Structs & Alignment

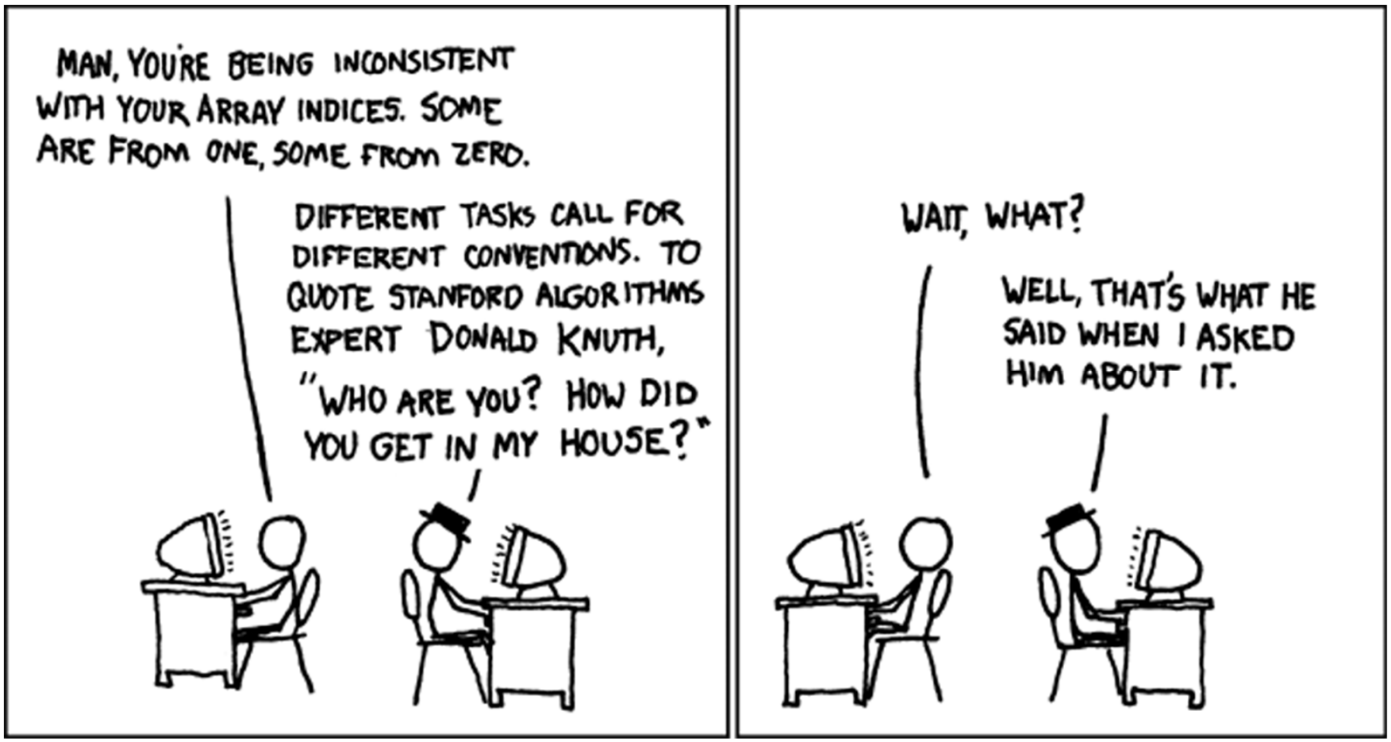
CSE 351 Spring 2019

Instructor:

Ruth Anderson

Teaching Assistants:

Gavin Cai
Jack Eggleston
John Feltrup
Britt Henderson
Richard Jiang
Jack Skalitzky
Sophie Tian
Connie Wang
Sam Wolfson
Casey Xing
Chin Yeoh



http://xkcd.com/163/

Administrivia

- Lab 2 (x86-64) due TONIGHT (5/01)
- Homework 3, due Wednesday (5/8)
  - On midterm material, but due after the midterm

- **Midterm** (Fri 5/03, 4:30-5:30pm in KNE 130)
  - Review Session: Thurs: 6:30-8:30pm in Sieg 134
  - No lecture on Friday 5/03
  - Ruth will hold office hours instead
    - Fri 11:30am-12:30pm in CSE 460
    - Fri 2:30-3:30pm in CSE 460

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:

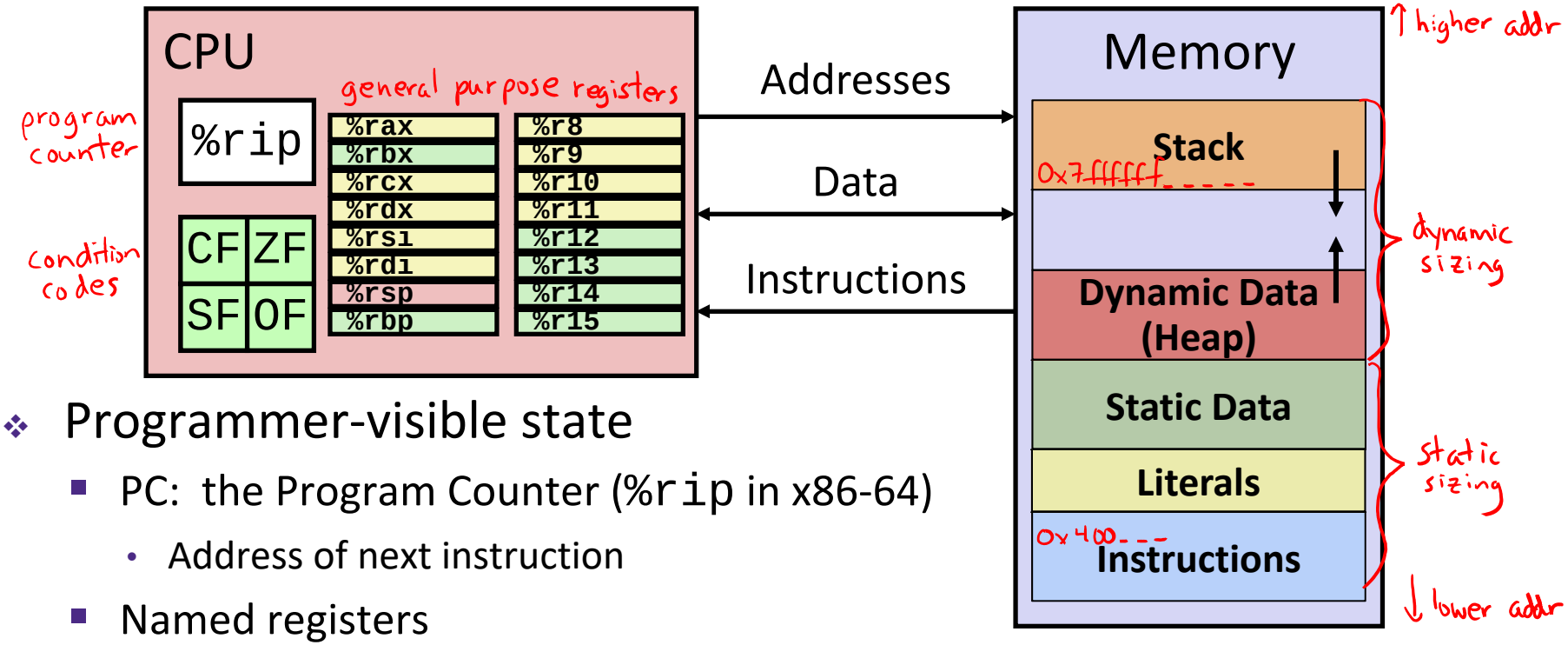
```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100011111
```

Computer system:

OS:

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

Assembly Programmer's View



- Programmer-visible state
  - PC: the Program Counter (`%rip` in x86-64)
    - Address of next instruction
  - Named registers
    - Together in "register file"
    - Heavily used program data
  - Condition codes
    - Store status information about most recent arithmetic operation
    - Used for conditional branching
- Memory
  - Byte-addressable array
  - Code and user data
  - Includes *the Stack* (for supporting procedures)

x86-64 Instructions

size specifiers: b, w, l, q
1, 2, 4, 8 bytes

1. Data movement
   - `mov, movs, movz, ...`

operand types: Imm $ Reg % Mem ( )

