**
  - ▪ Big enough to hold all of the fields
- ❖ Fields ordered according to declaration order
  - ▪ Even if another ordering would be more compact
- ❖ Compiler determines overall size + positions of fields
  - ▪ Machine-level program has no understanding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

Accessing a Structure Member

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

r (addr)   r->i

```
    0   1   2   3
  ┌───┬───┬───┬───┬─────────┬─────────┐
  │ a │   │   │a[3]│    i    │  next   │
  └───┴───┴───┴───┴─────────┴─────────┘
  0   4   8  12  16        24        32
```

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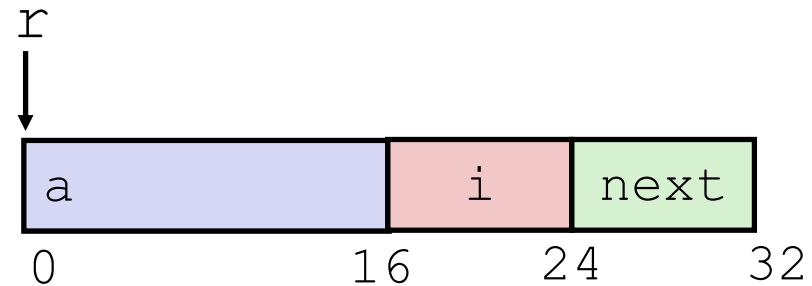
```
long get_a3(struct rec* r) {
  return r->a[3];
}
```

```
# 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;
```

r

a		i	next
0	16	24	32

```
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
```
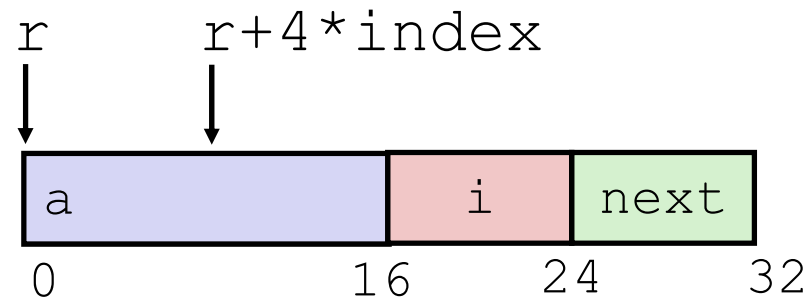
Generating Pointer to Array Element

