Structs & Alignment

CSE 351 Summer 2020

Instructor:

Porter Jones

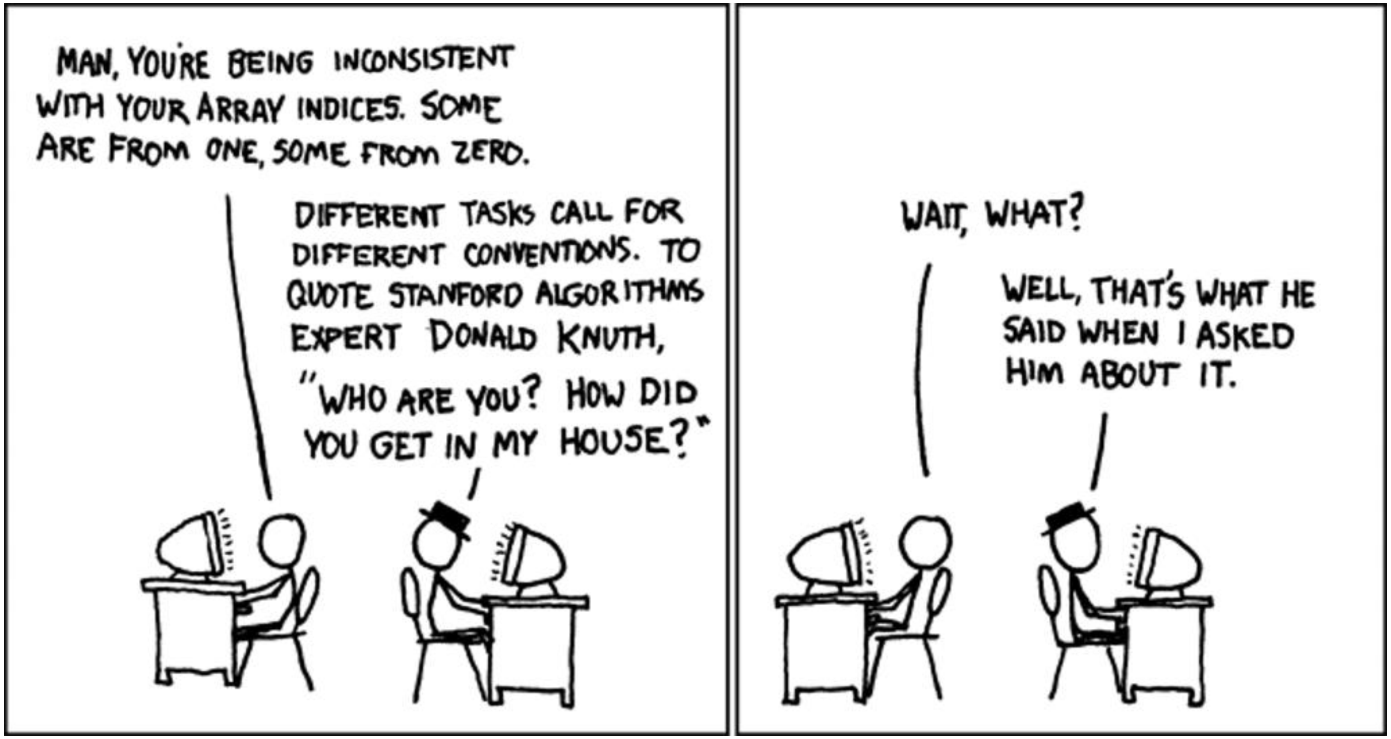
Teaching Assistants:

Amy Xu

Callum Walker

Sam Wolfson

Tim Mandzyuk


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

http://xkcd.com/163/

Administrivia

- Questions doc: https://tinyurl.com/CSE351-7-24

- hw13 due Monday (7/27) – 10:30am
- hw14 due Wednesday (7/29) – 10:30am
  - This one is especially long, please start early

- Lab 3 due next Friday (7/31) – 11:59pm
  - You get to write some buffer overflow exploits!

