Structs & Alignment

CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson

Dad, can i have classes?

Dad: No, we have classes at home.

Classes at home: struct

Playlist: CSE 351 24Sp Lecture Tunes!

Relevant Course Information

- HW 12 due tonight; Lab 2 due Friday!
  - Lab 3 released at the same time; due 08 May
- HW13/14 due 01 May

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;
```

8 B

Also 8B

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

16 B, no need for padding!

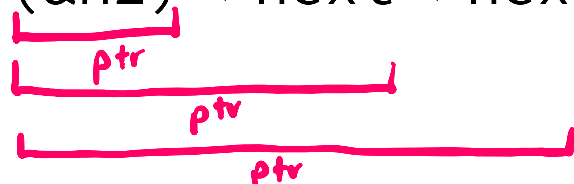
- Which of the following independent statements are syntactically valid?

✓ A. `n1.next = &n2;`

✗ B. `n2->data = 351;` // can't use "->" on a struct!

✓ C. `n1.next->data = 333;`

✗ D. `(&n2)->next->next.data = 451;`



Remember:

$p \rightarrow y \Leftrightarrow (*p).y$

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
  - Kind of like Java object, but no methods nor inheritance; just fields 😤
  - Way of defining compound data types

So primitive...

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

struct song song1;
song1.title = "Lavender Haze";
song1.lengthInSeconds = 182;
song1.yearReleased = 2022;

struct song song2;
song2.title = "State of Grace";
song2.lengthInSeconds = 295;
song2.yearReleased = 2012;
```

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

```
song1
title: "Lavender Haze"
lengthInSeconds:       182
yearReleased:         2022
```

```
song2
title:    "State of Grace"
lengthInSeconds:       295
yearReleased:         2012
```

Struct Definitions (Review)

- Structure definition:
  - Does **not** declare a variable; lets compiler know we're defining it and will be using instances of it
  - Variable type is "**struct name**"; gotta say it all every time we declare! Or do we?...

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

Really easy to forget the semicolon!

I'm not saying you will, but... you will.

- Variable declarations like any other data type:

```
struct cat matilda;      ← instance
struct cat *pc;          ← pointer
struct cat kitty_ar[3];  ← array
```

- Can also combine struct and instance definitions:

```
struct cat {
  /* fields */
} c, *pc = &c;
```

Instances of cat

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

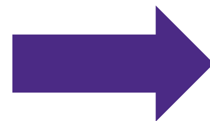
Typedef in C (Review)

- 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
  - *e.g.*,

```
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 cat {
  /* fields */
};
typedef struct cat kitty;
kitty olivia;
```



```
typedef struct {
  /* fields */
} kitty;

kitty olivia;
```

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 to set aside!

```
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
  - Top of singular C files, or if using a header file, place there!

Accessing Structure Members (Review)

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

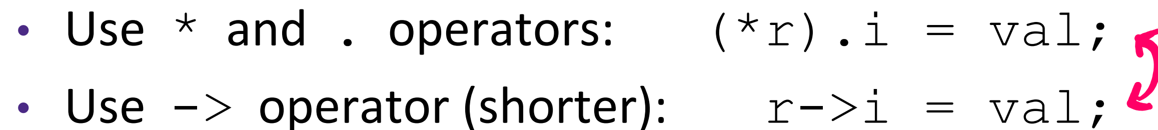
- Given a <u>struct instance</u>, access member using the . operator:

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

- Given a <u>pointer</u> to a struct:

```
struct rec* r; // r is a pointer, remember!
r = &r1;   // or malloc space for r to point to
```

We have **two equivalent 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

Java side-note

- 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`
  - Structs are really as close you can get to "objects" in Java

Ex: arr.length

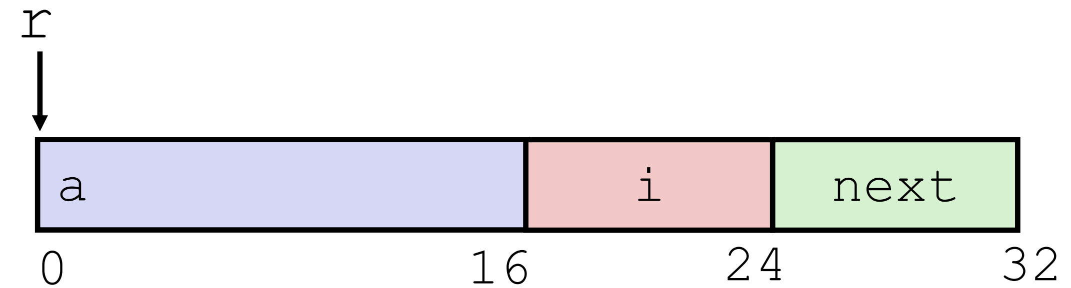
From CSE 142/121! :)

- 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)

