Structs and Alignment

CSE 351 Spring 2017

Instructor:

Ruth Anderson

Teaching Assistants:

Dylan Johnson

Kevin Bi

Linxing Preston Jiang

Cody Ohlsen

Yufang Sun

Joshua Curtis

Administrivia

- Lab 2 due TONIGHT (4/26)
- Homework 3 coming soon
- Lab 3 coming soon

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:

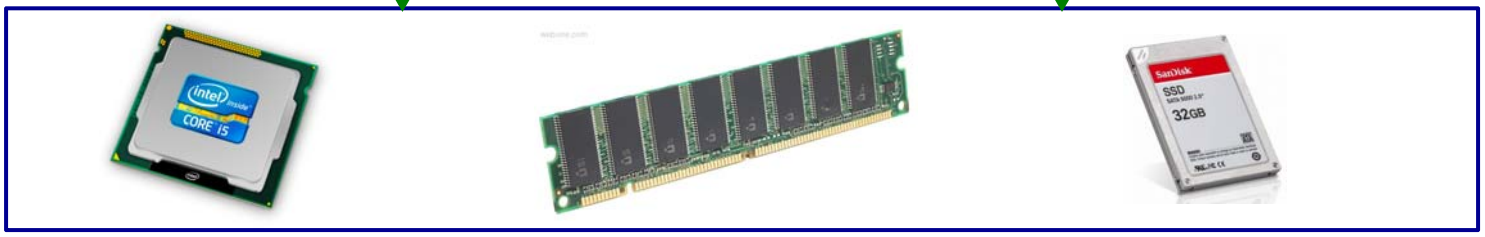
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111110100011111
```

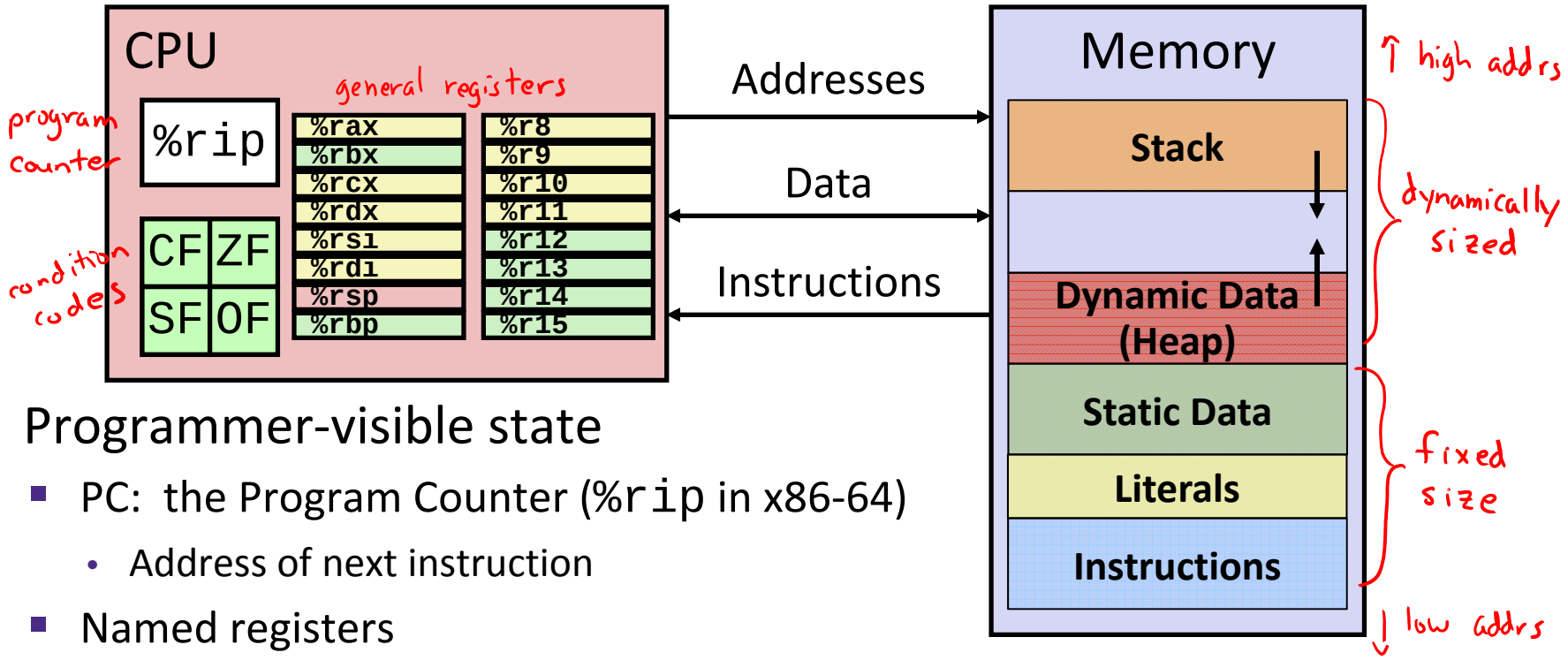
OS:

Computer system:



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

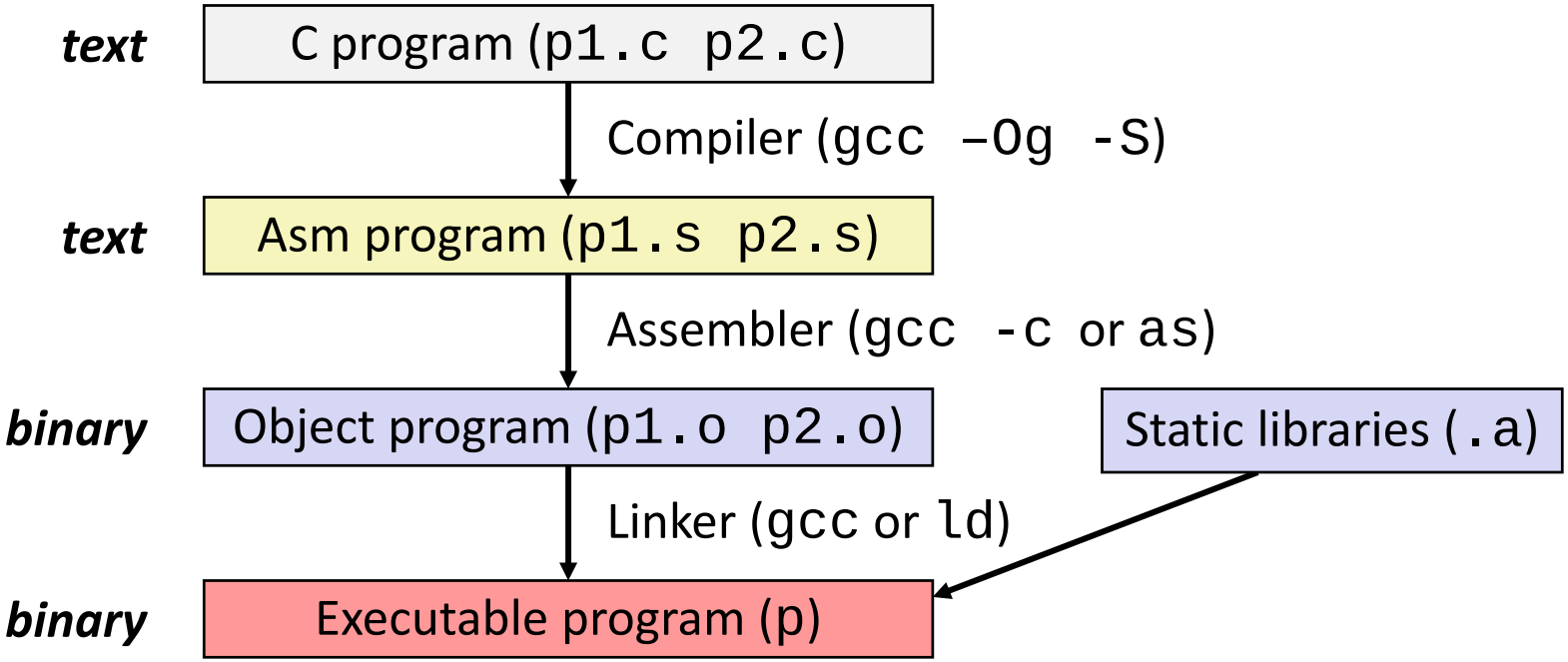