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();
```

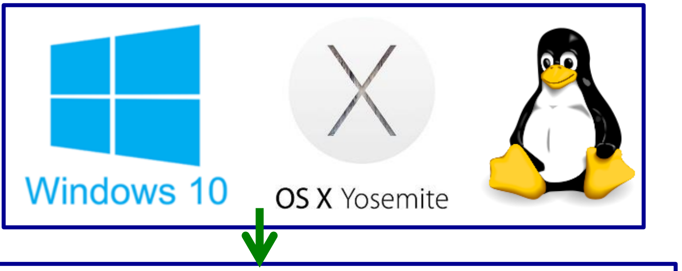
Assembly language:

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

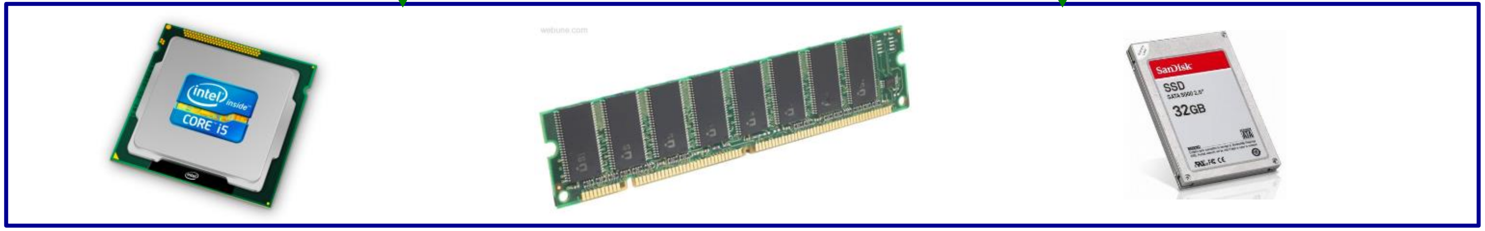
Machine code:

```
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~~

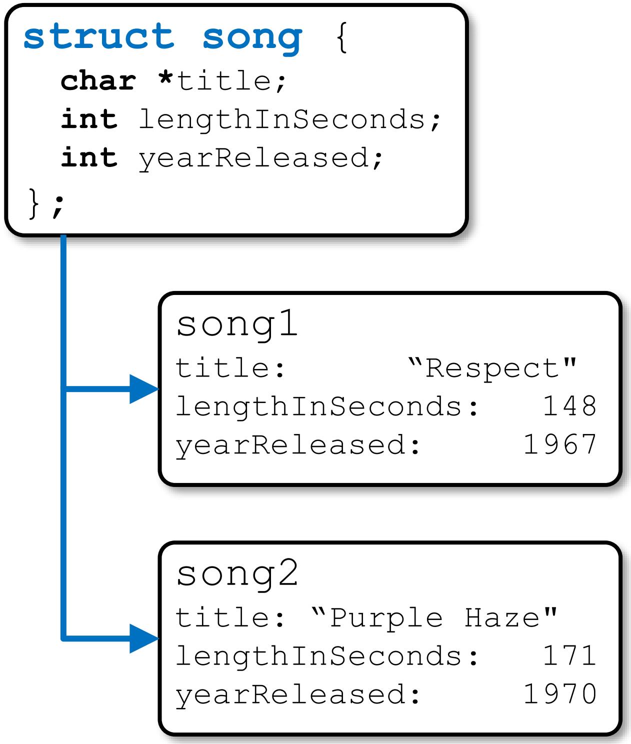
Structs in C

- 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 = "Respect";
song1.lengthInSeconds = 148;
song1.yearReleased = 1967;

struct song song2;
song2.title = "Purple Haze";
song2.lengthInSeconds = 171;
song2.yearReleased = 1970;
```



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

```
song1
title:          "Respect"
lengthInSeconds:      148
yearReleased:        1967
```

