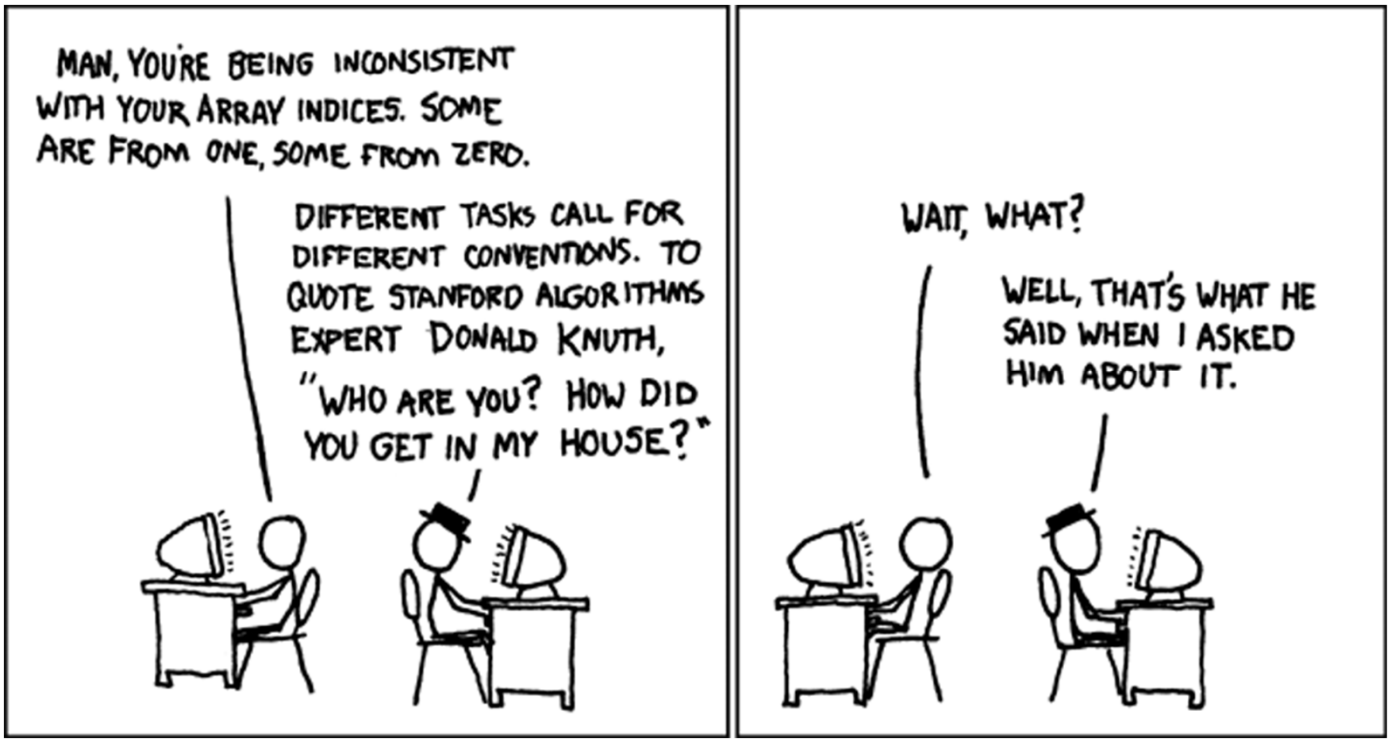
Structs & Alignment

CSE 351 Spring 2021

Instructor:
Ruth Anderson

Teaching Assistants:
Allen Aby
Joy Dang
Alena Dickmann
Catherine Guevara
Corinne Herzog
Ian Hsiao
Diya Joy
Jim Limprasert
Armin Magness
Aman Mohammed
Monty Nitschke
Allie Pfleger
Neil Ryan
Alex Saveau
Sanjana Sridhar
Amy Xu



http://xkcd.com/163/

Administrivia

- **Mid-quarter survey** due Saturday (5/01) on Canvas
- Lab 2 (x86-64) due Friday (4/30)
  - 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
- **Questions Docs**: Use @uw google account to access!!
  - https://tinyurl.com/CSE351-21sp-Questions