Assembly Programmer's View



- 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)

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**

assembler

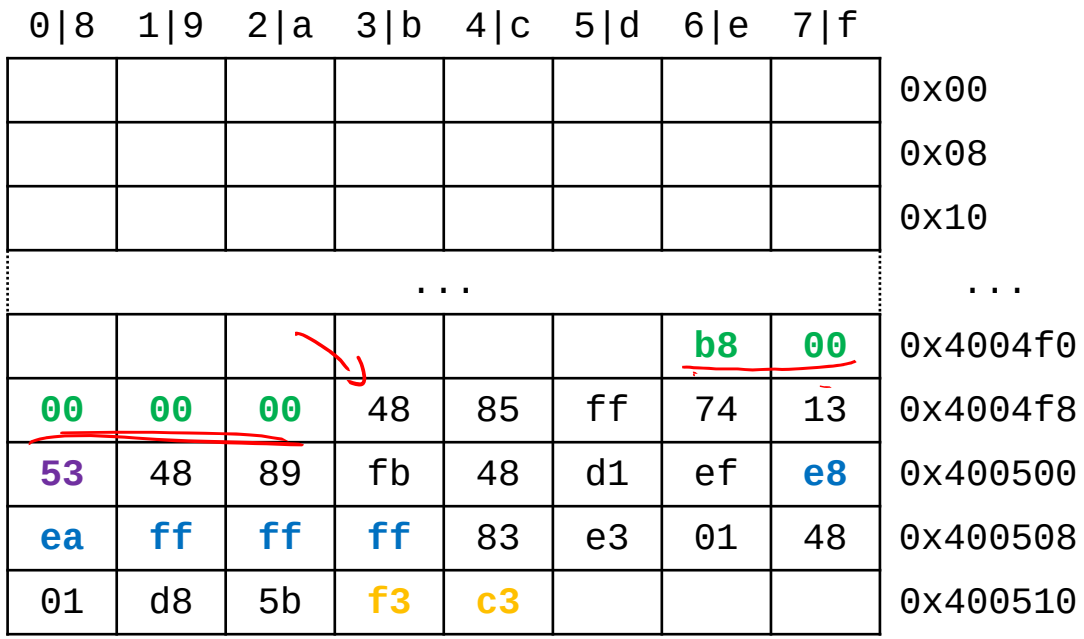
```
0000000004004f6 <pcount_r>:
  4004f6:  b8 00 00 00 00     mov    $0x0,%eax
  4004fb:  48 85 ff           test   %rdi,%rdi
  4004fe:  74 13              je     400513 <pcount_r+0x1d>
  400500:  53                 push   %rbx
  400501:  48 89 fb           mov    %rdi,%rbx
  400504:  48 d1 ef           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
```

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
```