```
song2
title:  "Purple Haze"
lengthInSeconds:      171
yearReleased:        1970
```

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

instance

pointer

array

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

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

Size = _____ bytes

Size = _____ 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 for short: `r->i = val;`

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

Java connection

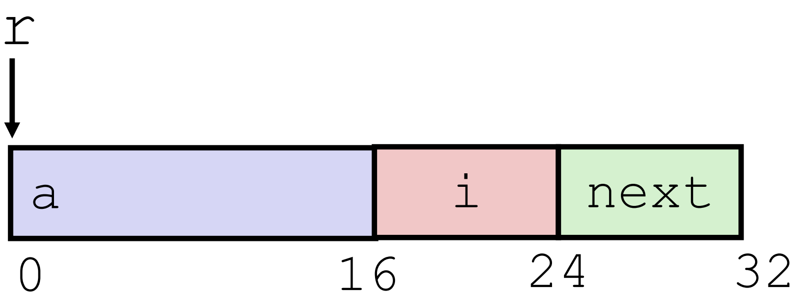
```
class Record { ... }
Record x = new Record();
```

- 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

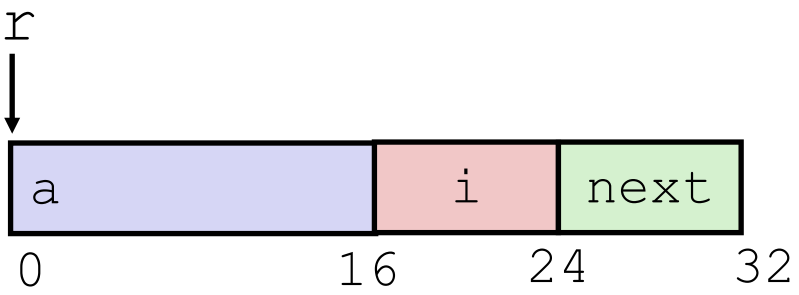
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
struct rec st;
struct rec *r = &st;
```



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

Structure Representation

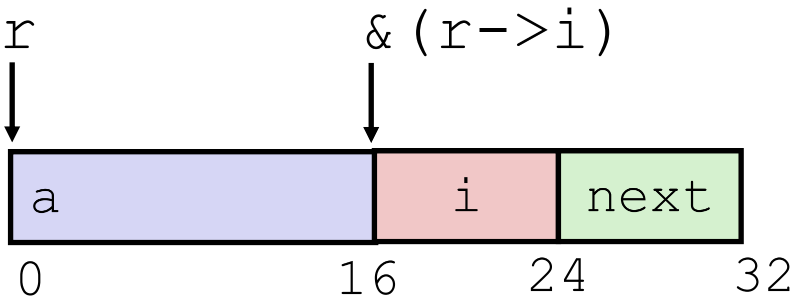
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
struct rec st;
struct rec *r = &st;
```



- 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;
};
struct rec st;
struct rec *r = &st;
```



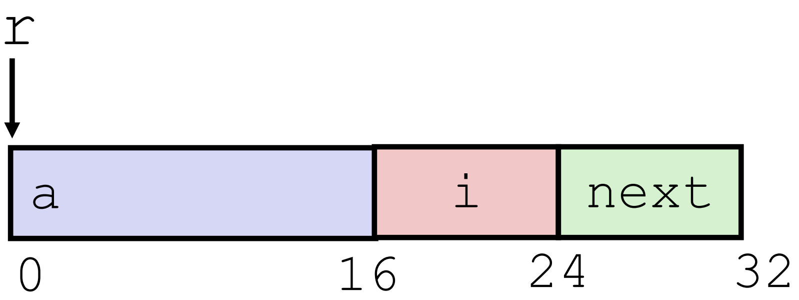
- Compiler knows the *offset* of each member within a struct
  - Compute as `*(r+offset)`
    - Referring to absolute offset, so no pointer arithmetic