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

r   r+4*index

a		i	next
0		16	24	32

❖ **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 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
```

r   r+12

Memory:

"r->a[3]"

18
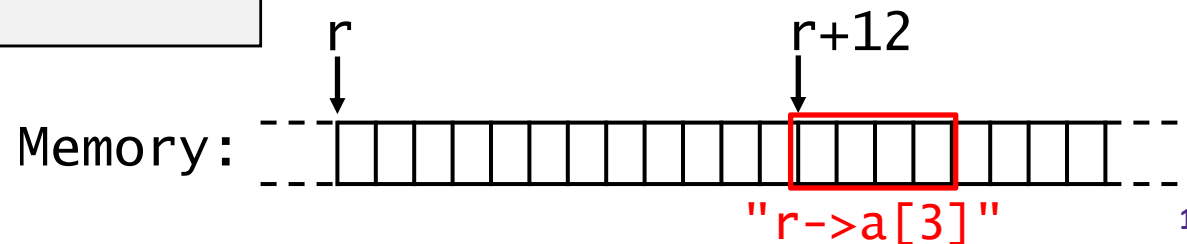
Alignment Principles

❖ 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 if its ~~address~~ is a multiple of its size)

❖ *Aligned* means that any primitive object of K bytes must have an address that is a multiple of K

❖ Aligned addresses for data types:

sizes

$$\ldots \frac{01}{8} \frac{00}{42} \frac{0}{1}$$
$$2^3\ 2^2\ 2^1\ 2^0$$

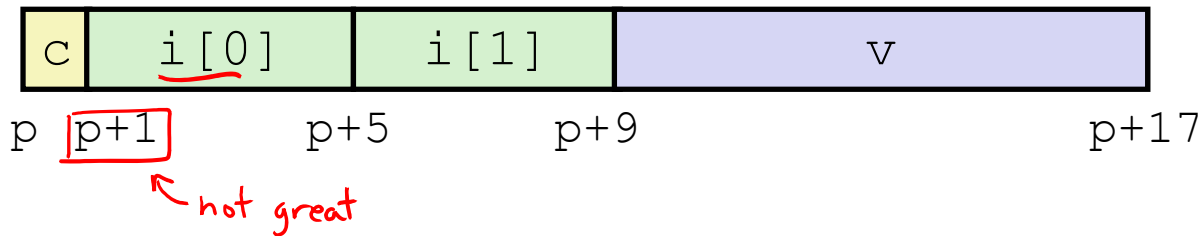| K	Type	Addresses
1	`char`	No restrictions
2	`short`	Lowest bit must be zero: $\ldots 0_2$
4	`int, float`	Lowest 2 bits zero: $\ldots 00_2$
8	`long, double, *`	Lowest 3 bits zero: $\ldots 000_2$
16	`long double`	Lowest 4 bits zero: $\ldots 0000_2$
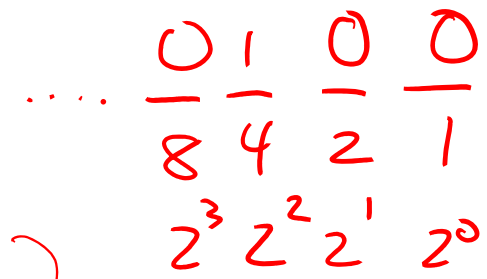
lowest $\log_2(K)$ bits should be 0