not aligned, 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:

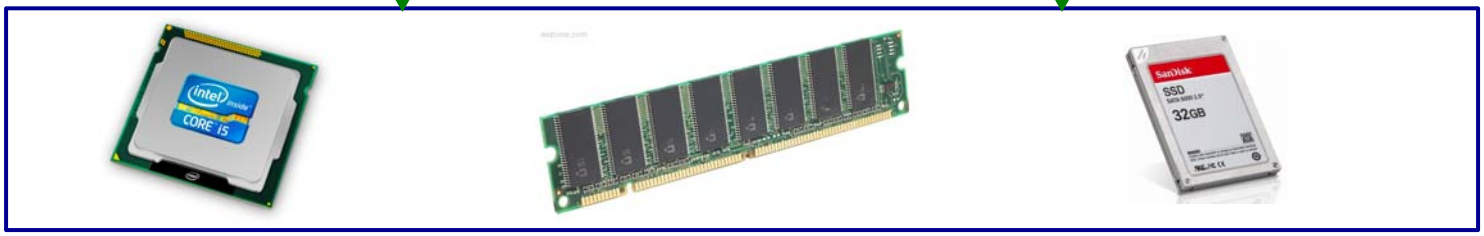
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Computer system:



- Memory & data
- Integers & floats
- x86 assembly
- Procedures & stacks
- Executables
- Arrays & structs
- Memory & caches
- Processes
- Virtual memory
- Operating Systems

Data Structures in Assembly

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

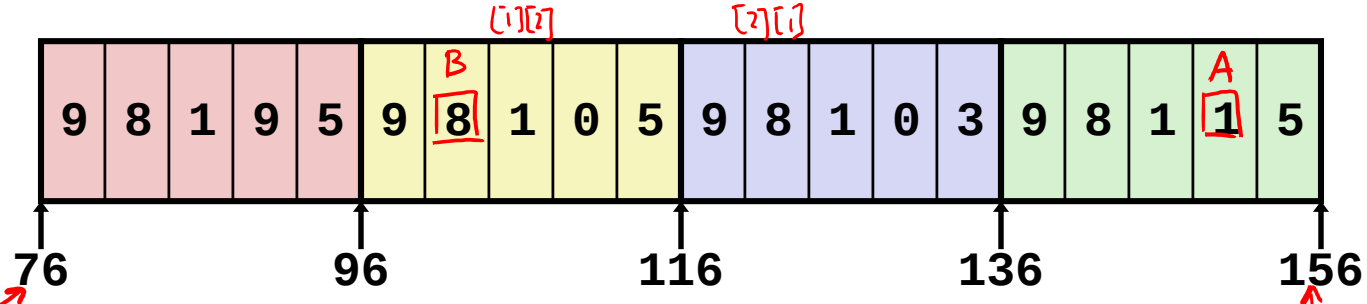
Question

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

colum-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?
  -

```
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

Structs in C

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

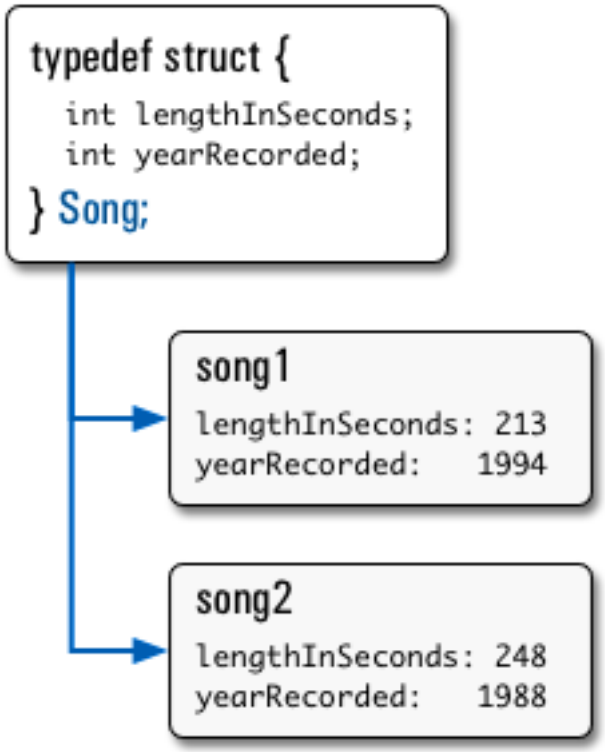
```
typedef struct {
  int lengthInSeconds;
  int yearRecorded;
} Song;