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

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

Exercise: Pointer to Structure Member

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



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

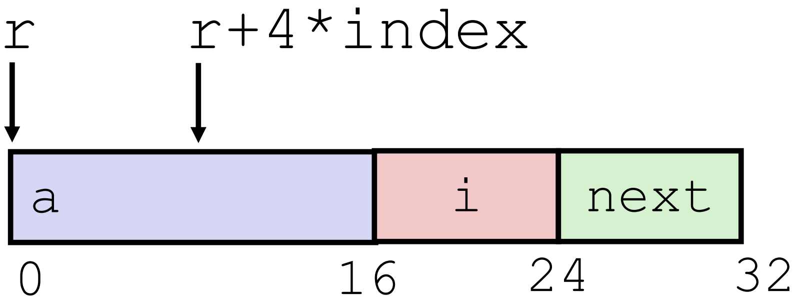
```
  # r in %rdi
  ______   ___________,%rax
  ret
```

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

```
  # r in %rdi
  ______   ___________,%rax
  ret
```

Generating Pointer to Array Element

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
struct rec st;
struct rec *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: $\ldots0_2$
4	`int, float`	Lowest 2 bits zero: $\ldots00_2$
8	`long, double, *`	Lowest 3 bits zero: $\ldots000_2$
16	`long double`	Lowest 4 bits zero: $\ldots0000_2$

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

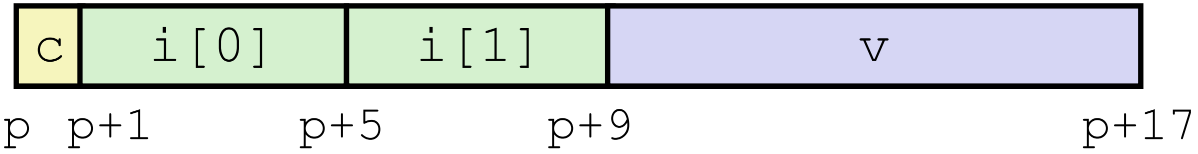
- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on x86-64

- 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

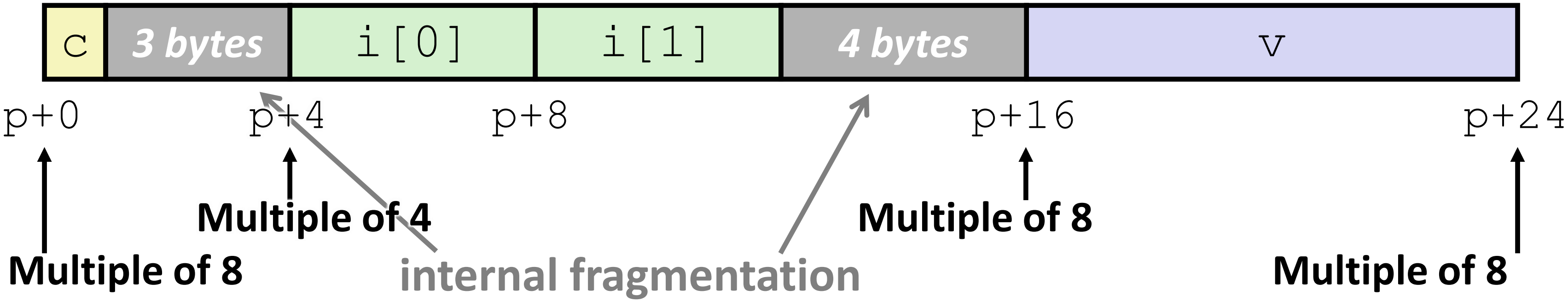
Structures & Alignment

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

- Unaligned Data



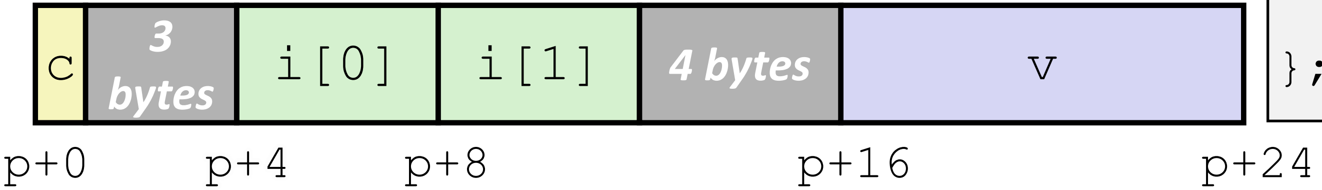
- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K



