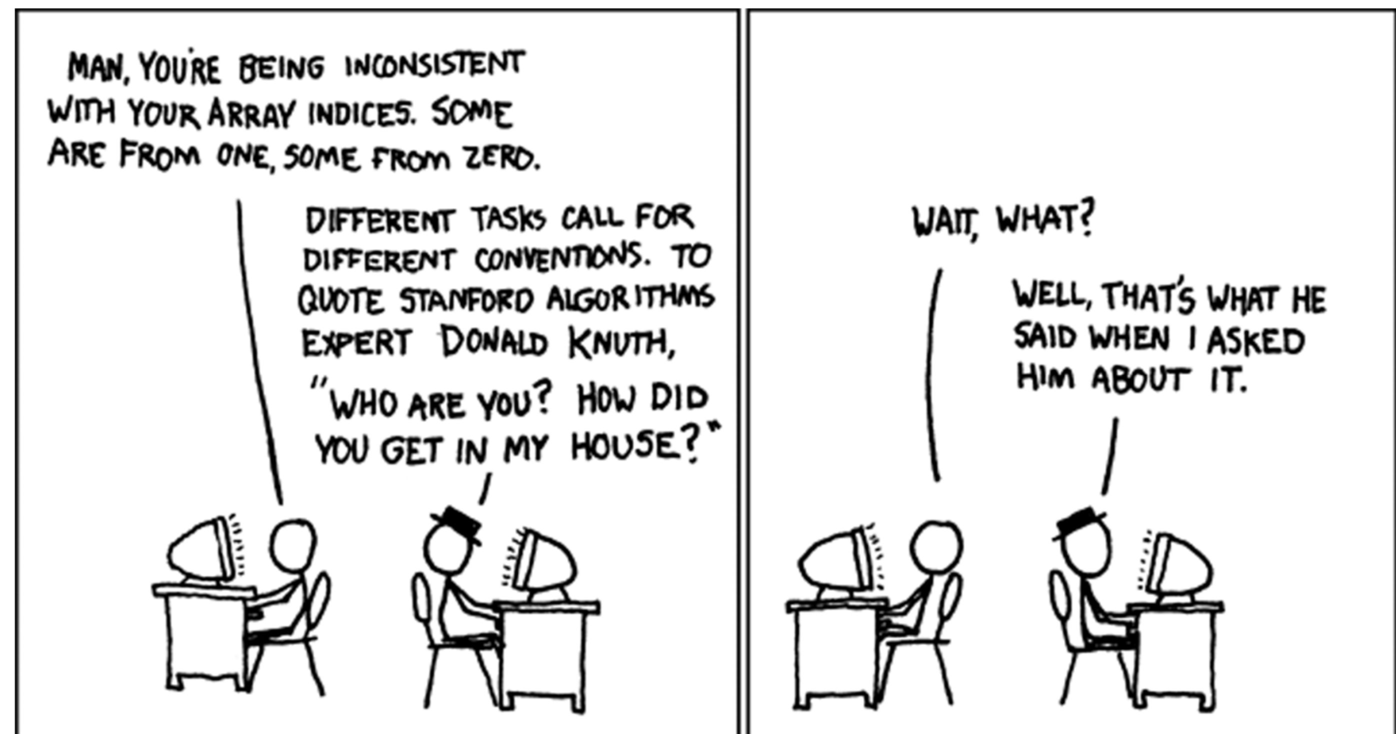
Structs & Alignment

CSE 351 Spring 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Alex Olshanskyy
Rehaan Bhimani
Callum Walker
Chin Yeoh
Diya Joy
Eric Fan
Edan Sneh
Jonathan Chen
Jeffery Tian
Millicent Li
Melissa Birchfield
Porter Jones
Joseph Schafer
Connie Wang
Eddy (Tianyi) Zhou



http://xkcd.com/163/

Administrivia

- Mid-quarter survey due TONIGHT, (4/29) on Canvas
- Lab 2 (x86-64) due Friday (5/01)
  - Optional GDB Tutorial homework on Gradescope
  - Since you are submitting a text file (`defuser.txt`), there won't be any Gradescope autograder output this time
  - Extra credit needs to be submitted to the extra credit assignment
- **You must log on with your @uw google account to access!!**
  - **Google doc** for 11:30 Lecture: https://tinyurl.com/351-04-29A
  - **Google doc** for 2:30 Lecture: https://tinyurl.com/351-04-29B