Song song1;

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

Song song2;

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



Review: Structs in Lab 0

```
// Use typedef to create a type: FourInts
typedef struct {
   int a, b, c, d;
} FourInts;  // Name of type is "FourInts"

int main(int argc, char* argv[]) {
   FourInts f1;  // Allocates memory to hold a FourInts
                 // (16 bytes) on stack (local variable)
   f1.a = 0;     // Assign first field in f1 to be zero

   FourInts* f2; // Declare f2 as a pointer to FourInts

   // Allocate space for a FourInts on the heap,
   //   f2 is a "pointer to"/"address of" this space.
   f2 = (FourInts*) malloc(sizeof(FourInts));
   f2->b = 17;   // Assign the second field to be 17
   ...
}
```

Aside: Syntax for structs without typedef

```
struct rec {            // Declares the type "struct rec"
    int a[4];           // Total size = ____ bytes
    long i;
    struct rec *next;
};
struct rec r1;          // Allocates memory to hold a struct rec
                        // named r1, on stack or globally,
                        // depending on where this code appears

struct rec *r;          // Allocates memory for a pointer
r = &r1;                // Initializes r to "point to" r1
```

4 * 4 = 32

8

8

r1.i = val

r->i = val

(*r).i = val

More Structs Syntax

Declaring a struct **`struct rec`**, then declaring a variable **`r1`**:

```
struct rec {          // Declares the type "struct rec"
    int a[4];
    long i;
    struct rec *next;
};
struct rec r1;        // Declares r1 as a struct rec
```

Equivalent to:

```
struct rec {          // Declares the type "struct rec"
    int a[4];
    long i;
    struct rec *next;
} r1;                 // Declares r1 as a struct rec
```

Declare type `struct rec` and variable `r1` at the same time!

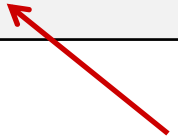
Another Syntax Example

Declaring a struct **`struct rec`**, then declaring a variable **`r`**:

```
struct rec {          // Declares the type "struct rec"
    int a[4];
    long i;
    struct rec *next;
};
struct rec *r;        // Declares r as pointer to a struct rec
```

Equivalent to:

```
struct rec {          // Declares the type "struct rec"
    int a[4];
    long i;
    struct rec *next;
} *r;                 // Declares r as pointer to a struct rec
```



Declare type `struct rec` and variable `r` at the same time!

Struct Definitions

- Structure definition:
  - Does NOT declare a variable

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

Easy to forget semicolon!

- Variable definitions:
  - Variable type is "`struct name`"

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

pointer

array

- Joint struct definition and typedef

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

① define struct

② typedef

combined

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

unnamed!

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

4B x 4 = 8B

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

← Size = 24 bytes

Size = 32 bytes →

```
struct rec {
    int a[4];
    long i;
8B  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
② access field
equivalent

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

Java side-note

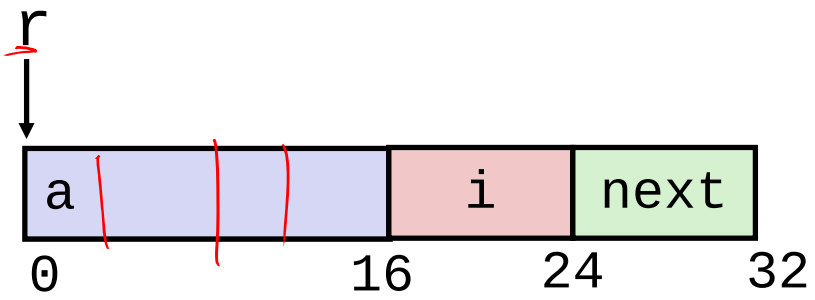
```
class Record { ... }
Record x = new Record();
```

- 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

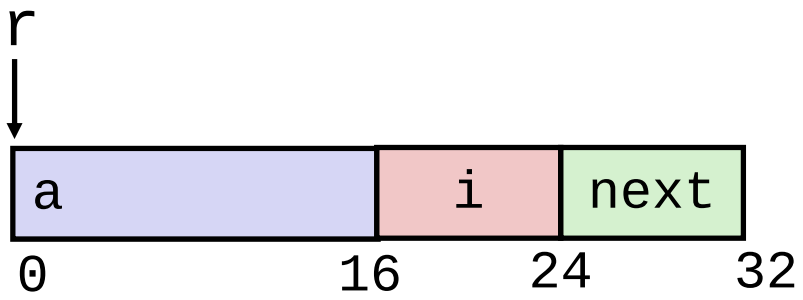
```
struct rec {
    int a[4];
    long i;
    struct rec *next;
} *r;
```



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

Structure Representation

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

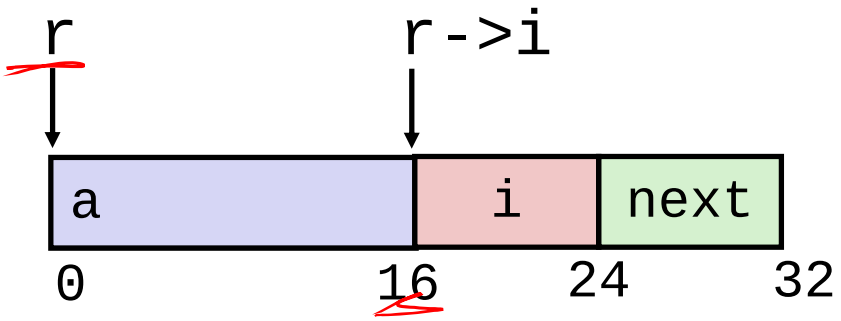


- Structure represented as block of memory
  - Big enough to hold all of the fields
- Fields ordered according to declaration order
  - Even if another ordering would be more compact
- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structures in the source code

Accessing a Structure Member

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

declared pointer to struct rec



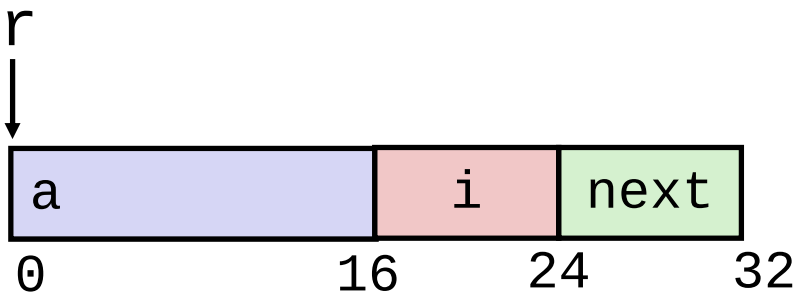
- Compiler knows the *offset* of each member within a struct
  - Compute as `*(r+offset)`
    - Referring to absolute offset, so no pointer arithmetic

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