Structures & Alignment: Fragmentation

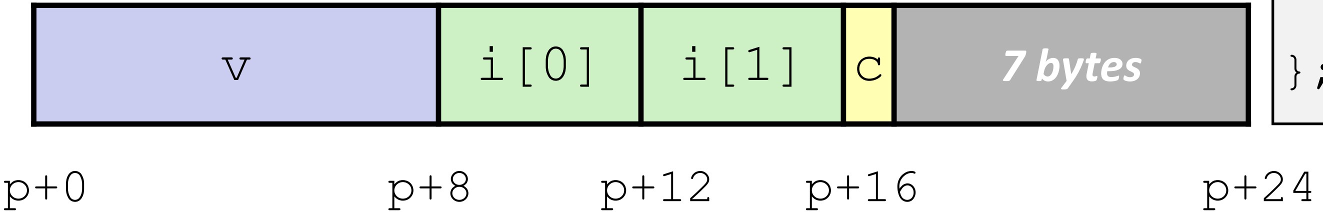
- Fragmentation occurs when there are unused portions of a struct
- Internal Fragmentation
  - Unused portion(s) occur *between* fields

```
struct S1 {
   char c;
   int i[2];
   double v;
};
```



- External Fragmentation
  - Unused portion at the end of the struct

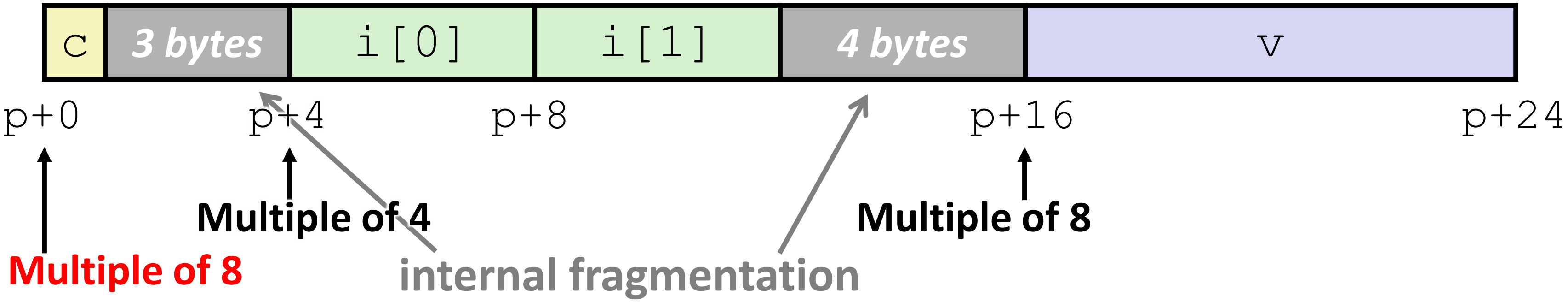
```
struct S2 {
   double v;
   int i[2];
   char c;
};
```



Satisfying Alignment with Structures (1)

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

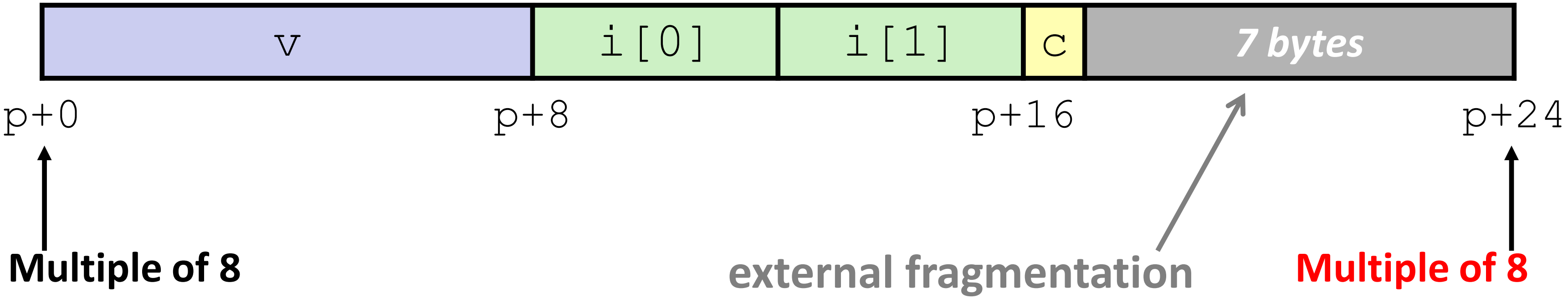
- Within structure:
  - Must satisfy each element's alignment requirement
- Overall structure placement
  - Each structure has alignment requirement $K_{\max}$
    - $K_{\max}$ = Largest alignment of any element
    - Counts array elements individually as elements
- Example:
  - $K_{\max}$ = 8, due to `double` element