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();
```

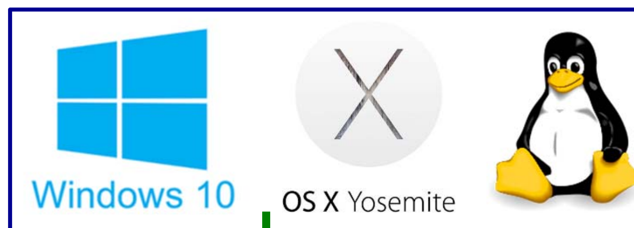
Assembly language:

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

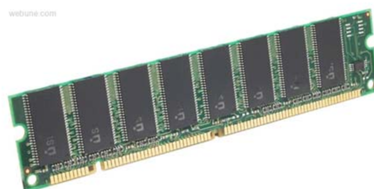
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



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

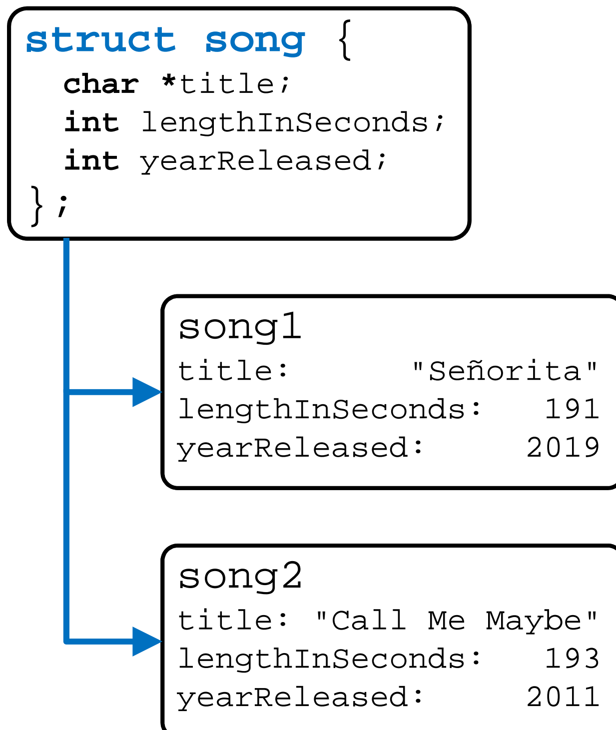
Data Structures in Assembly