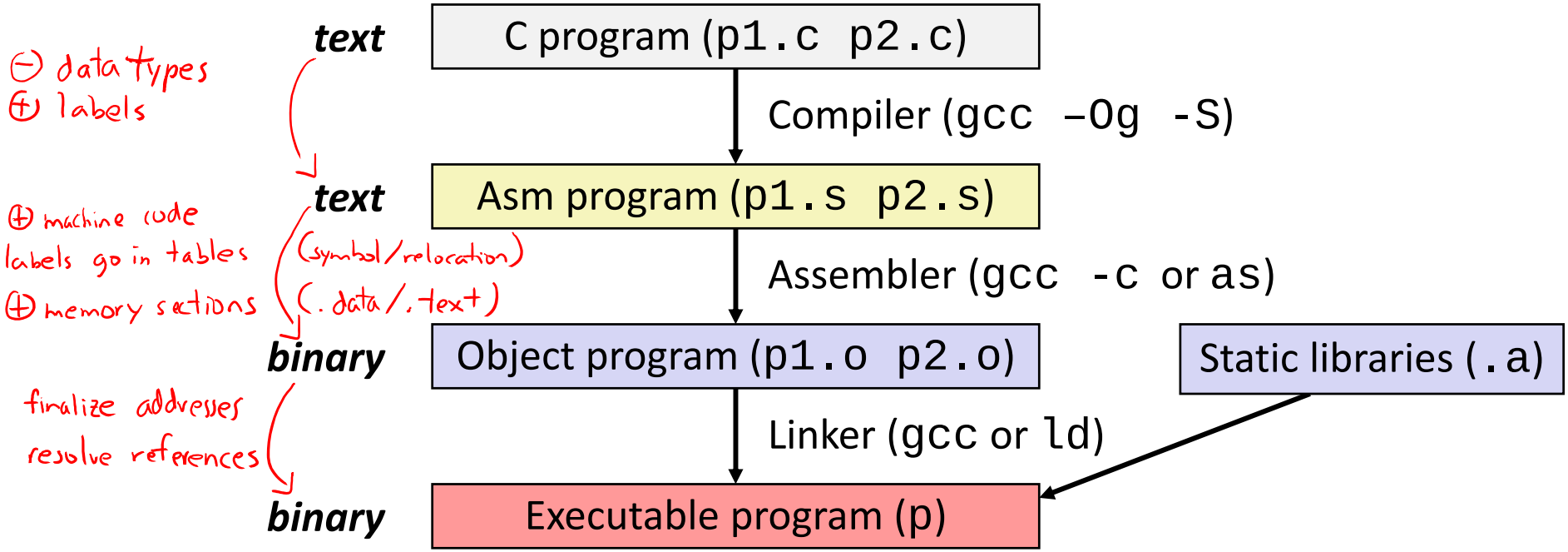
2. Arithmetic
   - `add, sub, shl, sar, lea, ...`

Labels are addresses

3. Control flow
   - `cmp, test, j*, set*, ...`
4. Stack/procedures
   - `push, pop, call, ret, ...`

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (`-Og`) [New to recent versions of GCC]
  - Put resulting machine code in file p



Assembling

- Executable has **addresses** (no more labels)

assembler

```
0000000004004f6 <pcount_r>:
  4004f6:  b8 00 00 00 00    mov    $0x0,%eax
  4004fb:  48 85 ff          test   %rdi,%rdi
  4004fe:  74 13             je     400513 <pcount_r+0x1d>
  400500:  53                push   %rbx
  400501:  48 89 fb          mov    %rdi,%rbx
  400504:  48 d1 ef          shr    %rdi
  400507:  e8 ea ff ff ff    callq  4004f6 <pcount_r>
  40050c:  83 e3 01          and    $0x1,%ebx
  40050f:  48 01 d8          add    %rbx,%rax
  400512:  5b                pop    %rbx
  400513:  f3 c3             rep ret
```

used to be a label (Exit: or .L6:)

pcount_r + 0x1d = 30 bytes after start of pcount_r

- gcc -g pcount.c –o pcount
- objdump –d pcount

A Picture of Memory (64-bit view)

```
00000000004004f6 <pcount_r>:
  4004f6:  b8 00 00 00 00     mov    $0x0,%eax
  4004fb:  48 85 ff           test   %rdi,%rdi
  4004fe:  74 13              je     400513 <pcount_r+0x1d>
  400500:  53                 push   %rbx
  400501:  48 89 fb           mov    %rdi,%rbx
  400504:  48 d1 ef           shr    %rdi
  400507:  e8 ea ff ff ff     callq  4004f6 <pcount_r>
  40050c:  83 e3 01           and    $0x1,%ebx
  40050f:  48 01 d8           add    %rbx,%rax
  400512:  5b                 pop    %rbx
  400513:  f3 c3              rep ret
```

instruction addresses

stored bytes

0|8	1|9	2|a	3|b	4|c	5|d	6|e	7|f	
								0x00
								0x08
								0x10
...								...
						b8	00	0x4004f0
00	00	00	48	85	ff	74	13	0x4004f8
53	48	89	fb	48	d1	ef	e8	0x400500
ea	ff	ff	ff	83	e3	01	48	0x400508
01	d8	5b	f3	c3				0x400510

unaligned, but more compact

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:

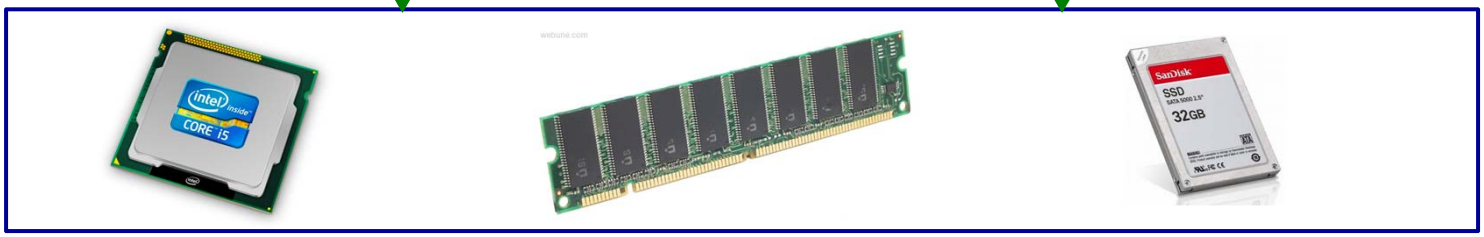
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

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

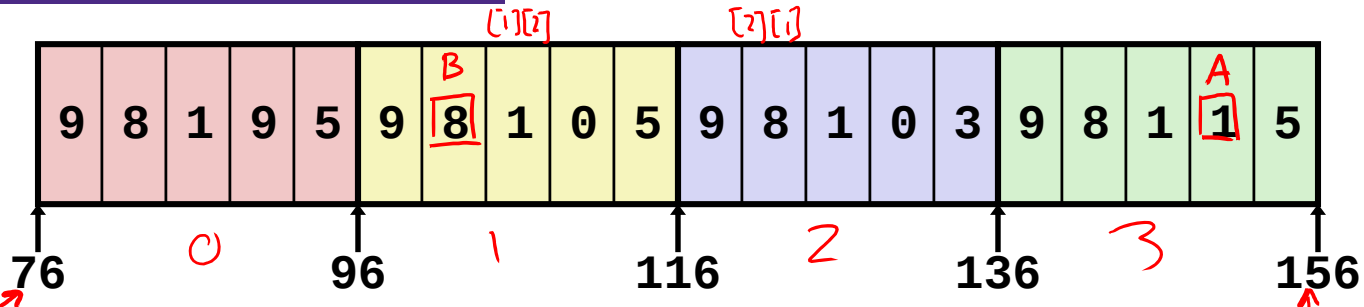
Peer Instruction Question

row-major:
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19

column-major:
0 4 8 12 16
1 5 9 13 17
2 6 10 14 18
3 7 11 15 19

- Which of the following statements is FALSE?
  - Vote at http://pollev.com/rea

```
int sea[4][5];
```



A. `sea[4][-2]` is a *valid* array reference
   Yes, returns 1
B. `sea[1][1]` makes *two* memory accesses
   No, only single memory access
C. `sea[2][1]` will *always* be a higher address than `sea[1][2]`
   Yes, because C is row-major
D. `sea[2]` is calculated using *only* `lea`
   Yes, sea[2] returns address of array row
E. We're lost…

Data Structures in Assembly

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

Structs in C

- Way of defining compound data types
- A structured group of variables, possibly including other structs

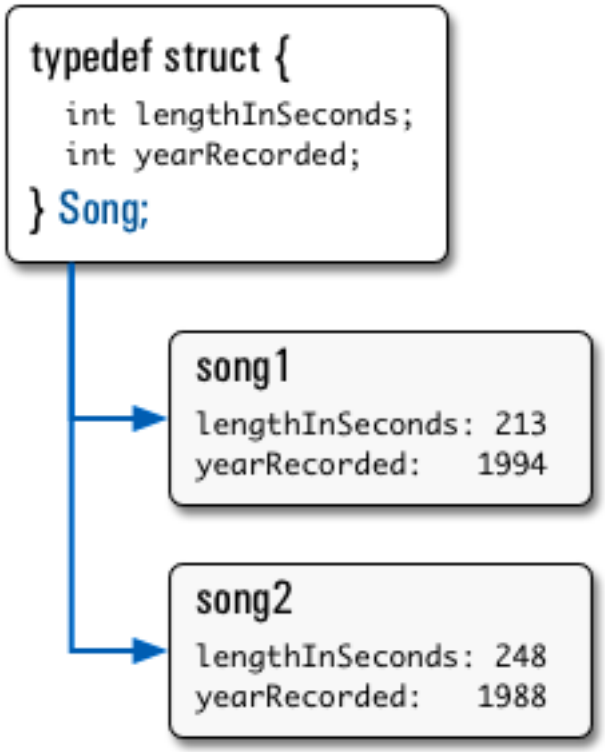
```
typedef struct {
  int lengthInSeconds;
  int yearRecorded;
} Song;

Song song1;

song1.lengthInSeconds =  213;
song1.yearRecorded    = 1994;

Song song2;

song2.lengthInSeconds =  248;
song2.yearRecorded    = 1988;
```



