Representing Data Structures

Multidimensional arrays

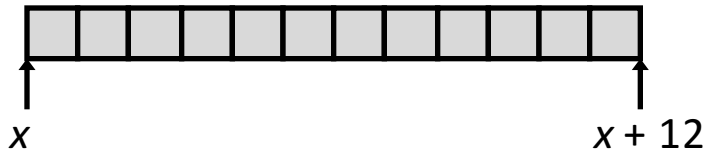
Structs

Array Layout and Indexing

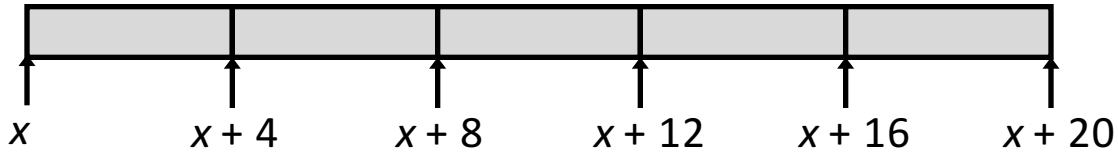
Array of length *N* with elements of type *T* and name *A*

Contiguous block of *N*sizeof(T)* bytes of memory

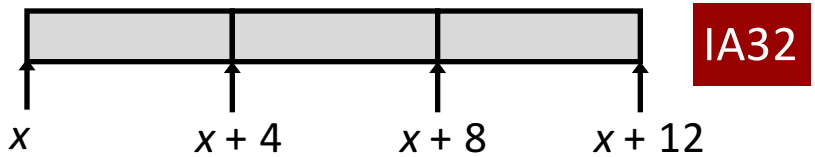
```
char string[12];
```




```
int val[5];
```




```
char* p[3];
```