- 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 Definitions

- 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

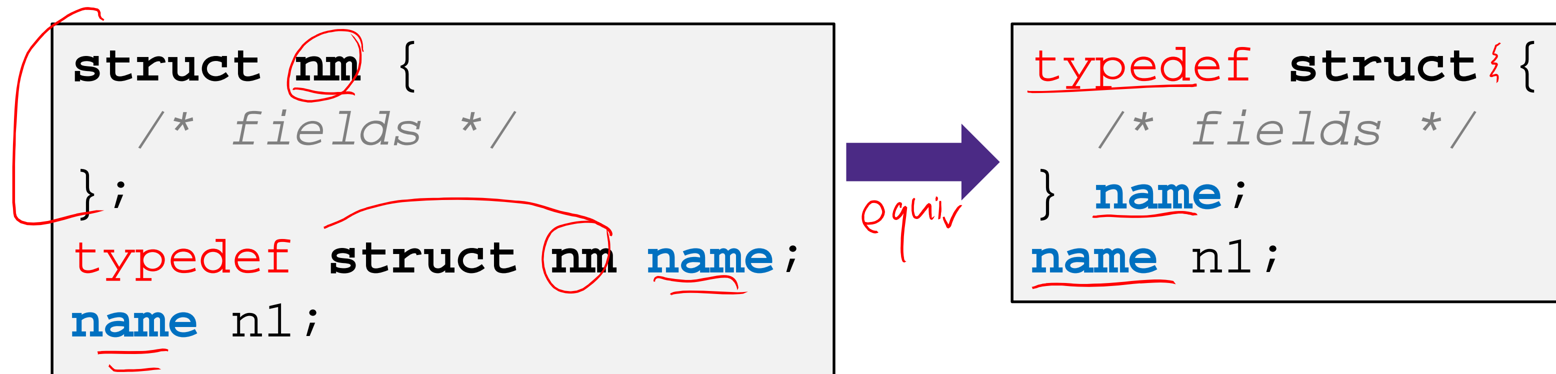
- Can also combine struct and instance definitions:
  - This syntax can be difficult to read, though

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

Typedef in C

- 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

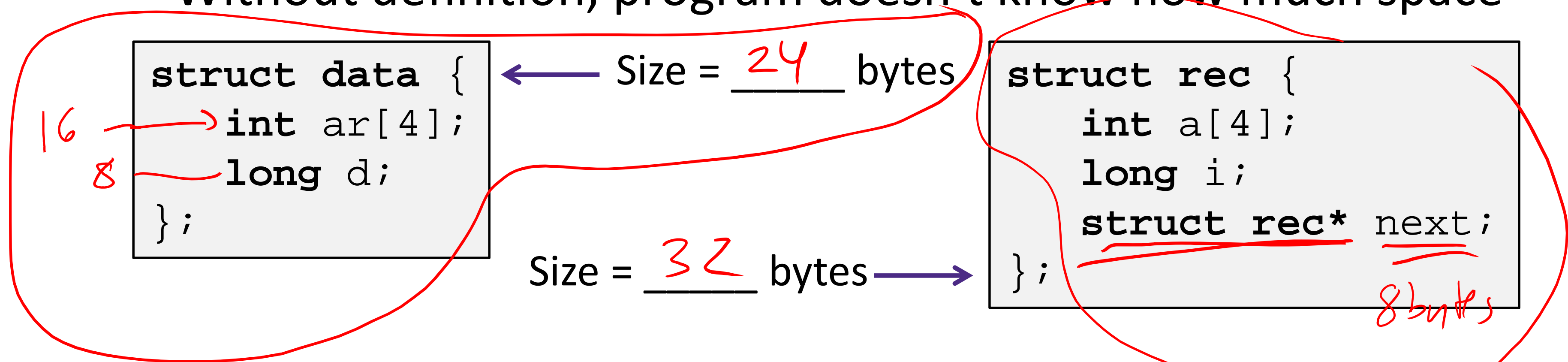