```
  # r in %rdi
  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
```

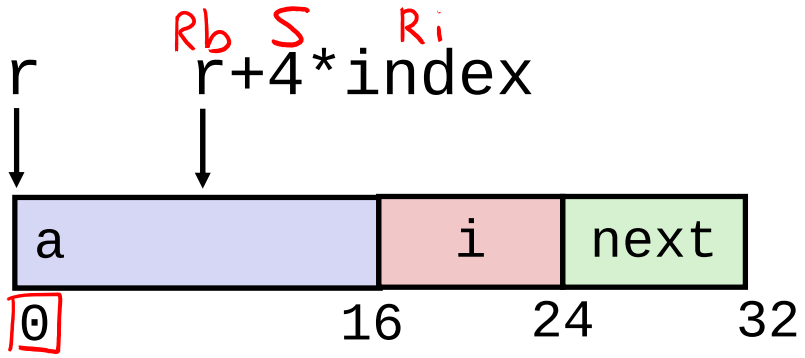
want address

```
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

0(%rdi) (,%rsi,4)

```
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

- For good memory system performance, Intel recommends data be aligned
  - However the x86-64 hardware will work correctly regardless of alignment of data
- *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, * (pointers)`	Lowest 3 bits zero: $\ldots000_2$
16	`long double`	Lowest 4 bits zero: $\ldots0000_2$

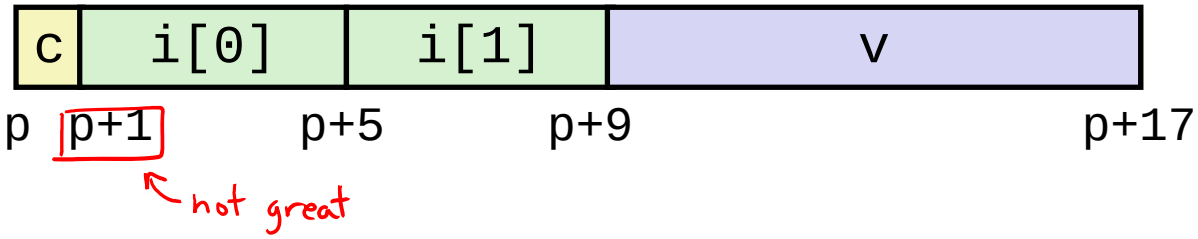
Alignment Principles

- 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 4 or 8 bytes (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)