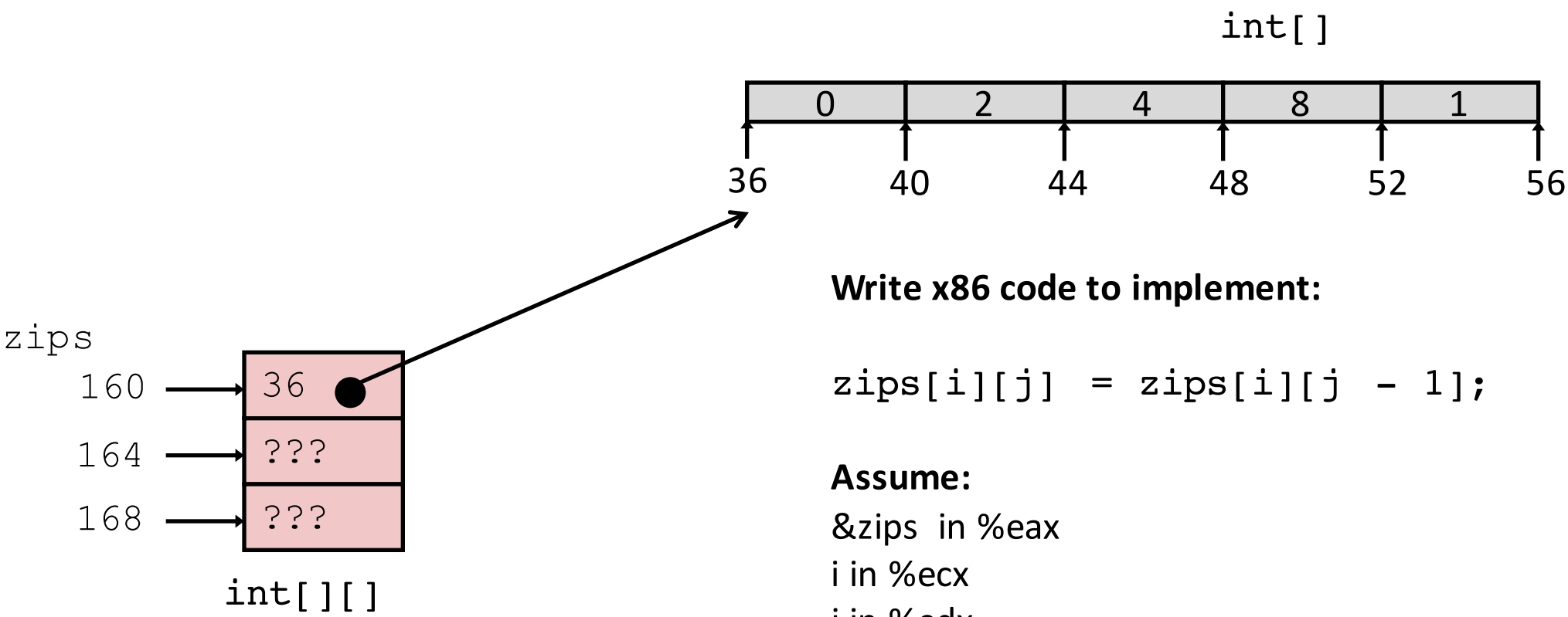


Write x86 code to load val[i] into %eax.

1. Assume:
   - &val is in %edx
   - i is in %ecx
2. Assume:
   - &val is -28(%ebp)
   - i is in %ecx

Multi-Level Arrays: C

```
int** zips = (int**)malloc(sizeof(int*)*3);
zips[0] = (int*)malloc(sizeof(int)*5);
...
```



Write x86 code to implement:

```
zips[i][j] = zips[i][j - 1];
```

Assume:

&zips in %eax

i in %ecx

j in %edx

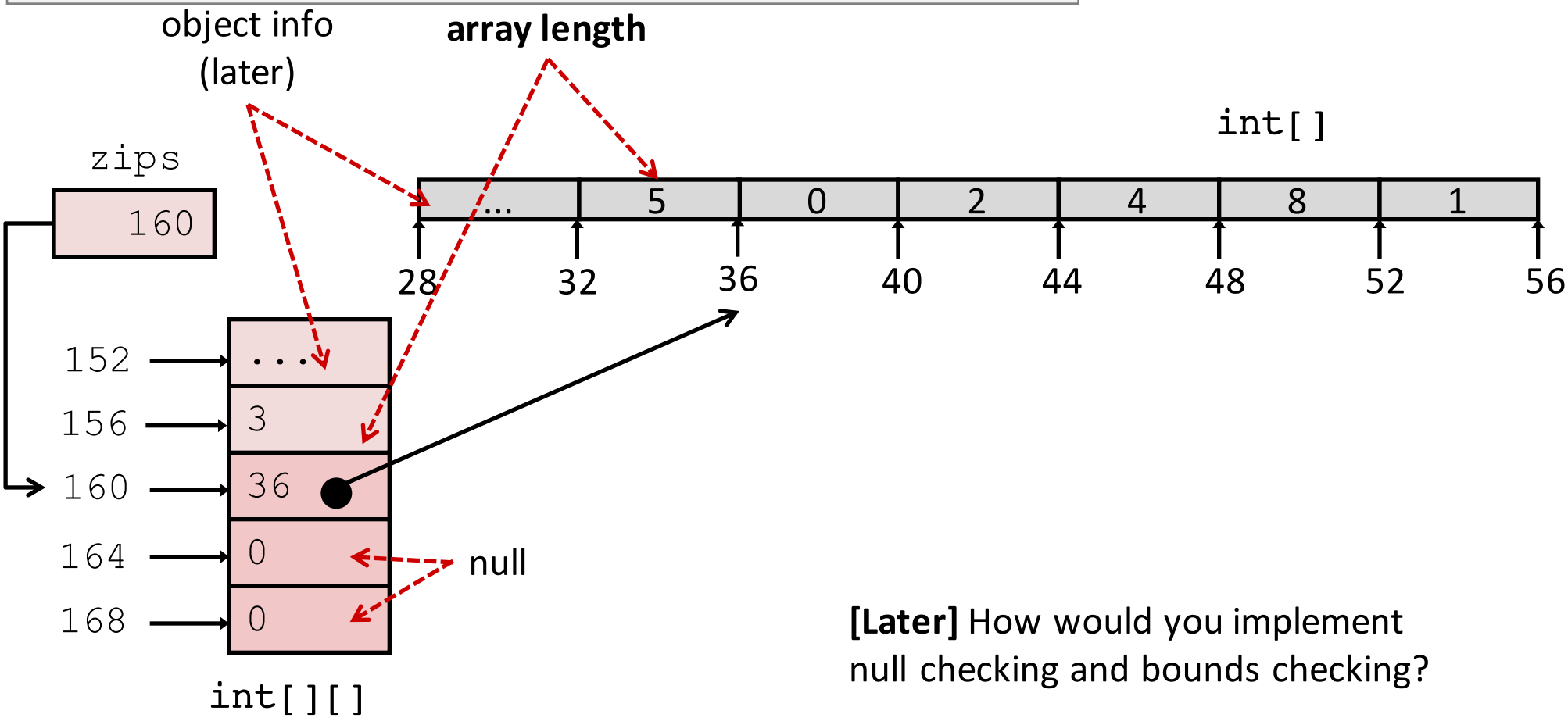
Multi-Level Arrays: Java

```
int[][] zips = new int[3][];
zips[0] = new int[5] {0, 2, 4, 8, 1};
```