Satisfying Alignment with Structures (2)

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

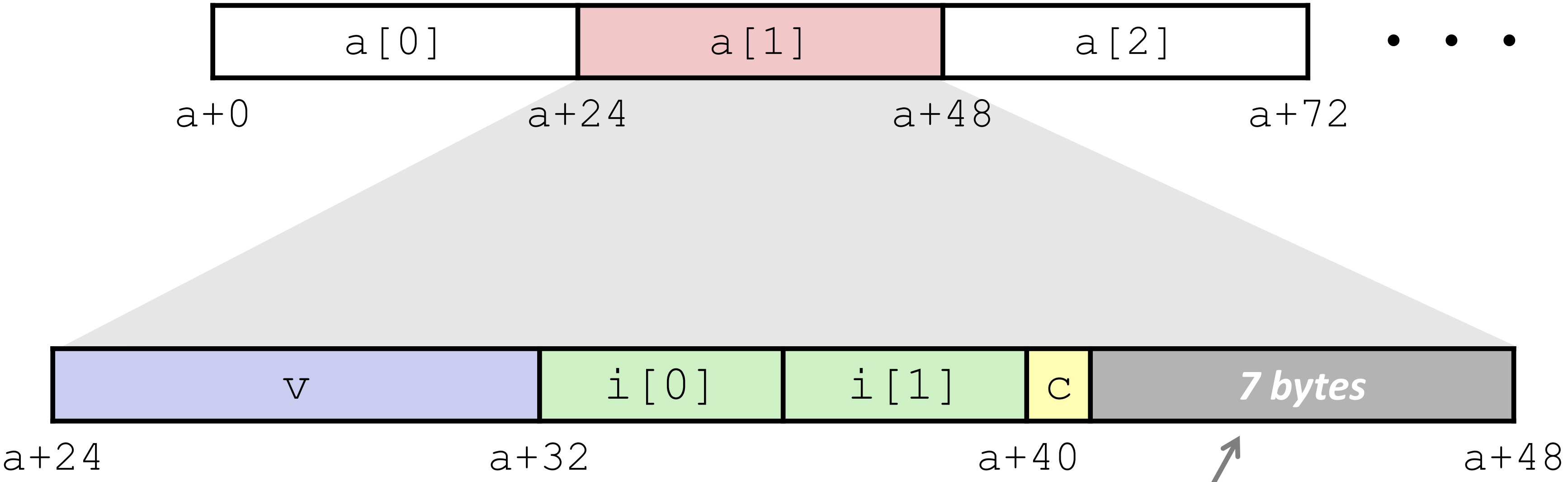
- Can find offset of individual fields using `offsetof()`
  - Need to `#include <stddef.h>`
  - *e.g.* `offsetof(struct S2,c)` returns 16
- For largest alignment requirement K_{max}, overall structure size must be multiple of K_{max}
  - Compiler will add padding at end of structure to meet overall structure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