Structures & Alignment

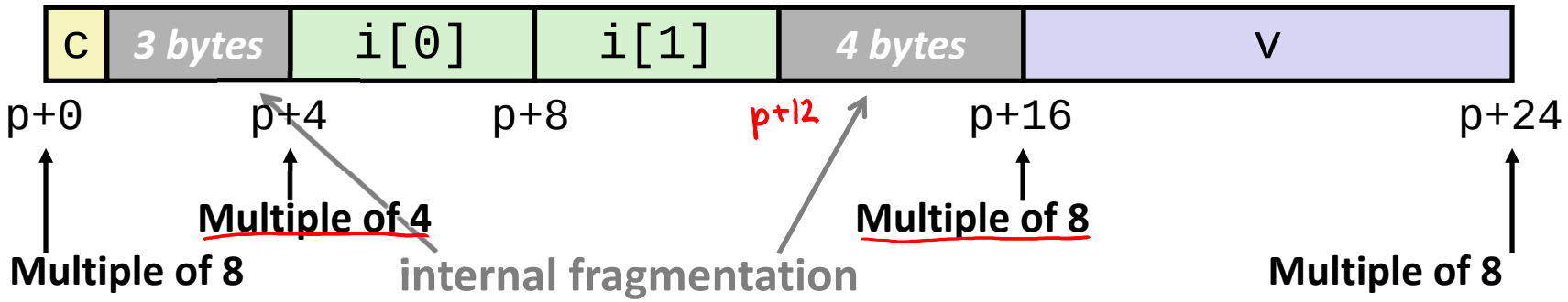
- Unaligned Data




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

K
1
4
8

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

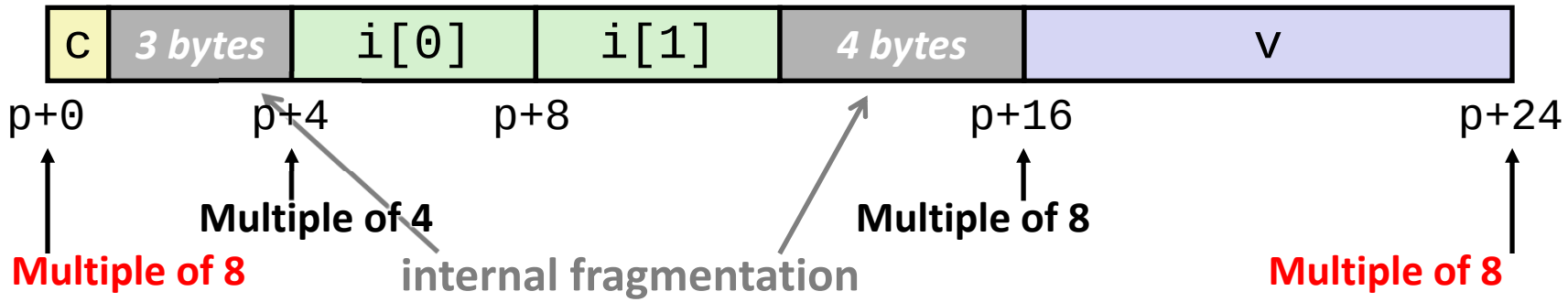


Satisfying Alignment with Structures (1)

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

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

- 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 individual items in the array as elements (entire array is not an "element")
  - **Address of structure & structure length must be multiples of $K_{\max}$**
- Example:
  - $K_{\max}$ = 8, due to `double` element

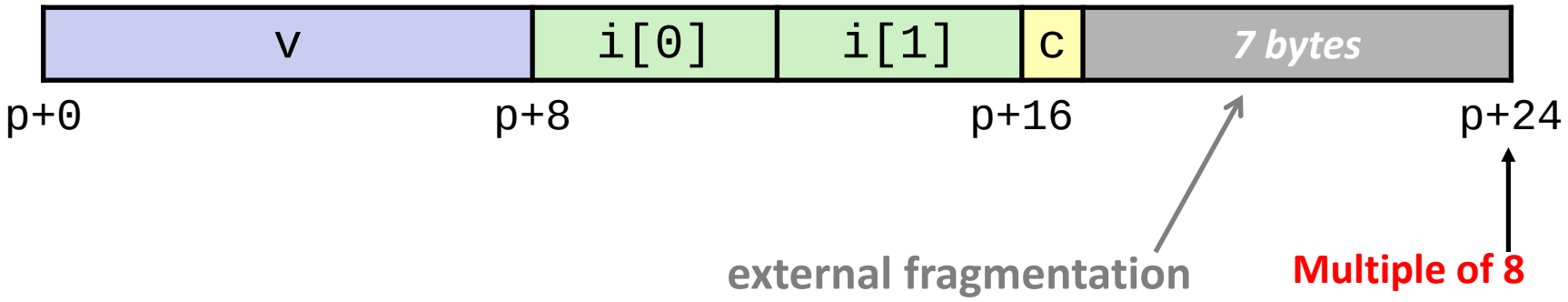


Satisfying Alignment with Structures (2)