Java

```
int** zips = (int**)malloc(sizeof(int*)*3);
zips[0] = (int*)malloc(sizeof(int)*5);
...
```

C



[Later] How would you implement null checking and bounds checking?

Nested Arrays

Declaration

2D array of data type T

R rows, C columns

Type T element requires K bytes

Layout

One contiguous block of memory

C: Row-major ordering

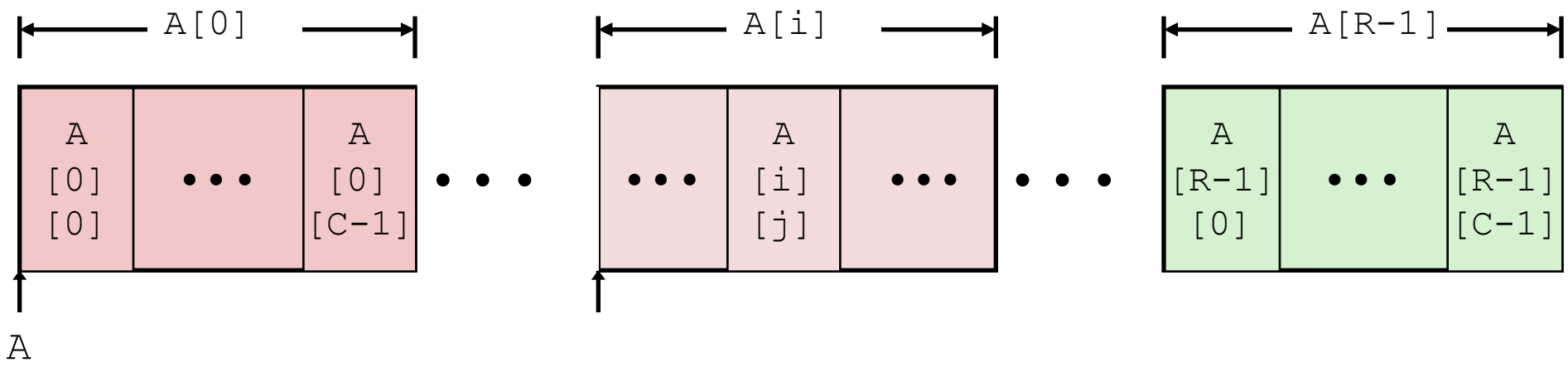
```
[ A[0][0]    • • •    A[0][C-1]
    •                     •
    •                     •
    •                     •
  A[R-1][0]  • • •  A[R-1][C-1] ]
```

```
int A[R][C];
```