"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 → all zeros in lowest $\log_2(K)$ bits.

Structures & Alignment (Review)

❖ <u>Unaligned Data</u>

c	i[0]	i[1]	v

p p+1   p+5   p+9   p+17

↖ not great

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

$K_{max} = 8$

24 B total

Compiler will do this

❖ <u>Aligned Data</u> 😊

■ Primitive data type requires K bytes

■ Address must be multiple of K

c	3 bytes	i[0]	i[1]	4 bytes	v

p+0   p+4   p+8   p+12   p+16   p+24

Multiple of 4   **Multiple of 8**

Multiple of 8   internal fragmentation   **Multiple of 8**

21

Satisfying Alignment with Structures (1)

K
1
4
8

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

- ❖ <u>Within</u> structure:
  - ▪ Must satisfy each element's alignment requirement
- ❖ <u>Overall</u> structure placement
  - ▪ Each <u>structure</u> has alignment requirement K_{max}
    - • K_{max} = Largest alignment of any element
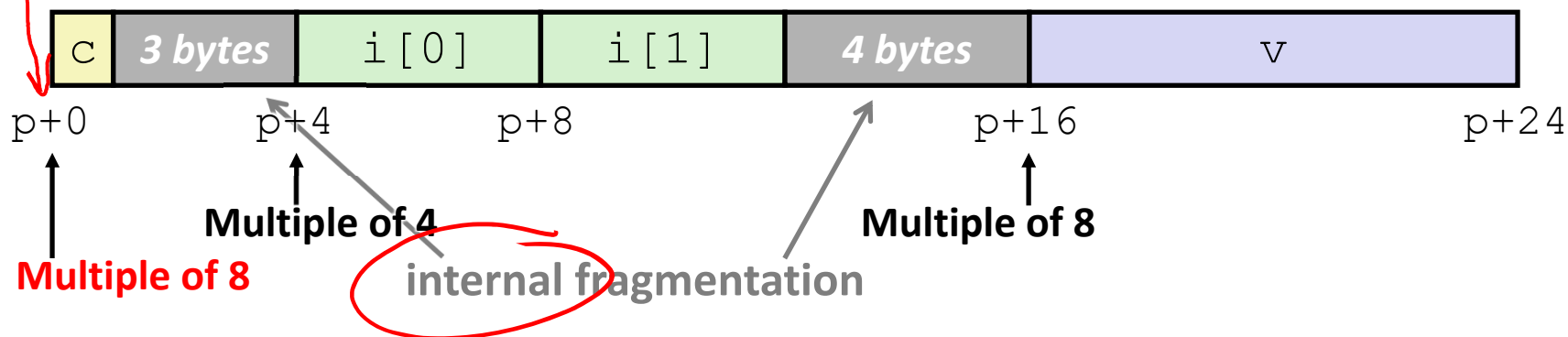   - • Counts array elements individually as elements