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();
```

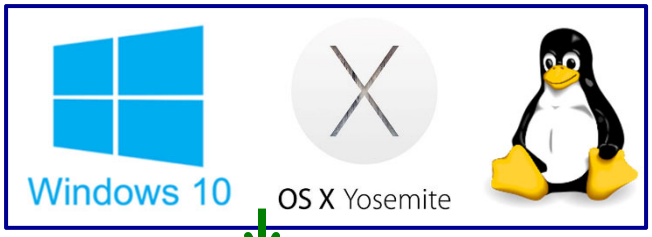
Assembly language:

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

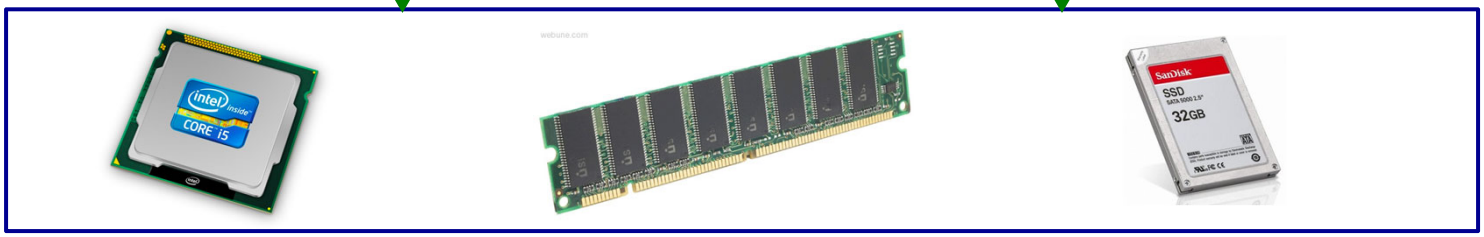
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100011111
```

OS:



Computer system:



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

Review Questions

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

tag; fields; two instances; $K_{max} = 8$

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

- Which of the following statements are syntactically valid?
  - for struct instances (inst), -> for struct pointers (ptr)
  - ✓ `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 Assembly

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

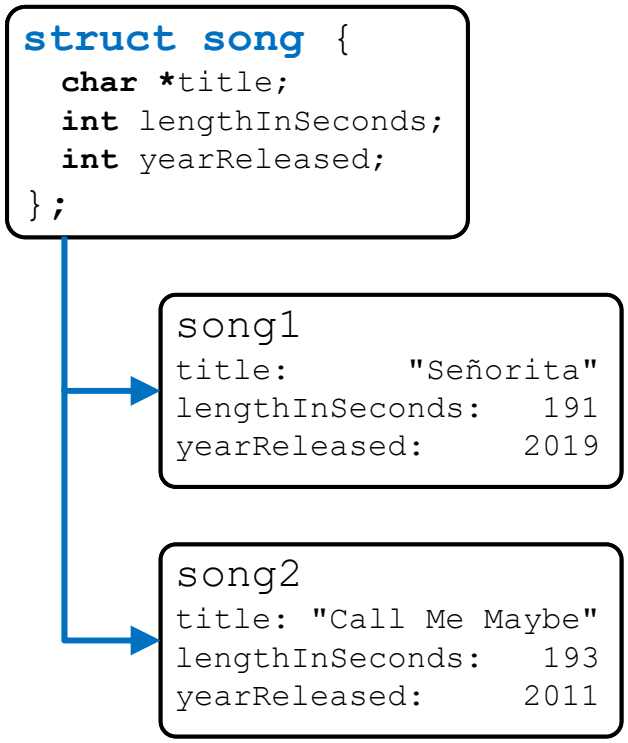
Structs in C

- 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`**"

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

- 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;`

uli 2;

data type

alias

- Joint struct definition and typedef
  - Don't need to give struct a name in this case

unnamed!