A [0] [0]	• • •	A [0] [C-1]	A [1] [0]	• • •	A [1] [C-1]	• • •	A [R-1] [0]	• • •	A [R-1] [C-1]

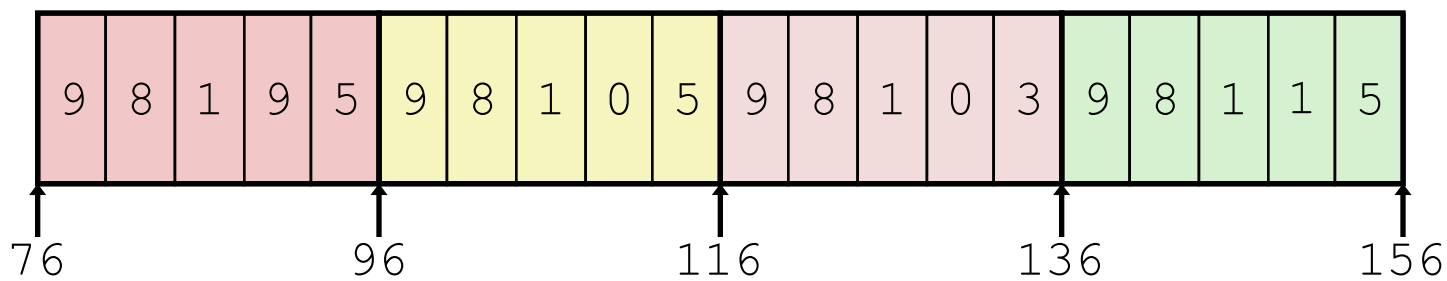
Nested Array Indexing

```
int A[R][C];
```



Strange Referencing Examples

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



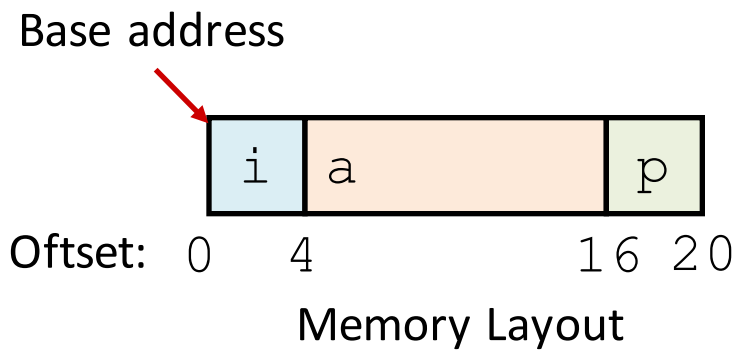
Reference	Address	Value	Guaranteed?
`sea[3][3]`	`76+20*3+4*3 = 148`	`1`	Yes
`sea[2][5]`			
`sea[2][-1]`			
`sea[4][-1]`			
`sea[0][19]`			
`sea[0][-1]`			

C does not do any bounds checking.

Row-major array layout is guaranteed.