$K_{max} = 8$

alignment requirement of starting addr

- ❖ Example:
  - ▪ K_{max} = 8, due to `double` element

c	*3 bytes*	`i[0]`	`i[1]`	*4 bytes*	v

p+0   p+4   p+8   p+16   p+24

Multiple of 4

Multiple of 8

Multiple of 8   **internal fragmentation**

Satisfying Alignment with Structures (2)

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

❖ Can find offset of individual fields using `offsetof()`

  ▪ Need to `#include <stddef.h>`

  ▪ <u>Example</u>: `offsetof(struct S2, c)` returns 16

❖ 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

not a mult of 8 ✗

pad

p+17

v	i[0]	i[1]	c	*7 bytes*

p+0   p+8   p+16   p+24

mult of 4 ✓   *mult of 1* ✓

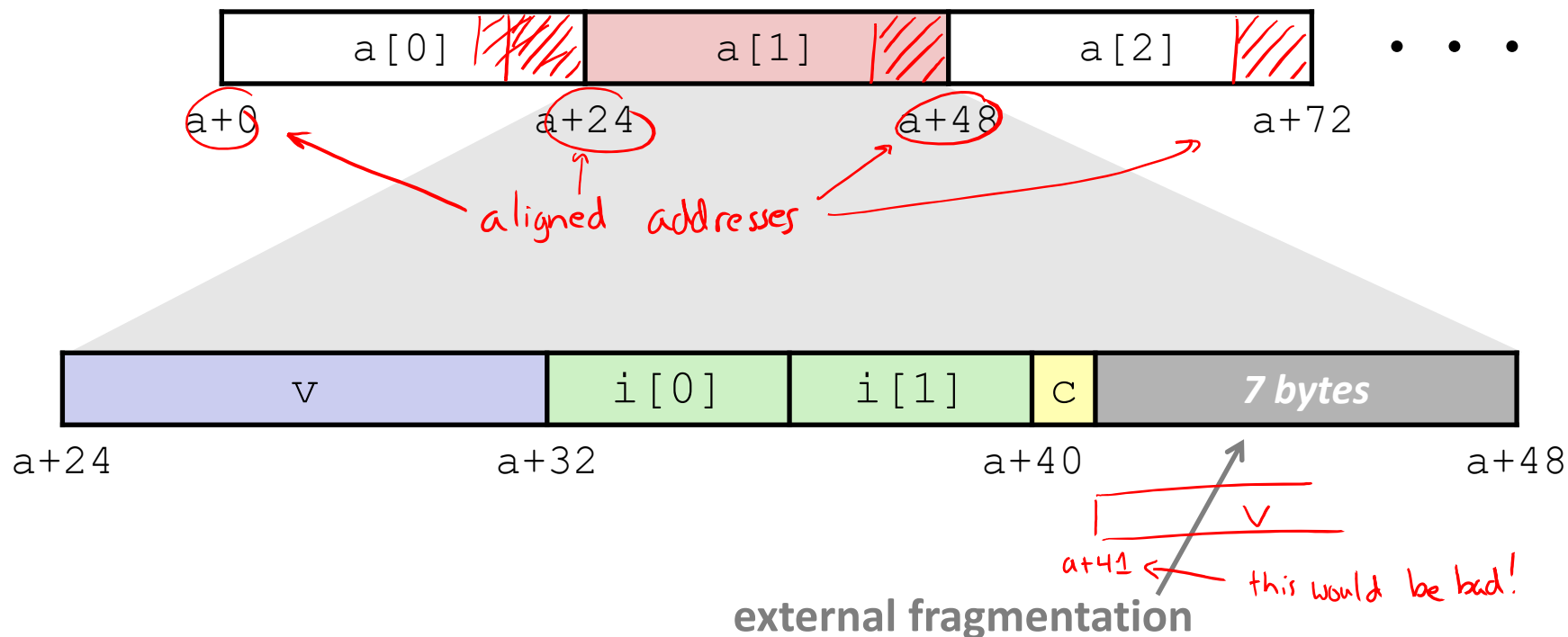**Multiple of 8**   external fragmentation   **Multiple of 8**
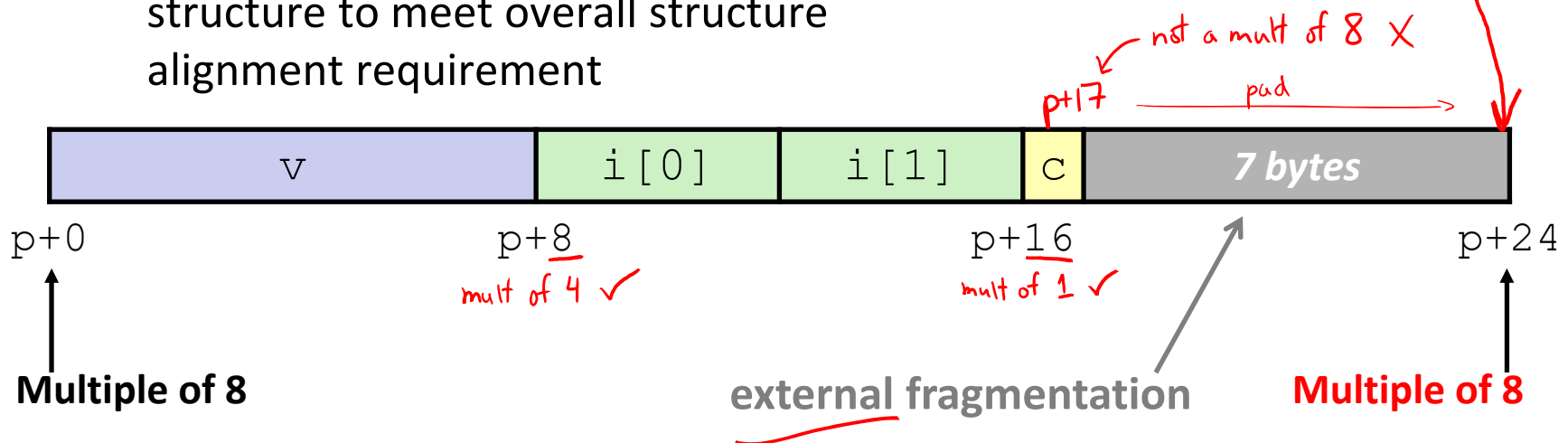
Arrays of Structures

❖ 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];
```
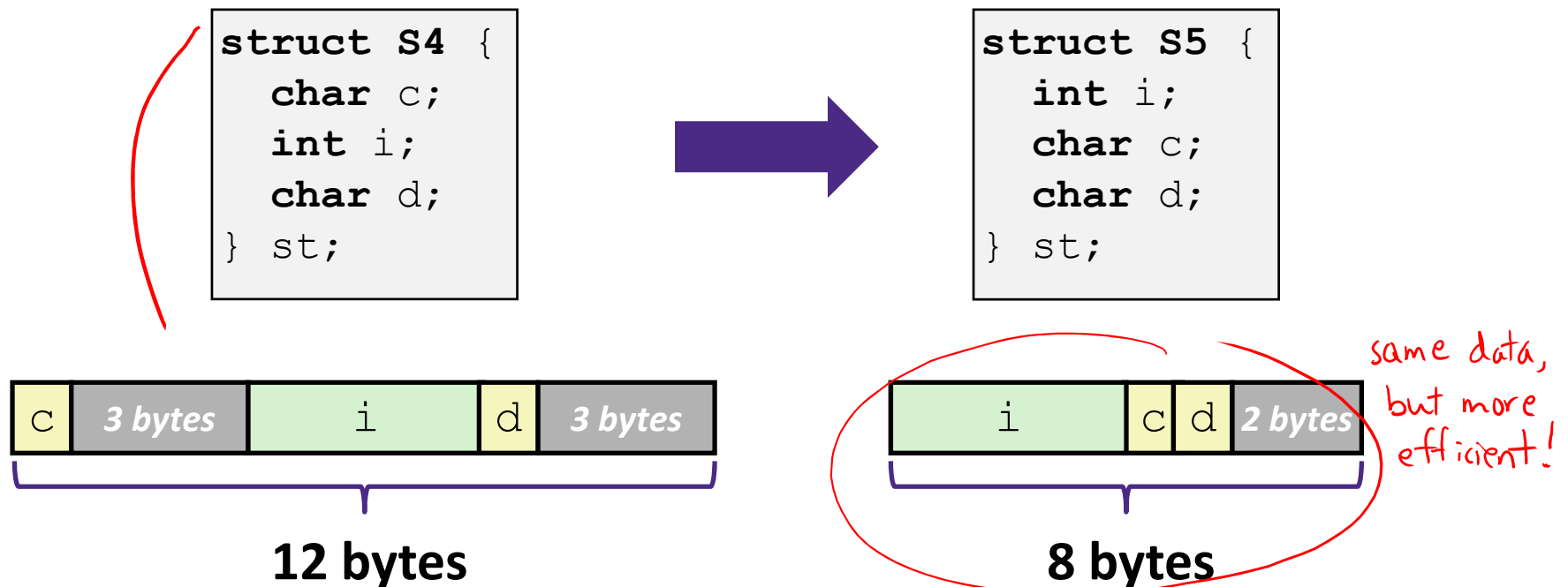