```
struct nm {
  /* fields */
};
typedef struct nm name;
name n1;
```

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

Scope of Struct Definition

- Why is the placement of struct definition important?
  - What actually happens when you declare a variable?
    - Creating 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

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

- 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;`

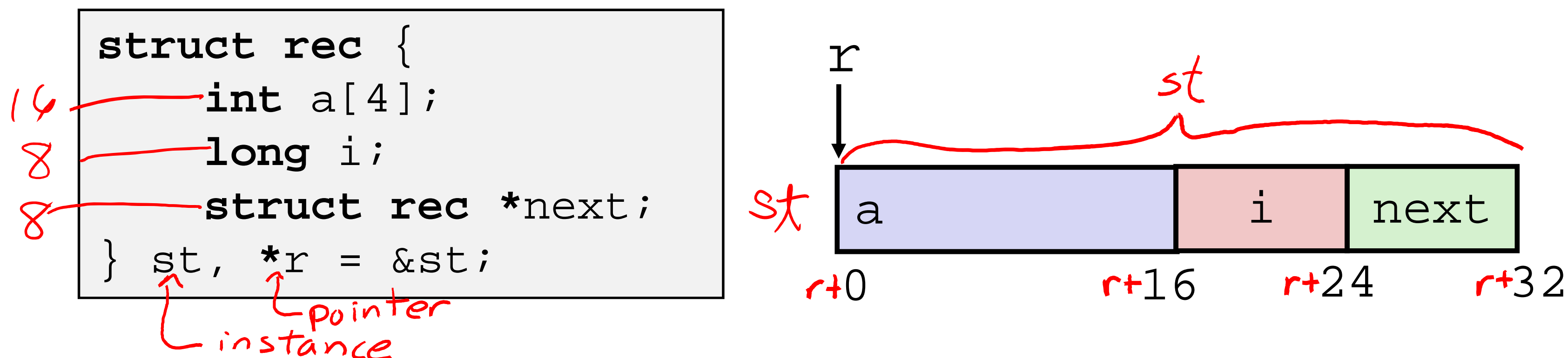
- **In assembly:** register holds address of the first byte
  - Access members with offsets

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)

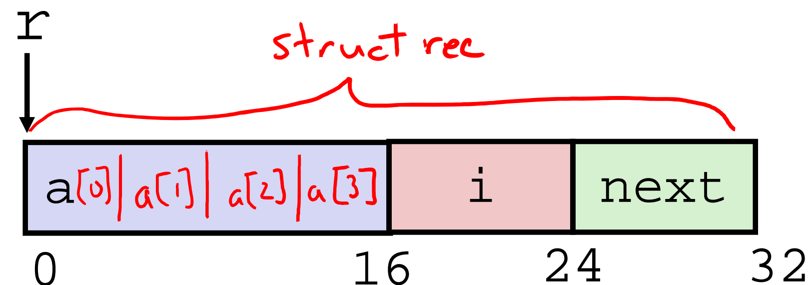
Structure Representation



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

Structure Representation

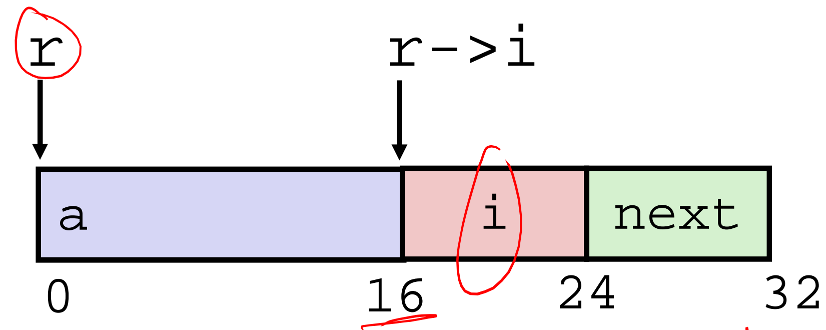
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} st, *r = &st;
```