typedef unsigned long int uli;
old var type
new name

Struct Definitions

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

your choice

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

Easy to forget semicolon!

pointer

```
struct name name1, *pn, name_ar[3];
```

instance

array

- 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 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 = 24 bytes

16
8

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 for short: `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`

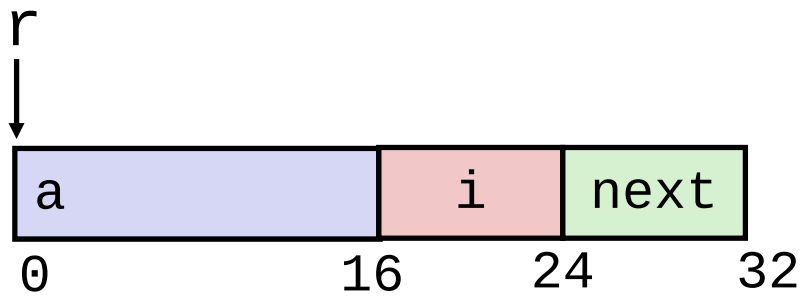
- In Java, almost everything is a pointer ("*reference*") to an object
  - Cannot declare variables or fields that are structs or arrays
  - Always a *pointer* to a struct or array
  - So every Java variable or field is ≤ 8 bytes (but can point to lots of data)

Structure Representation

struct definition

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

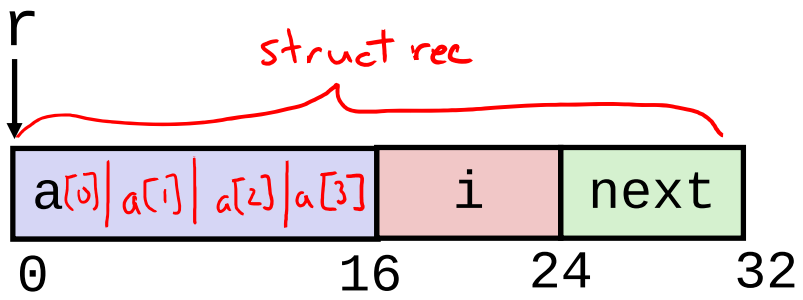
declare a pointer



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

Structure Representation

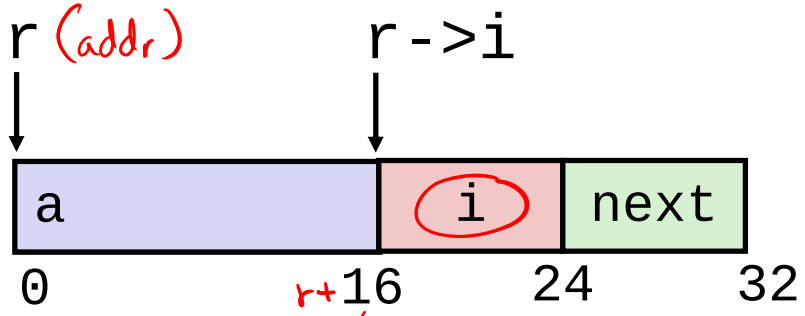
```
struct rec {
  ①  int a[4];
  ②  long i;
  ③  struct rec *next;
} *r;
```



- 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;
} *r;
```



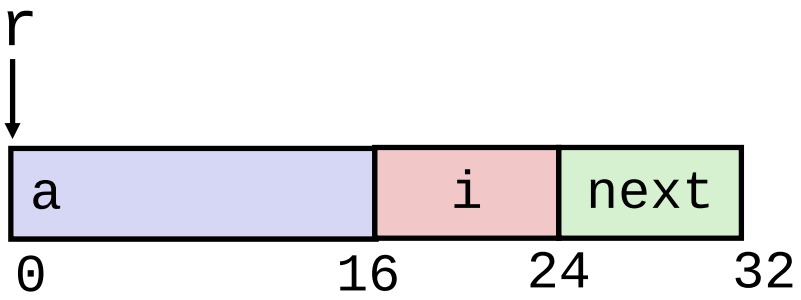
- Compiler knows the *offset* of each member within a struct
  - Compute as `*(r+offset)`
    - Referring to absolute offset, so no pointer arithmetic