```
class Record { ... }
Record x = new Record();
```

Structure Representation (Review)

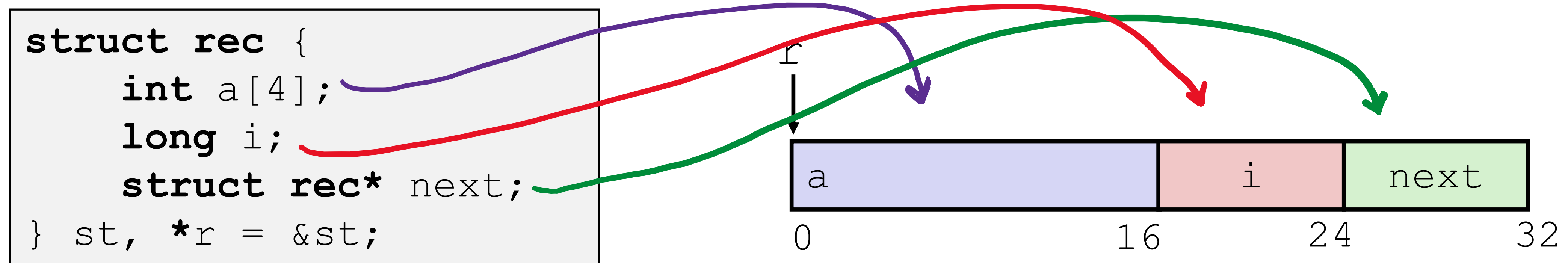
```
struct rec {
    int a[4];
    long i;
    struct rec* next;
} st, *r = &st;
```



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

Ex: r->i , st.i

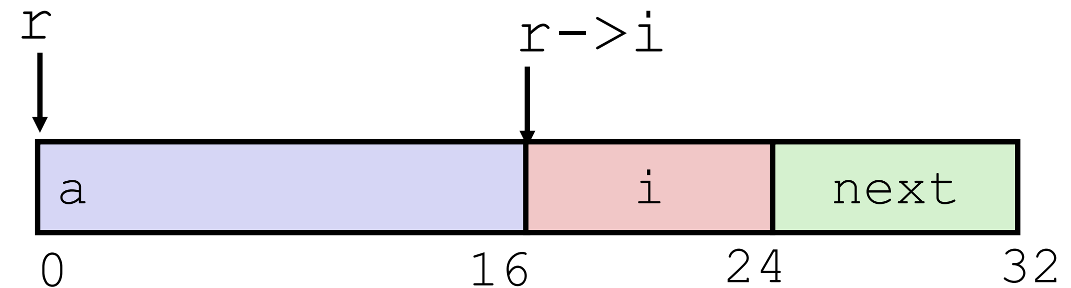
Structure Representation (Review)



- 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**
  - Good reason: debugging is easier, since in assembly, only get addr of first byte
- 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_of_member)`

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

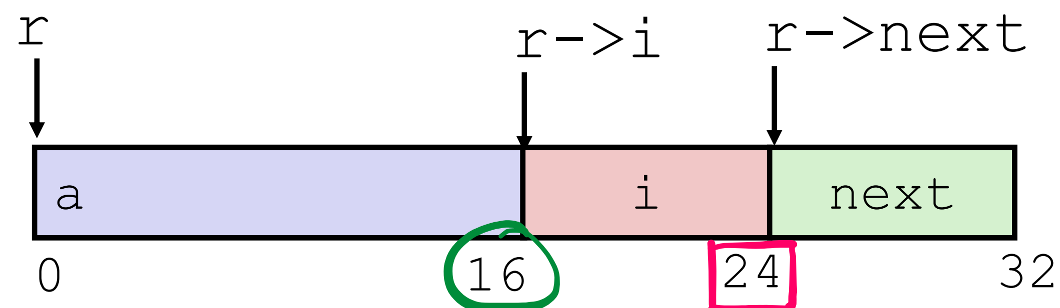
```
  # pointer r in %rdi
  movq  16(%rdi), %rax
  ret
```

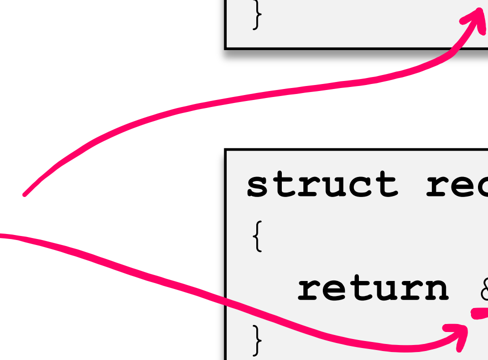
```
long get_a3(struct rec* r) {
  return r->a[3];
}
```