C structs

```
struct rec {
  int i;
  int a[3];
  int* p;
};
```

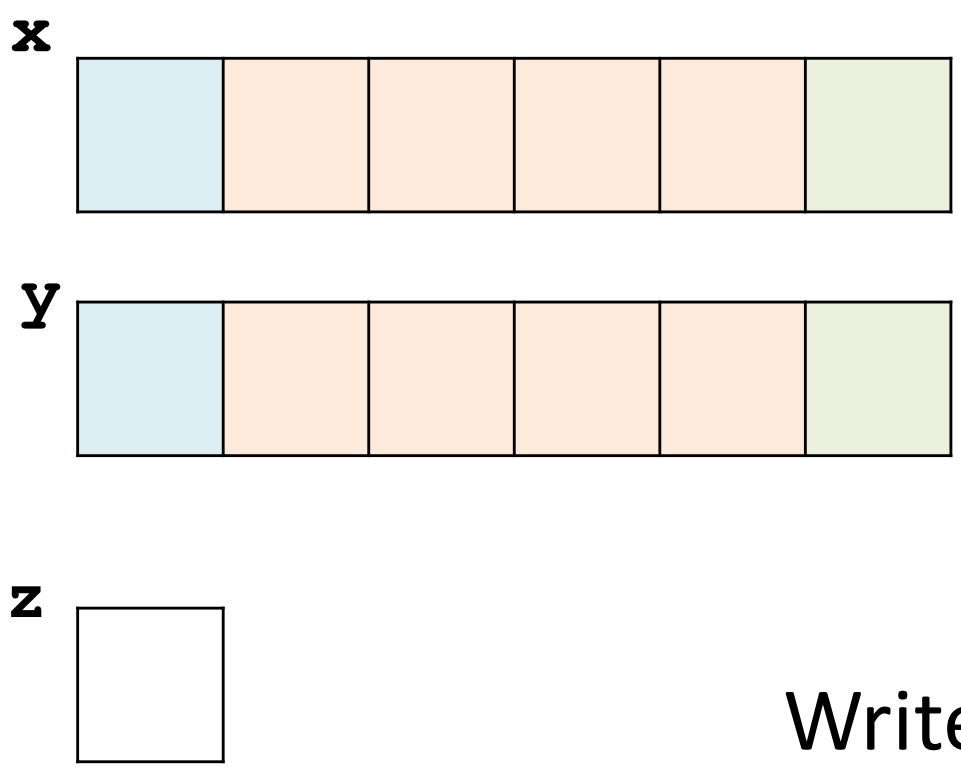


Like Java class/object, but without methods.

Compiler determines:

- Total size
- Offset of each field

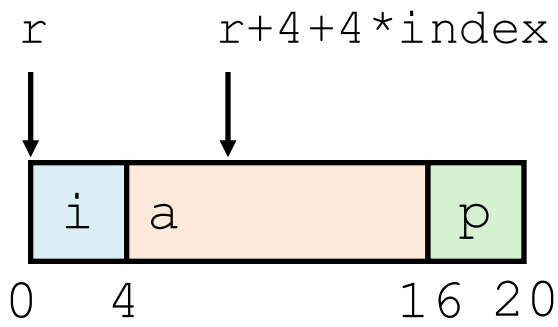
```
struct rec x;
struct rec y;
x.i = 1;
x.a[1] = 2;
x.p = &(x.i);
// copy full struct:
y = x;

struct rec* z;
z = &y;
(*z).i++;
// same as:
z->i++
```



Write x86.

Accessing Struct Field

```
struct rec {
  int i;
  int a[3];
  int* p;
};
```




```
int get_i_plus_elem(struct rec* r, int index) {
   return r->i + r->a[index];
}
```

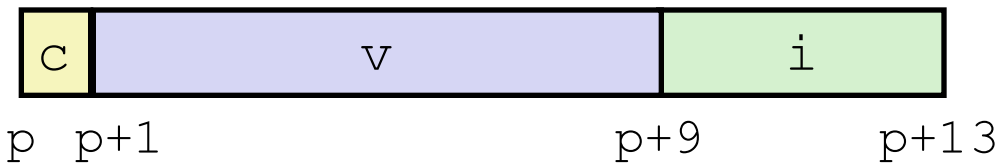
```
# %ecx = index
# %edx = r
movl 0(%edx),%eax            # Mem[r+0]
addl 4(%edx,%ecx,4),%eax     # Mem[r+4*index+4]
```

typedef

```
// give type T another name: U
typedef T U;

// struct types can be verbose
struct ListNode { ... };
...
struct ListNode* n = …;

// typedef can help
typedef struct list_node {
      ...
} ListNode;
...
ListNode* n = ...;
```