```
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;
} *r;
```



pointer

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

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

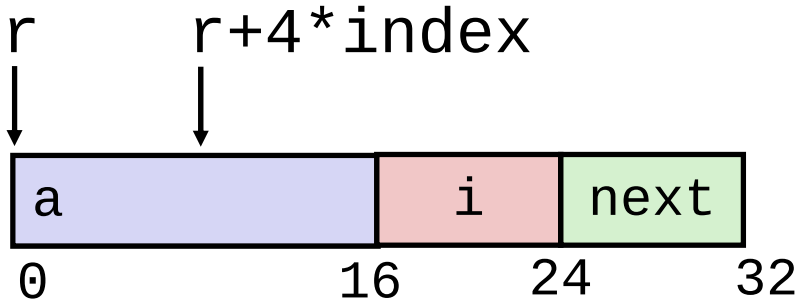
want address

```
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;
} *r;
```



- 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.

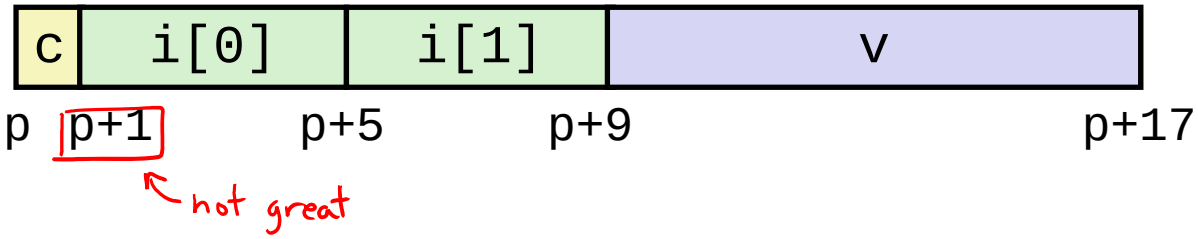
Alignment Principles

- 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 just possibly slower because we have to load/read more data

Structures & Alignment

- Unaligned Data

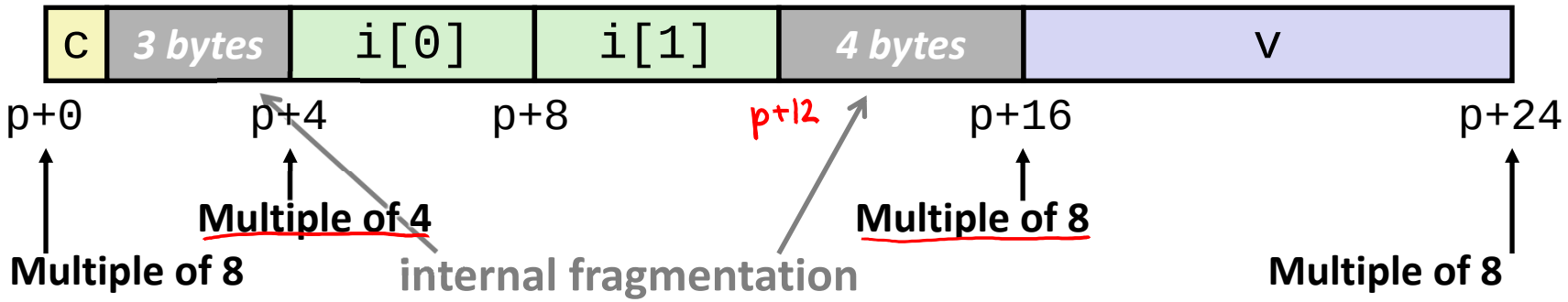