(1) define struct

(2) typedef

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

combined

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

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

```
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

Accessing Structure Members

- 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: `(*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`

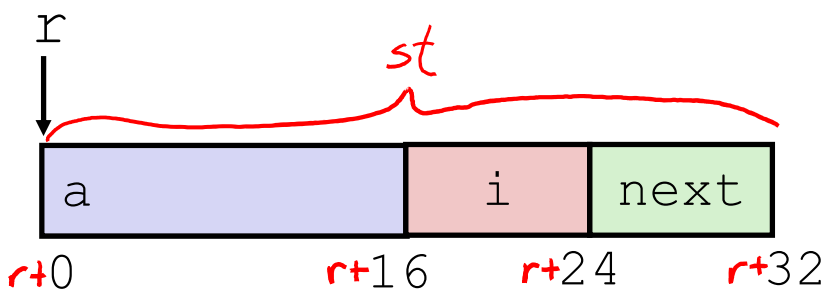
- In Java, almost everything is a pointer ("*reference*") to an object
  - Cannot declare variables or fields that are structs or arrays
  - Always a *pointer* to a struct or array
  - So every Java variable or field is ≤ 8 bytes (but can point to lots of data)

Structure Representation

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

pointer

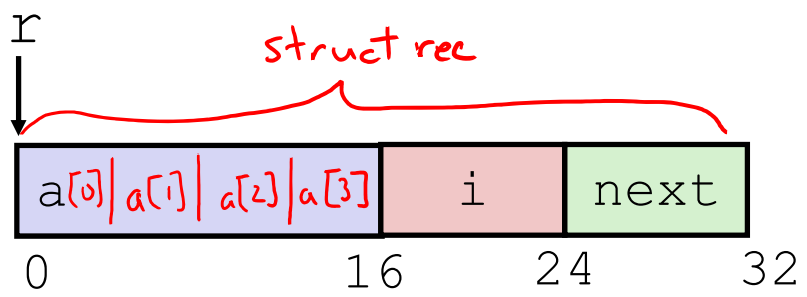
instance



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

Structure Representation

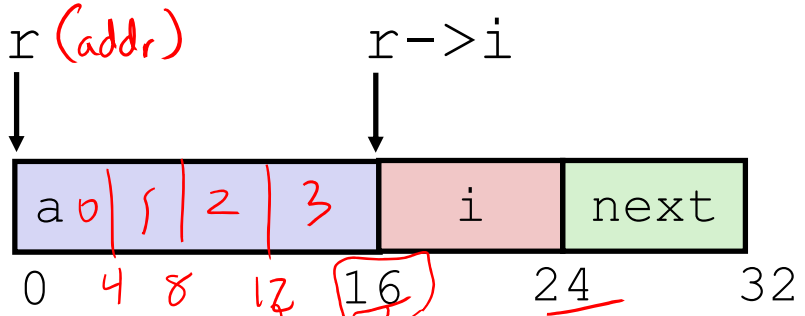
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} st, *r = &st;
```



- 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;
```



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

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

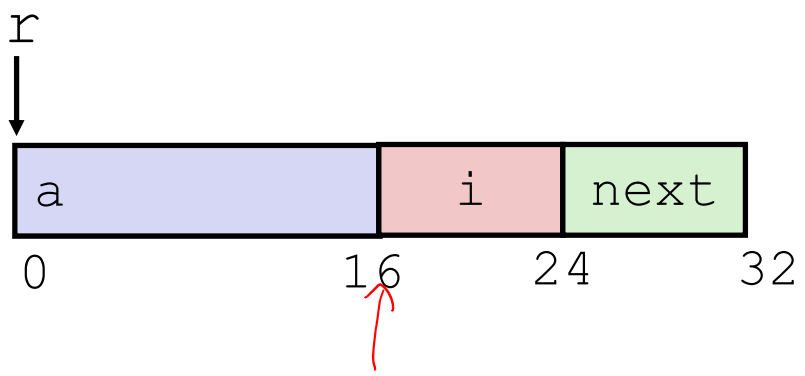
```
# r in %rdi
movq  16(%rdi), %rax
ret
```