Struct alignment (1)

Unaligned Data




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

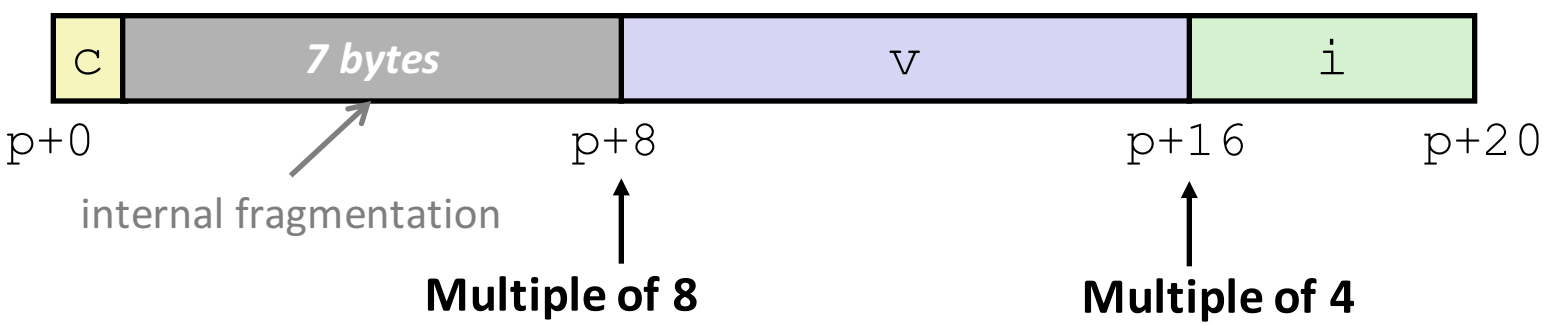
Defines new struct type and declares variable p of type struct S1*

Aligned Data

Primitive data type requires ***K*** bytes

Address must be multiple of ***K***

C: align every struct field accordingly.