- 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



Alignment of Structs

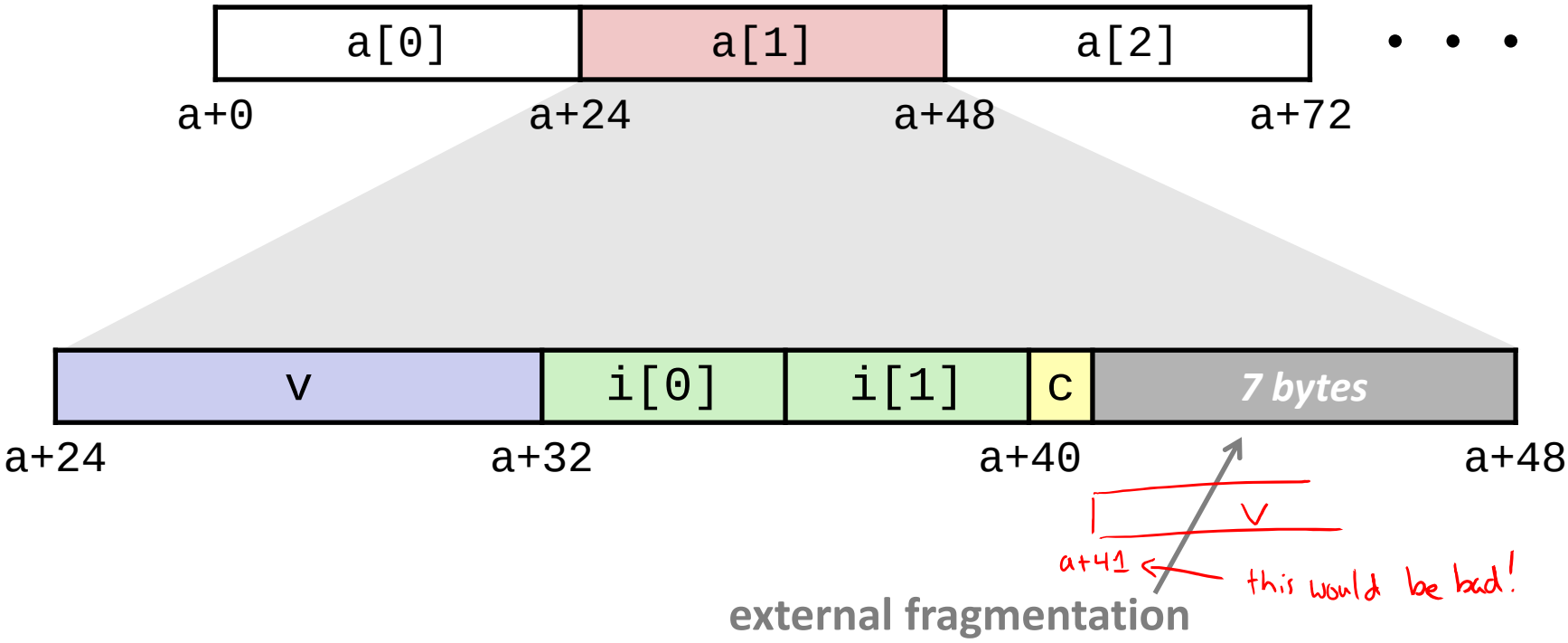
- Compiler will do the following:
  - Maintains declared *ordering* of fields in struct
  - Each ***field*** must be aligned *within* the struct *(may insert padding)*
    - `offsetof` can be used to get actual field offset
  - Overall struct must be ***aligned*** according to largest field
  - Total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

Arrays of Structures

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

Create an array of ten S2 structs called "a"

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



Accessing Array Elements

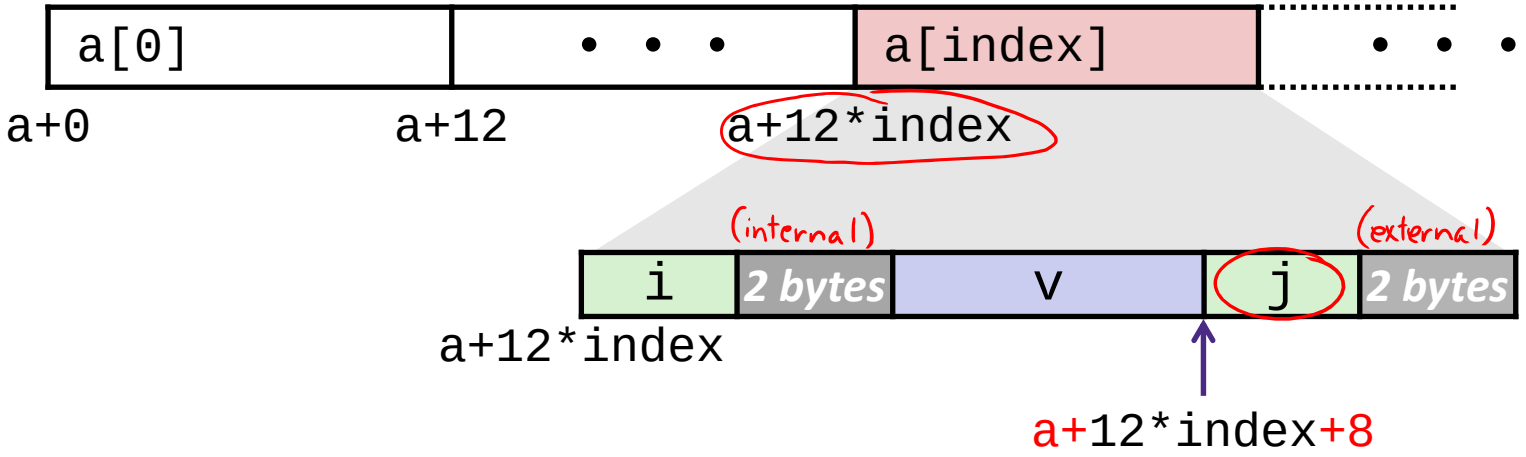
Create an array of ten S3 structs called "a"