```
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);
}
```

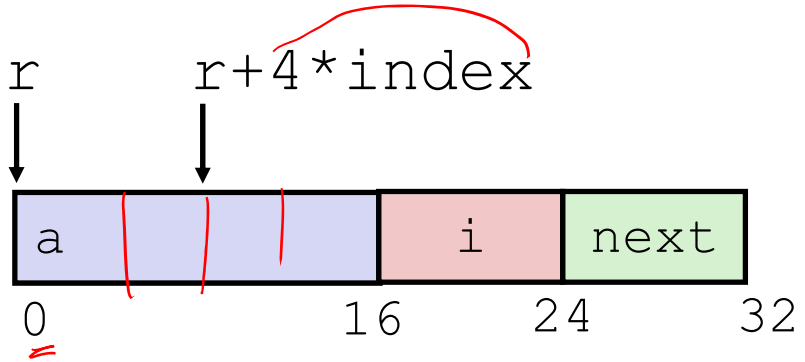
```
# r in %rdi
leaq  16(%rdi), %rax
ret
```

```
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];
}
```

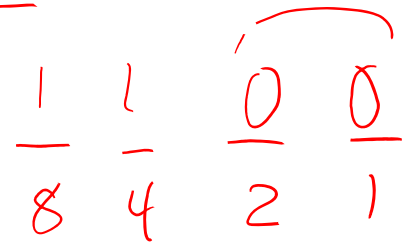
`&(r->a[index])`

```
  # r in %rdi, index in %rsi
  leaq 0(%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.