```
  # pointer 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);
}
```

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

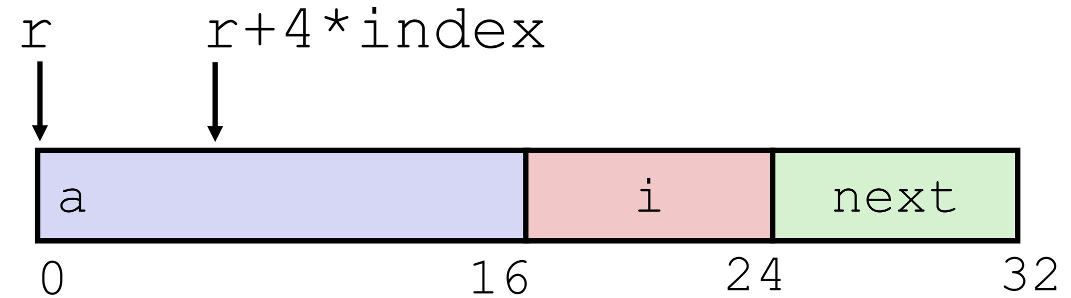
```
struct rec** addr_of_next(struct rec* r)
{
  return &(r->next);
}
```

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

We can also get addresses of members themselves!

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])`

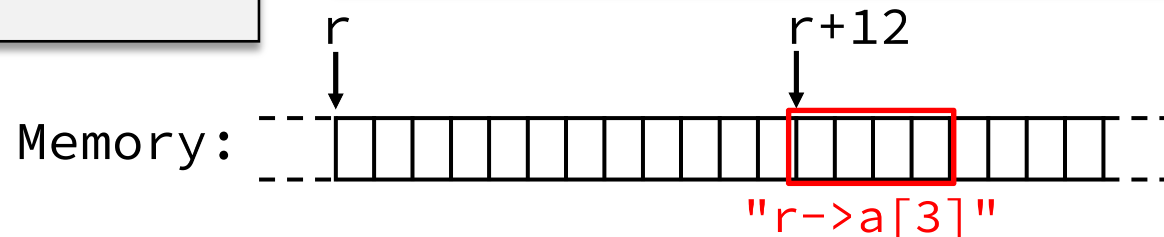
```
# pointerr in %rdi, index in %rsi
leaq  (%rdi,%rsi,4), %rax
ret
```

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, therefore
  - Address must be multiple of K
  - Required on some machines; advised on x86-64

short, K=2
int, K=4
long, K=8, etc.

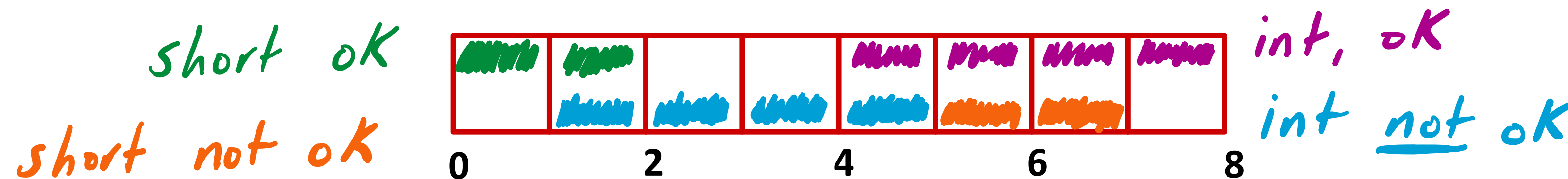
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of bytes (width is system dependent)
    - Important for caching and paging, virtual memory ← Sooooon...
    - Inefficient to load or store value that spans quad word boundaries