- 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;
```



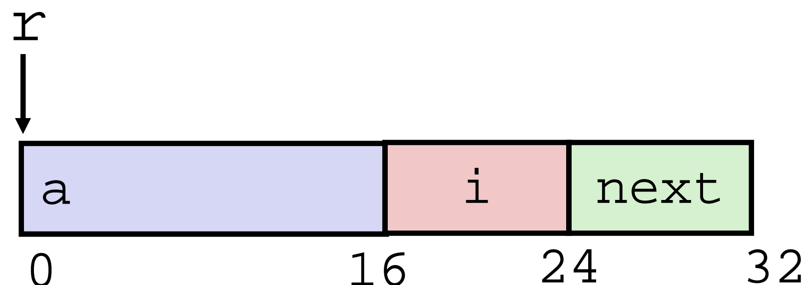

```
long get_i(struct rec *r)
{
  return r->i;
}
```

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

- Compiler knows the *offset* of each member within a struct
  - Compute as `*(r+offset)`
    - Referring to absolute offset, so no pointer arithmetic

Exercise: Pointer to Structure Member

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



pointer

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

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

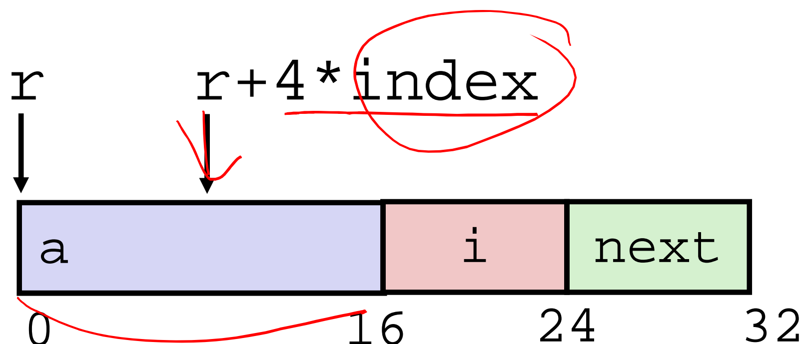
want address

```
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: $...0_2$
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: $...0000_2$

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 >> $\log_2(K)$.
No remainder means no weight is "lost" during the shift → all zeros in lowest $\log_2(K)$ bits.

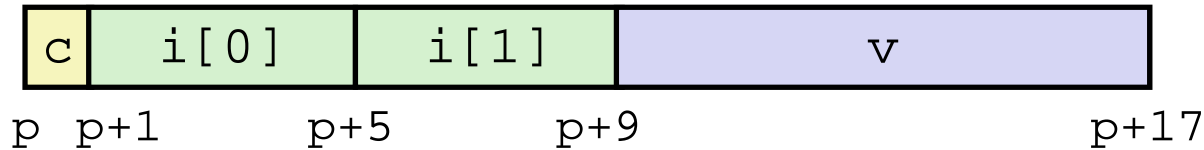
Alignment Principles

- 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

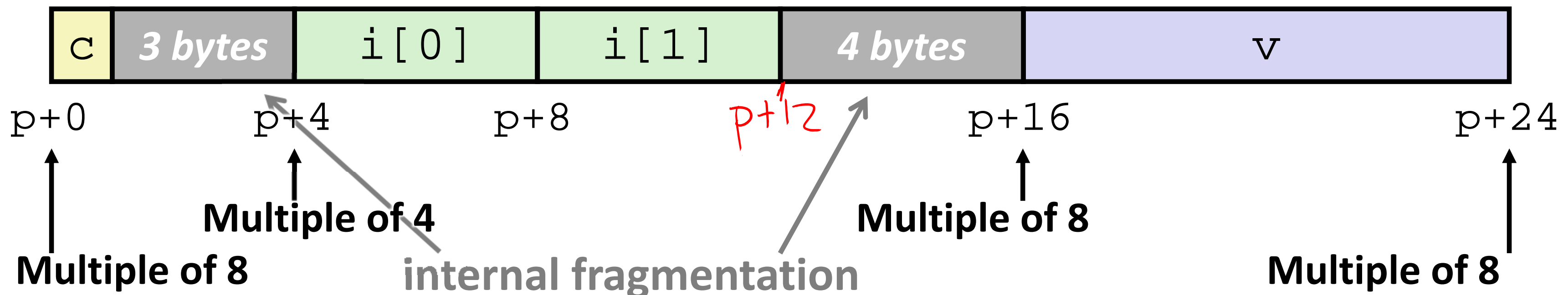
Structures & Alignment

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

- Unaligned Data



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



Satisfying Alignment with Structures (1)