```
struct S1 {
① char c;      ← 1
② int i[2];    ← 4
③ double v;    ← 8
} *p;
```

K

24 B total

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



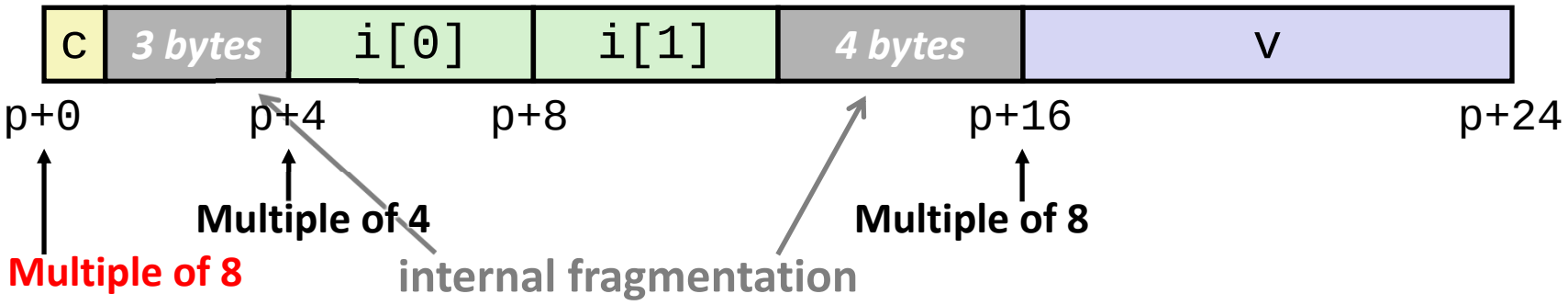
Satisfying Alignment with Structures (1)

- 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;
} *p;
```

K
1
4
8
$K_{max} = 8$

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

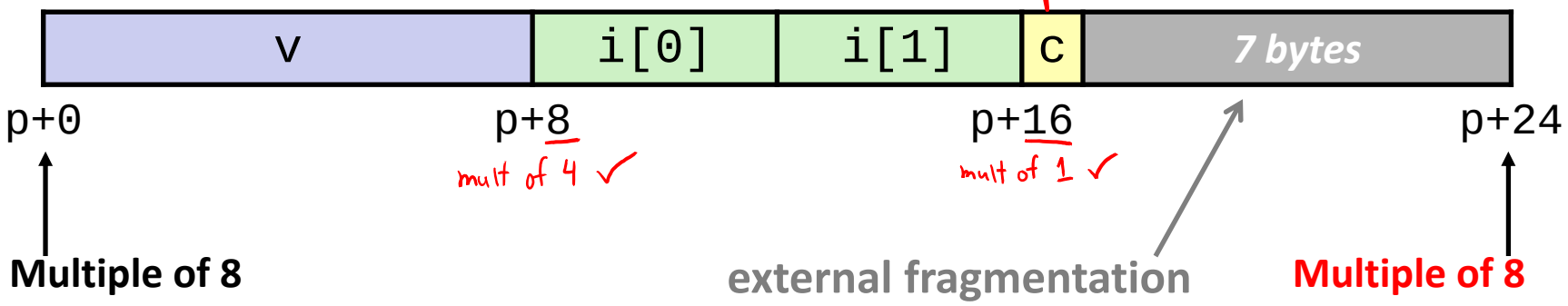


Satisfying Alignment with Structures (2)