    - Virtual memory trickier when value spans 2 pages (more on this later) Oh god, would be so bad!
  - Though x86-64 hardware will work regardless of alignment of data

Memory Alignment in x86-64

Remember withinBlock from Lab1a? Yeah, you were essentially checking that the 6 LSBs were the same 😅

- ***Aligned*** means that any primitive object of K bytes must have an address that is a multiple of K
- Aligned addresses for data types:

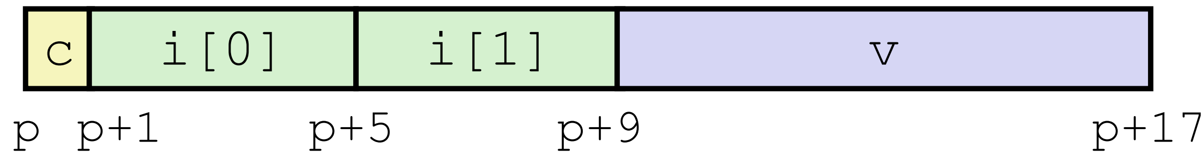
K	Type	Addresses
1	`char`	No restrictions
2	`short`	Lowest bit must be zero: ...0_2 Ex: 10, 100, 110, ...
4	`int, float`	Lowest 2 bits zero: ...00_2 Ex: 100, 1100, 1000, 10100...
8	`long, double, *`	Lowest 3 bits zero: ...000_2 Ex: 1000, 11000, 10000, ...
16	`long double`	Lowest 4 bits zero: ...0000_2 Ex: 10000, 110000, 100000...



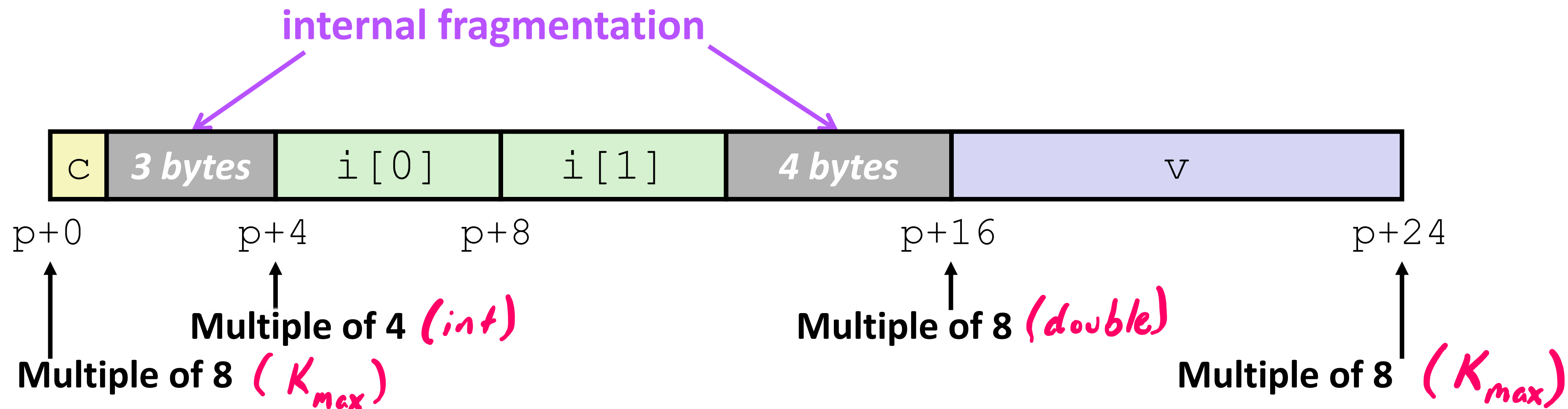
Structures & Alignment (Review)

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