Arrays of Structures

- Overall structure size multiple of K_{max}
- Satisfy alignment requirement for every element in array

```
struct S2 {
  double v;
  int i[2];
  char c;
};
struct S2 a[10];
```



external fragmentation

Alignment of Structs

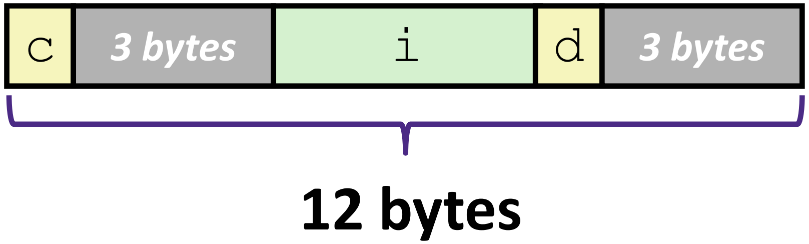
- Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each ***field*** must be aligned *within* the struct *(may insert padding)*
    - `offsetof` can be used to get actual field offset
  - Overall struct must be ***aligned*** according to largest field
  - Total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

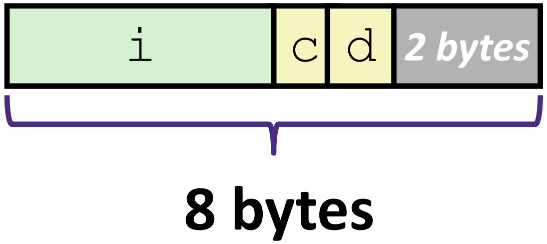
```
struct S4 {
  char c;
  int i;
  char d;
};
struct S4 st;
```

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



c | 3 bytes | i | d | 3 bytes

12 bytes



i | c | d | 2 bytes

8 bytes

Polling Question [Structs]

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

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

```
struct old {
  int i;

  short s[3];

  char *c;

  float f;
};
```

```
struct new {
  int     i;

  ______ ______;

  ______ ______;

  ______ ______;
};
```

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

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

A. 16 bytes

B. 22 bytes

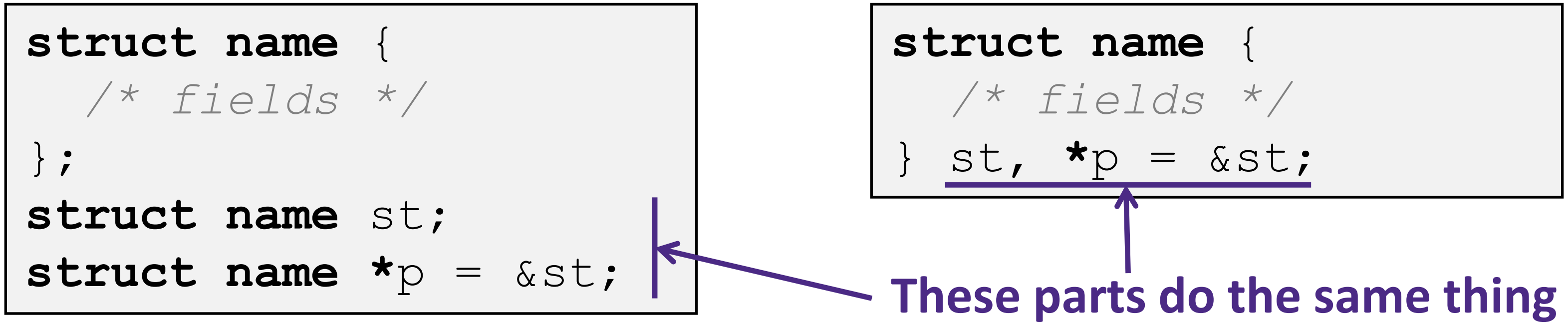
C. 28 bytes

D. 32 bytes

E. We're lost…

Aside: More Struct Definitions

- Can combine struct and instance definitions:



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

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

These parts do the same thing

- Defines a struct type (`struct name`), an instance of that type (`st`), and a pointer to that type (`p`)
- This syntax is difficult to read
  - Porter doesn’t like it in *most* situations because it conflates a type definition with an instance definition. But that’s just his opinion…
  - We are showing it because you may see it in code in the future (and on the homework ☺)

Aside: 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
  - `typedef` alone doesn’t create an instance of the struct!

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

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

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