```
struct S3 {
  short i;
  float v;
  short j;
} a[10];
```

K / 2 / 4 / 2

$K_{max} = 4$

- Compute start of array element as: `12*index`
  - `sizeof(S3) = 12`, including alignment padding
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8`




```
short get_j(int index)
{
  return a[index].j;
}
```

```
# %rdi = index
leaq (%rdi,%rdi,2),%rax  # 3*index
movzwl a+8(,%rax,4),%eax
```

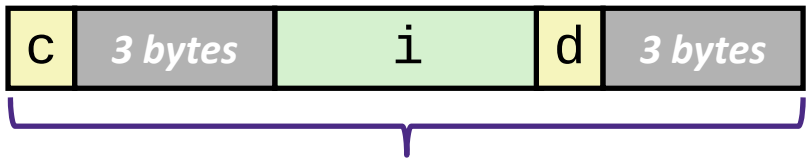
(3*index)*4

How the Programmer Can Save Space

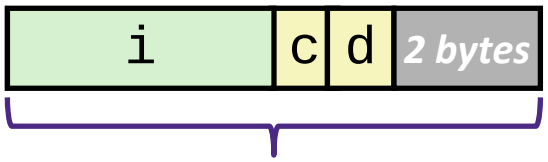
- Compiler must respect order elements are declared in
  - Sometimes the programmer can save space by declaring large data types first

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

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



12 bytes



8 bytes

Question

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

K

4

2

8

4

K_{max} = 8

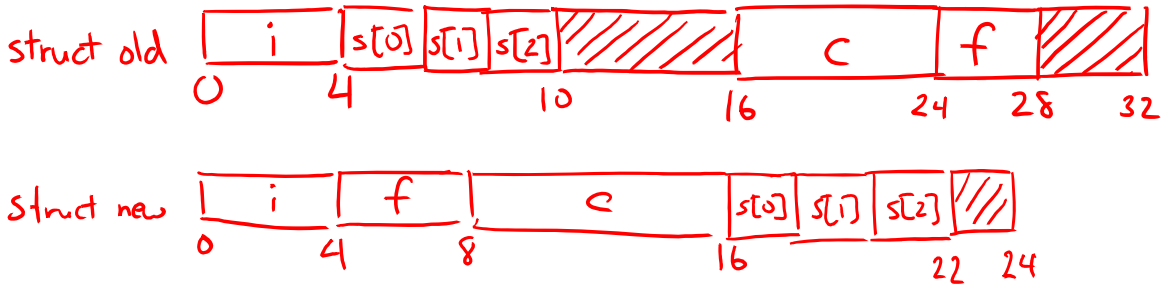
```
struct old {
  int i;
  short s[3];
  char *c;
  float f;
};
```

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

could also switch these (internal vs. external frag)

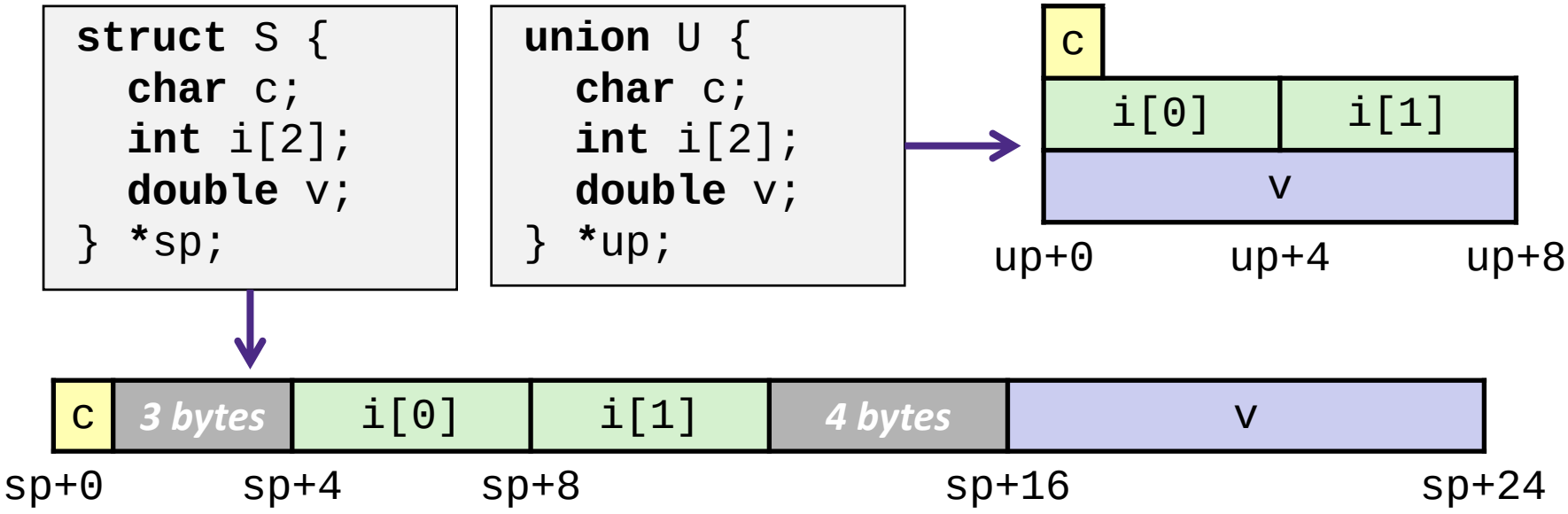
- What are the old and new sizes of the struct?

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



Unions

- Only allocates enough space for the largest element in union
- Can only use one member at a time



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
- Unions
  - Provide different views of the same memory location