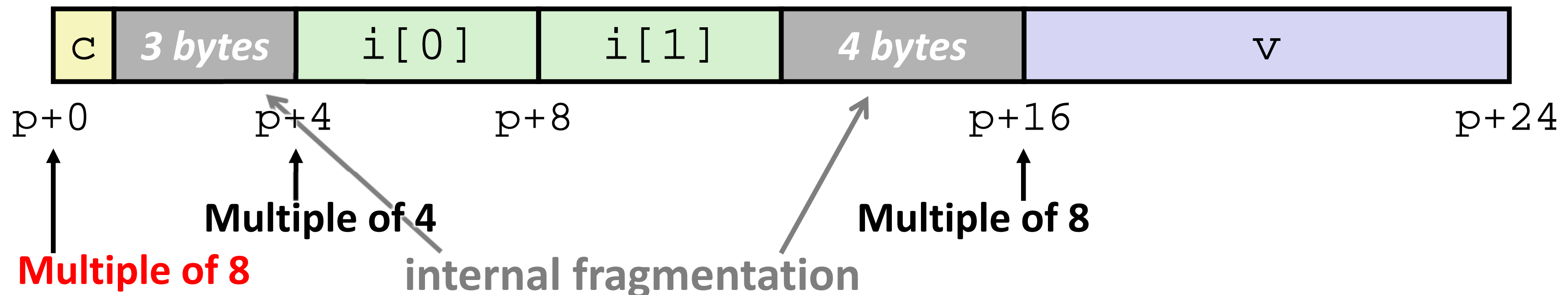
- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement K_{max}
    - K_{max} = Largest alignment of any element
    - Counts array elements individually as elements

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

K 1 4 8

Kmax = 8

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

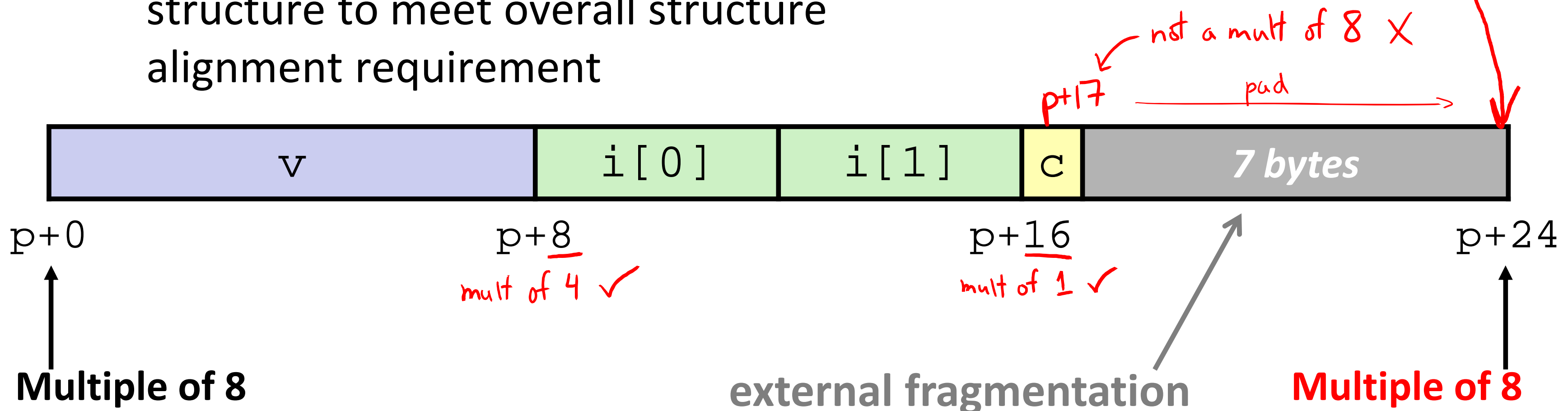


Satisfying Alignment with Structures (2)