- 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;
} *p;
```

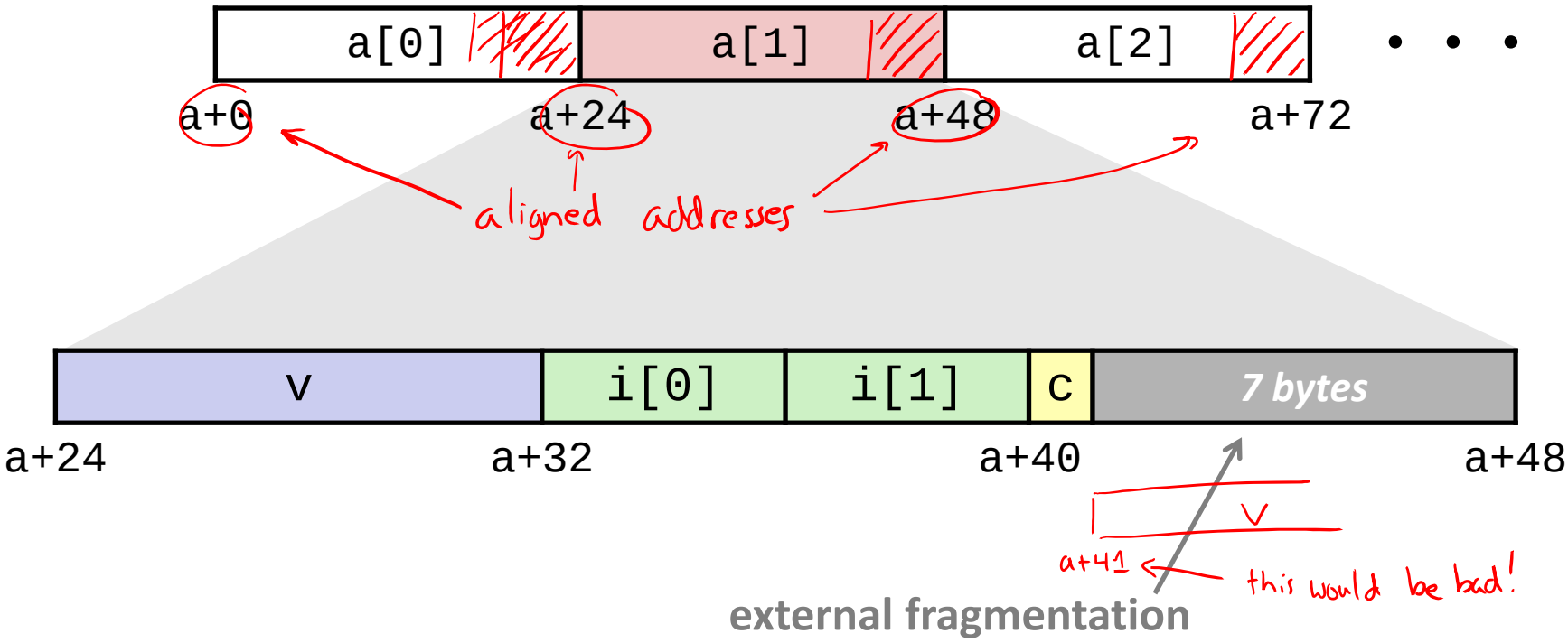
- For largest alignment requirement $K_{\max}$, overall structure size must be multiple of $K_{\max}$
  - Compiler will add padding at end of structure to meet overall structure alignment requirement



Arrays of Structures

- Overall structure 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

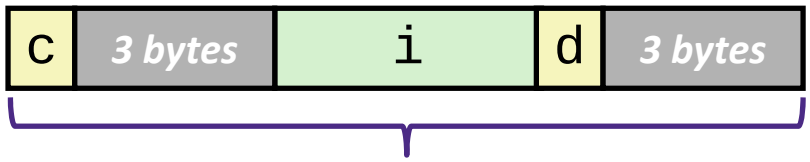
- Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each ***field*** must be aligned *within* the struct *(may insert padding)*
    - `offsetof` can be used to get actual field offset
  - Overall struct must be ***aligned*** according to largest field
  - Total struct ***size*** must be multiple of its alignment *(may insert padding)* so that if we just to store an array of structures, they are all aligned (e.g. so we can store the array elements contiguously in memory)
    - `sizeof` should be used to get true size of structs

How the Programmer Can Save Space

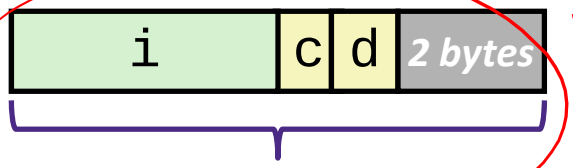
- Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```



12 bytes



8 bytes

same data, but more efficient!

Peer Instruction Question

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

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

```
struct old {
  int i;
  short s[3];
  char *c;
  float f;
};
```

K: 4, 2, 8, 4 → $K_{max} = 8$

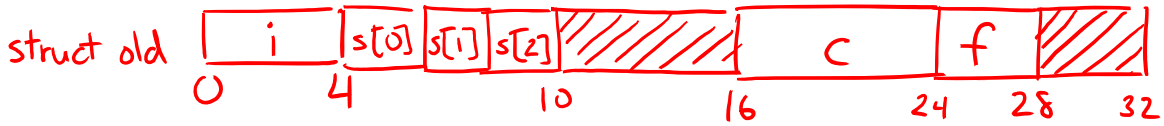
```
struct new {
  int    i;
  float  f;
  char*  c;
  short  s[3];
};
```

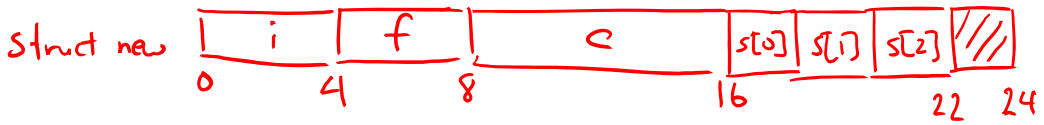
could also switch these (internal vs. external frag)

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

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

A. 16 bytes
B. 22 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 in order declared
  - Pad in middle and at end to satisfy alignment