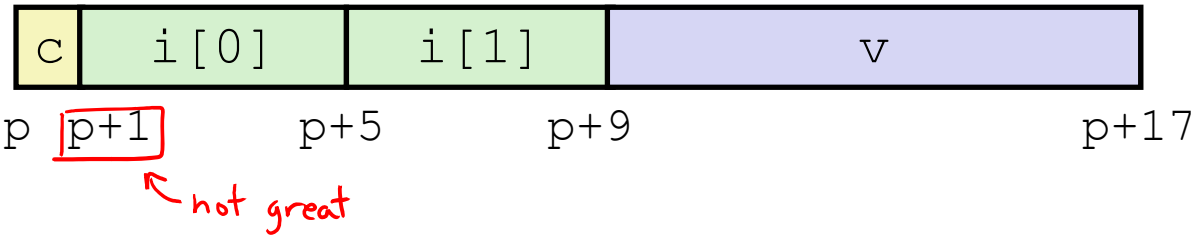
Alignment Principles

- 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

- Unaligned Data




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

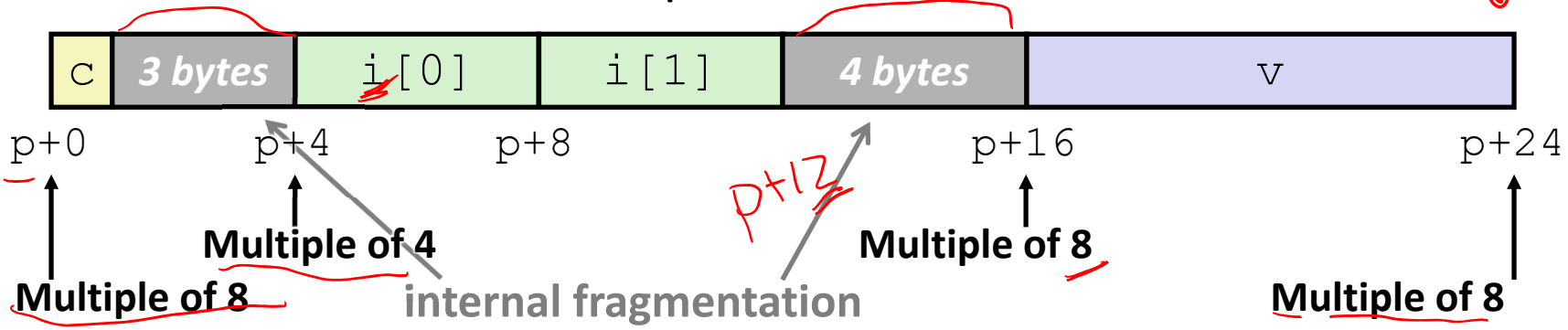
K.max = 8

24 B total

- Aligned Data

Compiler will do this

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

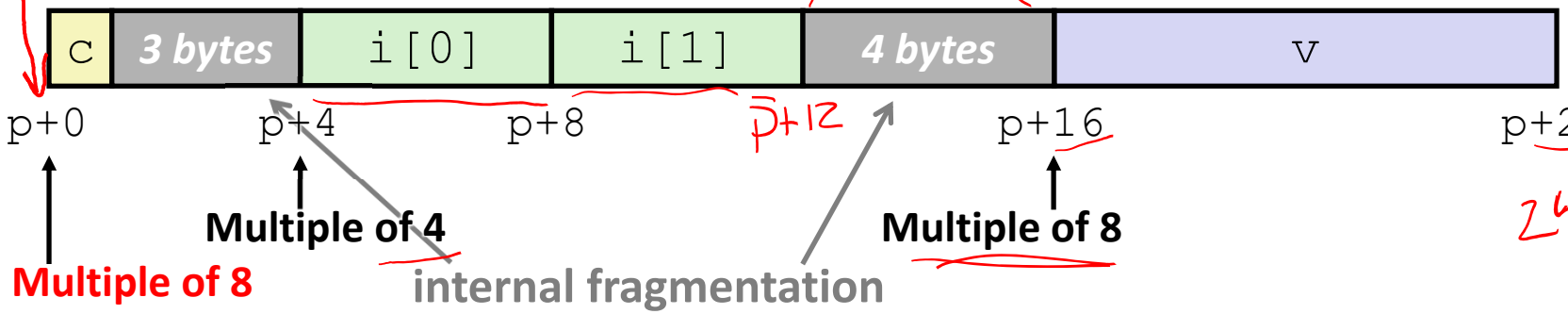


Satisfying Alignment with Structures (1)

- 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

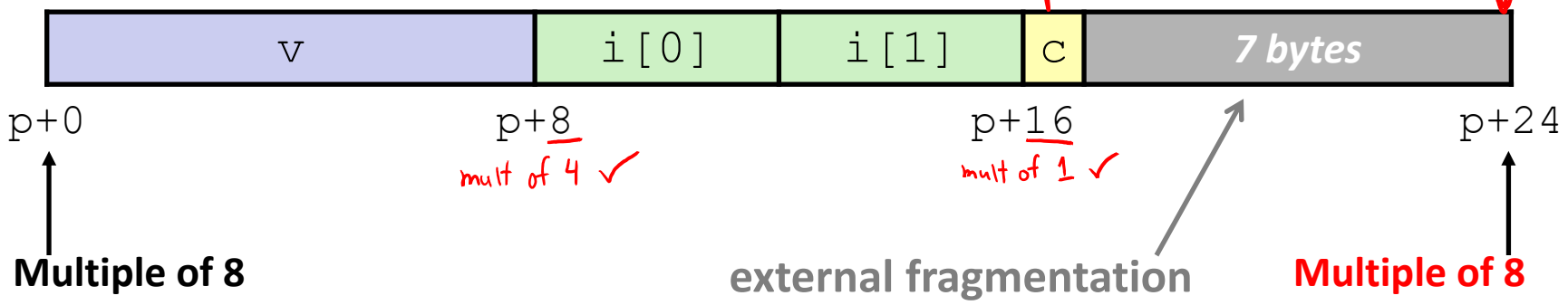


Satisfying Alignment with Structures (2)