aligned addresses

external fragmentation

this would be bad!

a+41

7 bytes

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)

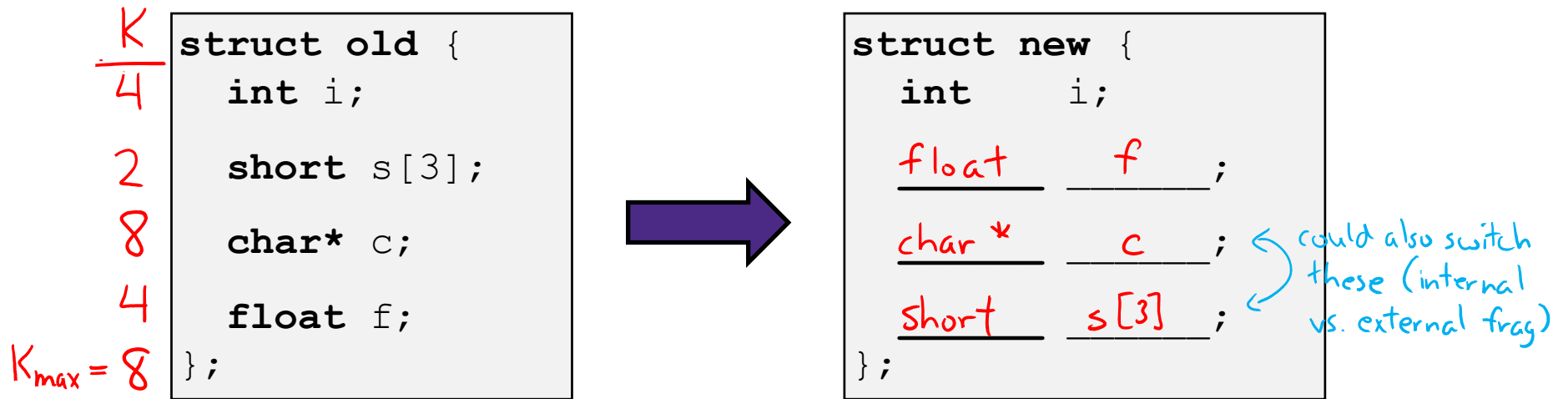 - `sizeof` should be used to get true size of structs

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;
```

c	*3 bytes*	i	d	*3 bytes*

12 bytes

i	c	d	*2 bytes*

same data, but more efficient!

8 bytes

Practice Question

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

K/4

2

8

4

$K_{max} = 8$

```
struct old {
    int i;

    short s[3];

    char* c;

    float f;
};
```

→

```
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**

struct old | i | s[0] s[1] s[2] ///// | c | f ///
0   4   10   16   24 28 32

struct new | i | f | c | s[0] s[1] s[2] // |
0   4   8   16   22 24
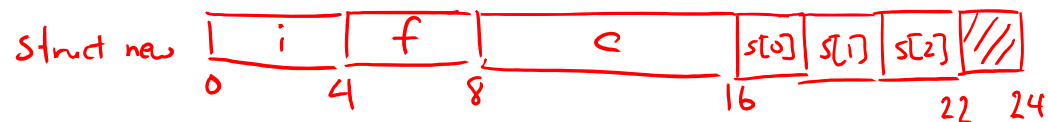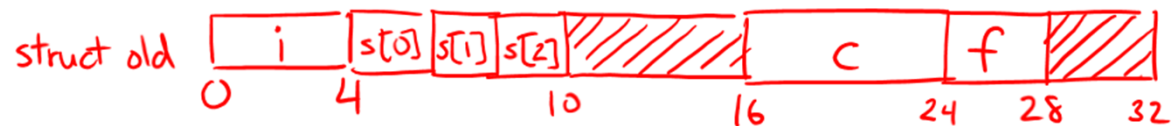
Summary

- ❖ 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