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



- **Aligned Data**: unused space, but benefits later on.
  - Primitive data type requires K bytes
  - Address must be multiple of K

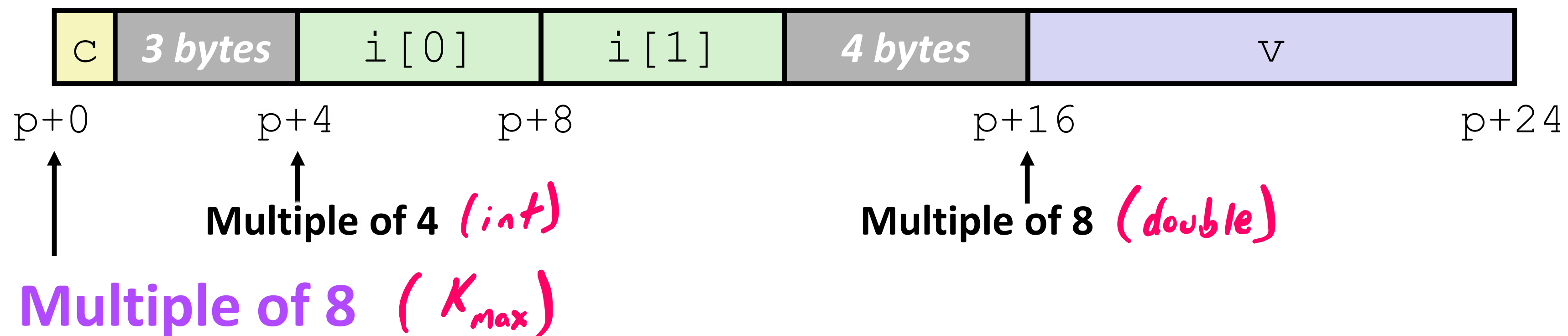


Satisfying Alignment with Structures

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

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

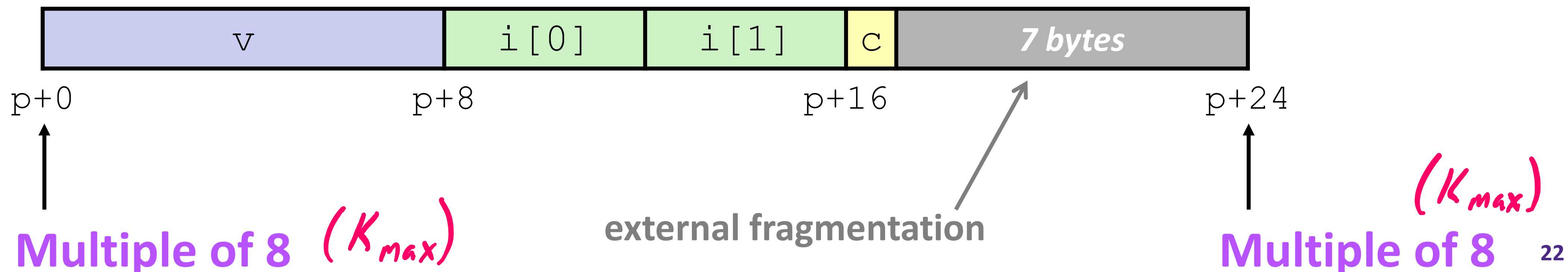
Here: K_{max} = 8



Okay, let's try to do that…

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

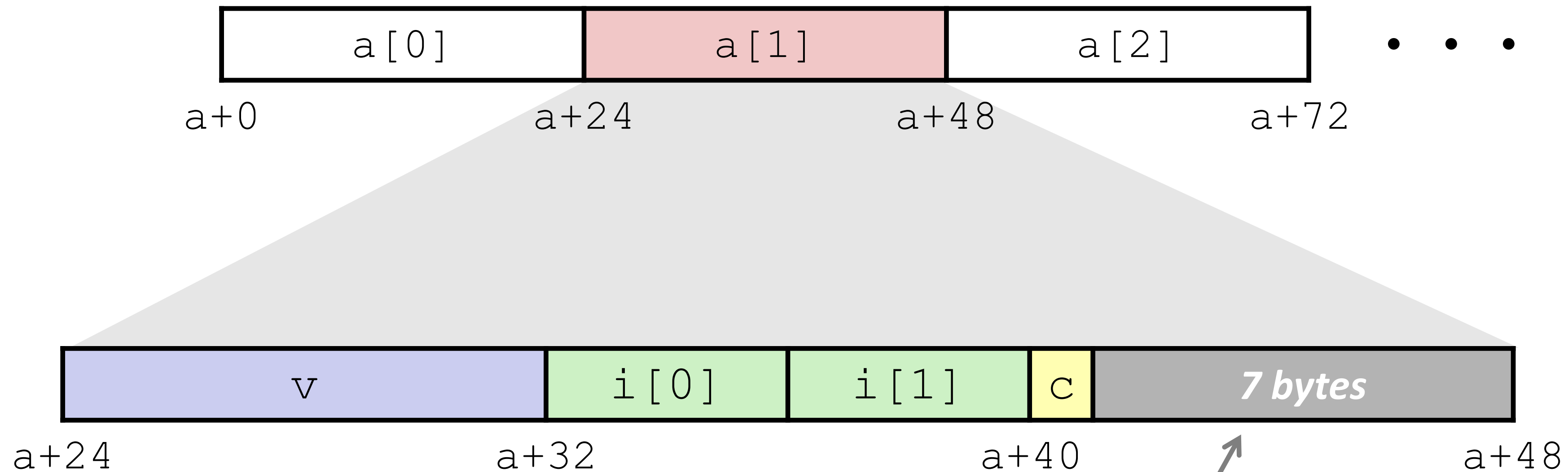
- Can find offset of individual fields using `offsetof()`
  - Need to `#include <stddef.h>`
  - Example: `offsetof(struct S2,c)` returns 16
- For largest alignment requirement K_{max}, overall structure size must be multiple of K_{max}
  - Compiler will add padding at end of structure to meet overall structure alignment requirement



A Benefit: 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;
} st, *p = &st;
```