- 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;
```

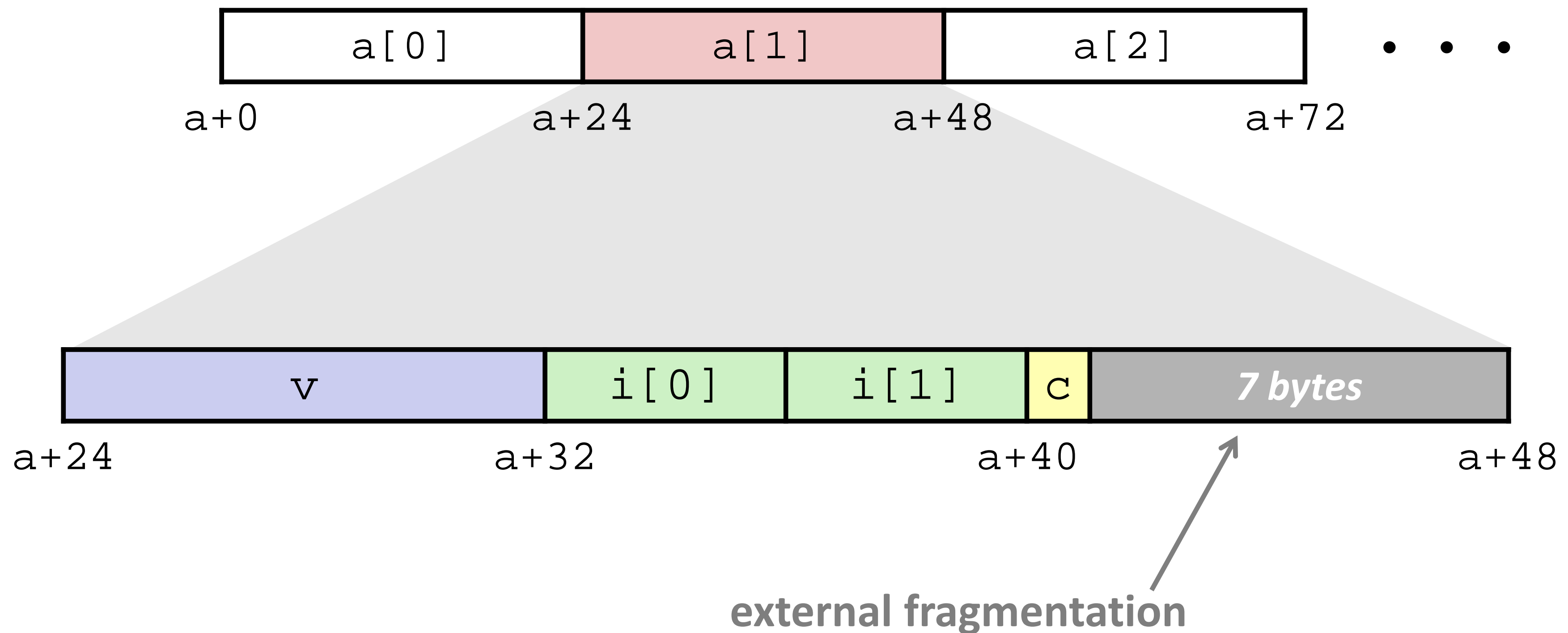
- For largest alignment requirement $K_{\max}$, overall structure size must be multiple of $K_{\max}$ = 8
  - Compiler will add padding at end of structure to meet overall structure alignment requirement



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];
```