- 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;
```

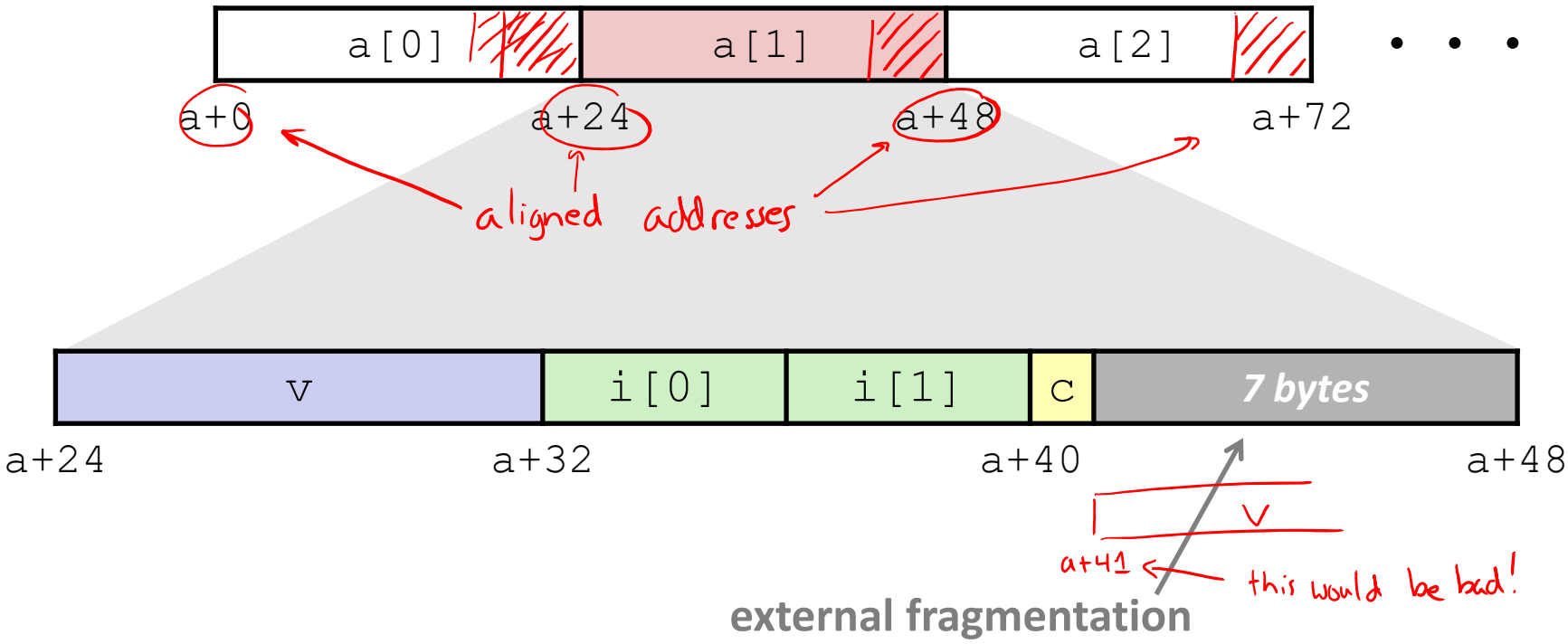
- For largest alignment requirement $K_{\max}$, overall structure size must be multiple of $K_{\max} = 8$
  - Compiler will add padding at end of structure to meet overall structure alignment requirement



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];
```



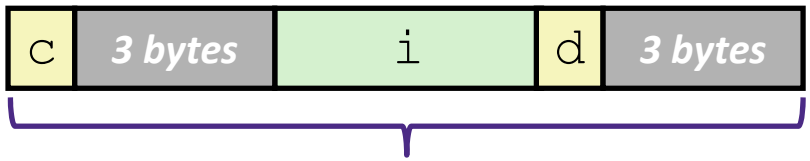
Alignment of Structs

- 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

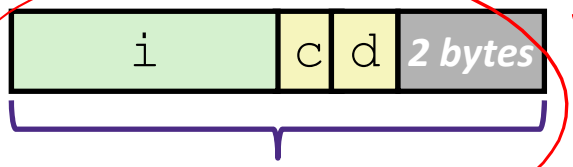
- Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first

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



12 bytes

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



8 bytes

same data, but more efficient!

Practice Questions

- 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];
};
```

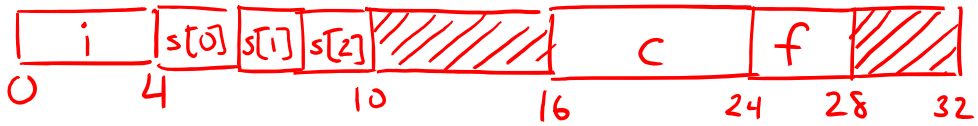
could also switch these (internal vs. external frag)

- 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