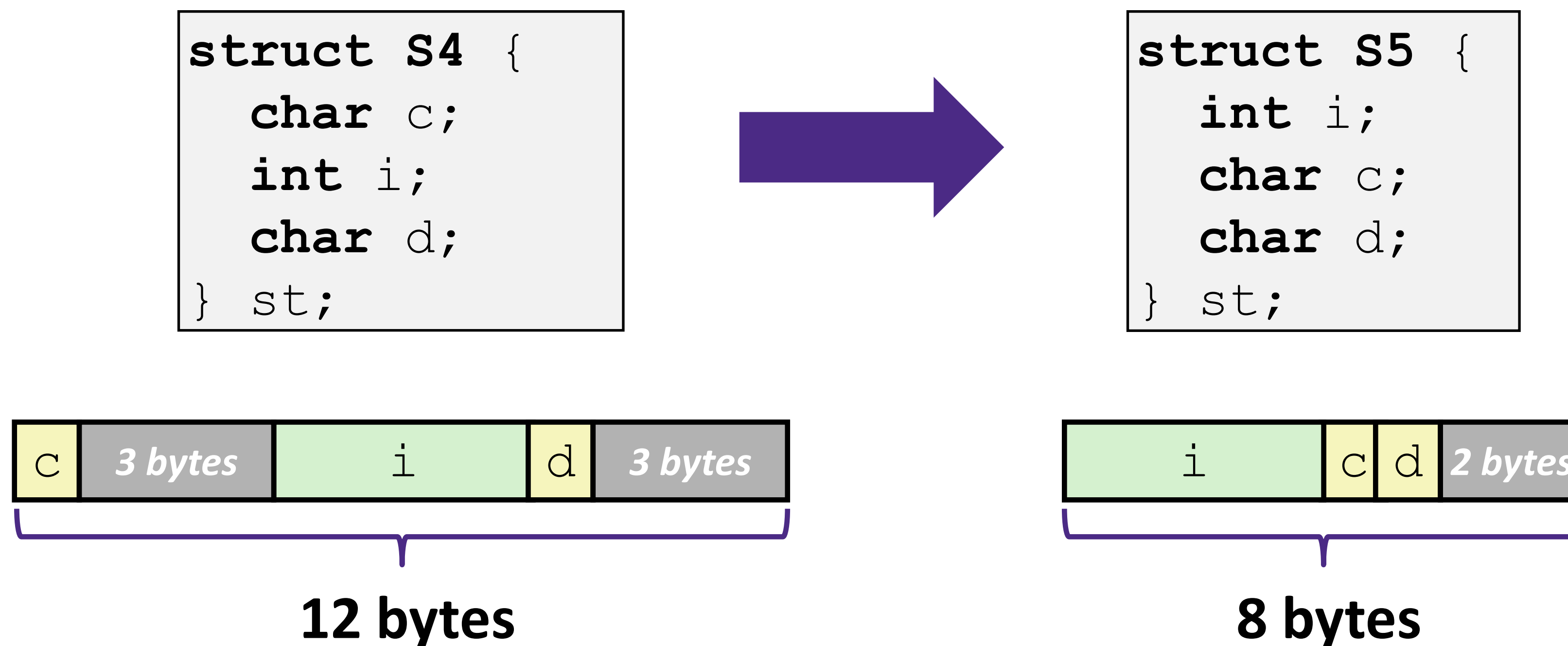


Alignment of Structs (Review)

- Compiler will do the following:
  - Still 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

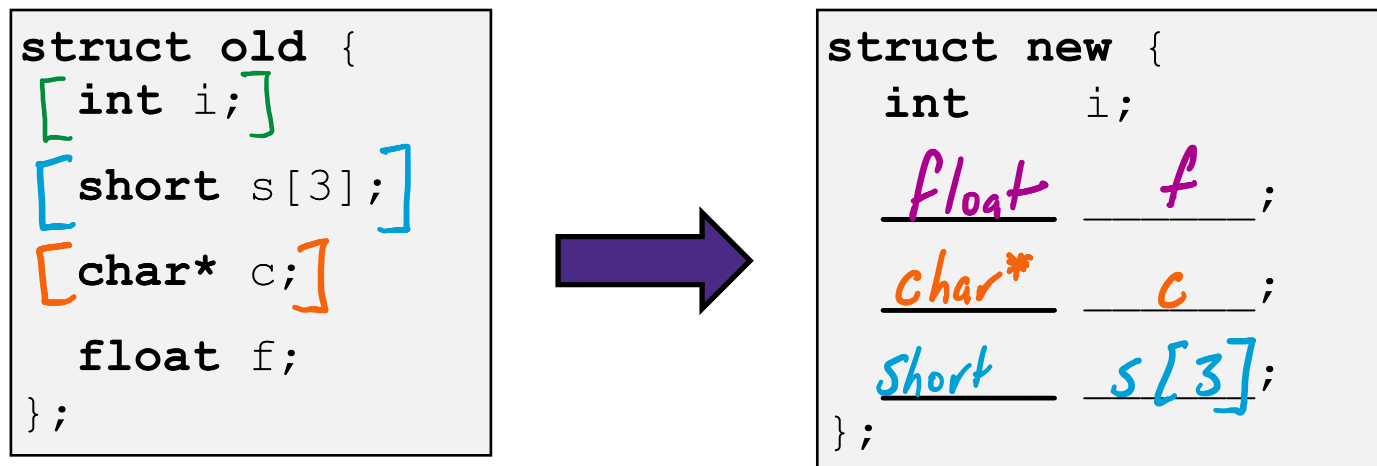
How the Programmer Can Save Space

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



Practice Question

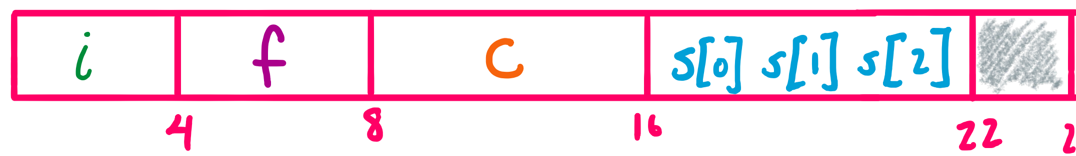
- Minimize the size of the struct by re-ordering the vars:



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

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

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