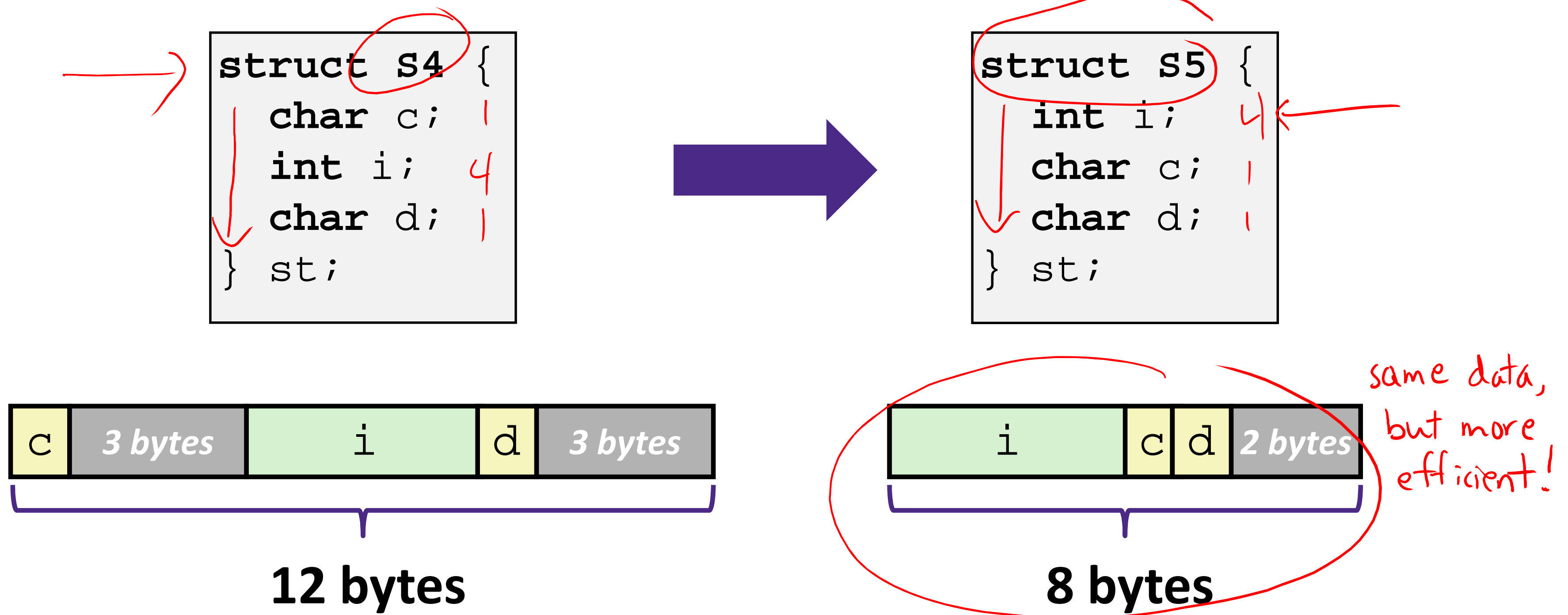


Alignment of Structs

- 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



Polling Question [Structs]

Vote on `sizeof(struct old)`:
http://pollev.com/rea

- 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;
};
```

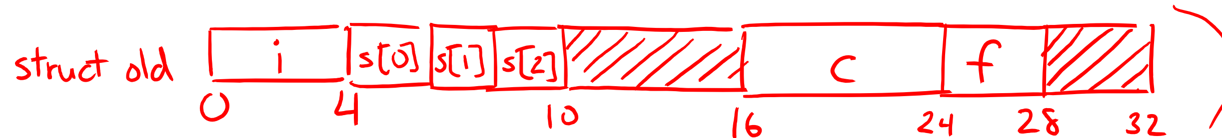
```
struct new {
  int      i;

  float    f;

  char *   c;

  short    s[3];
};
```

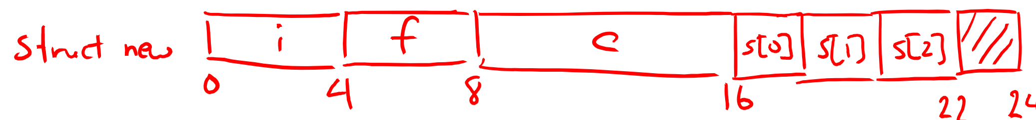
could also switch these (internal vs. external frag)

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

`sizeof(struct old)` = 32   `sizeof(struct new)` = 24

A. 16 bytes
B. 22 bytes
C. 28 bytes
D. 32 bytes
E. We're lost…

struct old


struct new


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