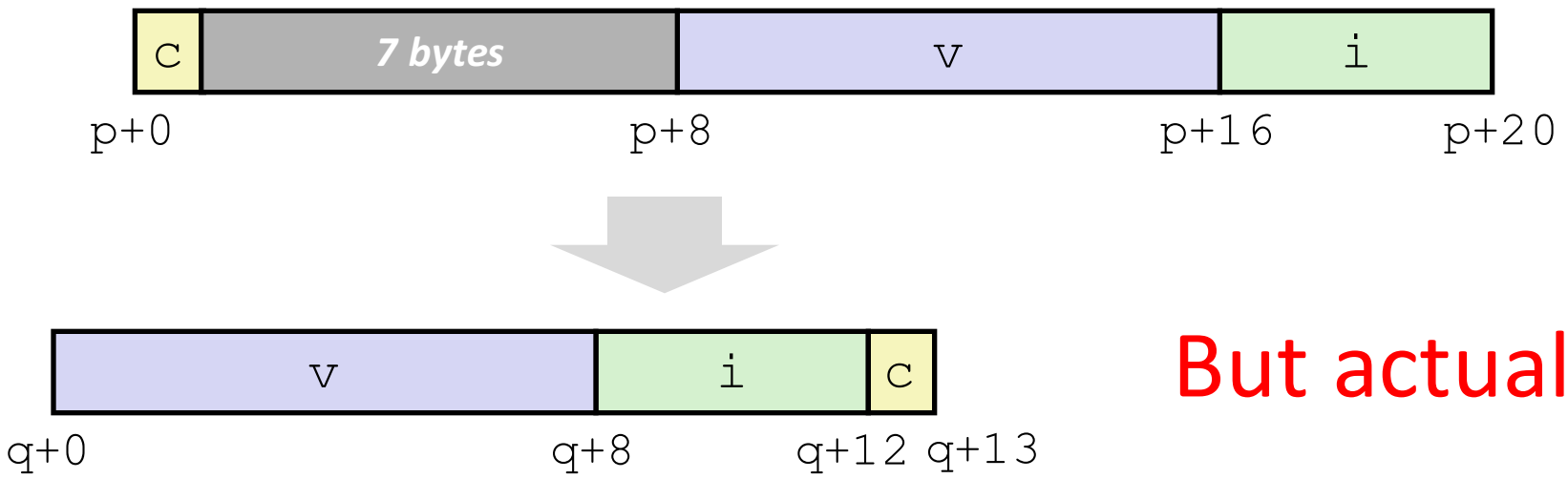


Struct packing saves space.

Put large data types first:

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

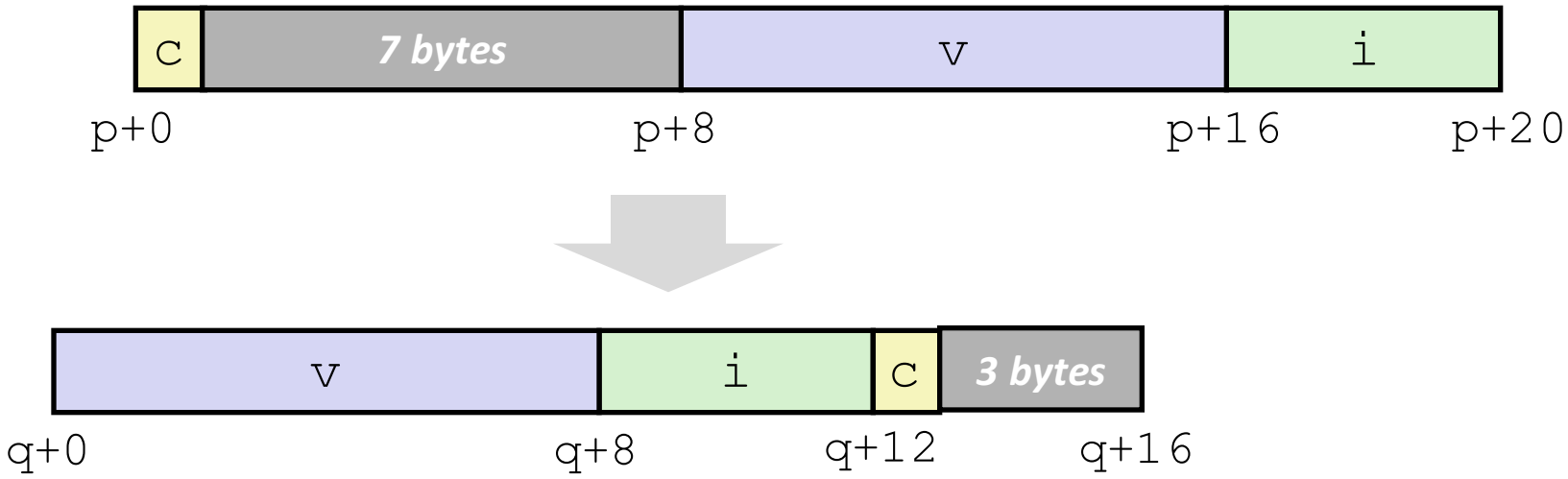
```
struct S2 {
  double v;
  int i;
  char c;
} * q;
```



But actually…

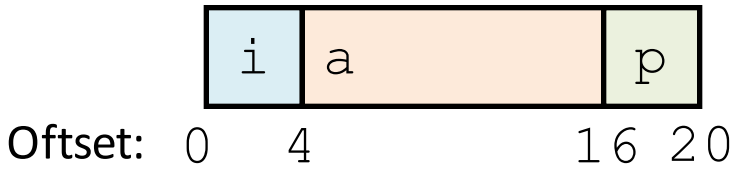
Struct alignment (full)

- Struct base address *and struct size* must align to size of largest internal primitive type.
- Each struct field's offset must be aligned to the size of the field type's largest alignment requirement.



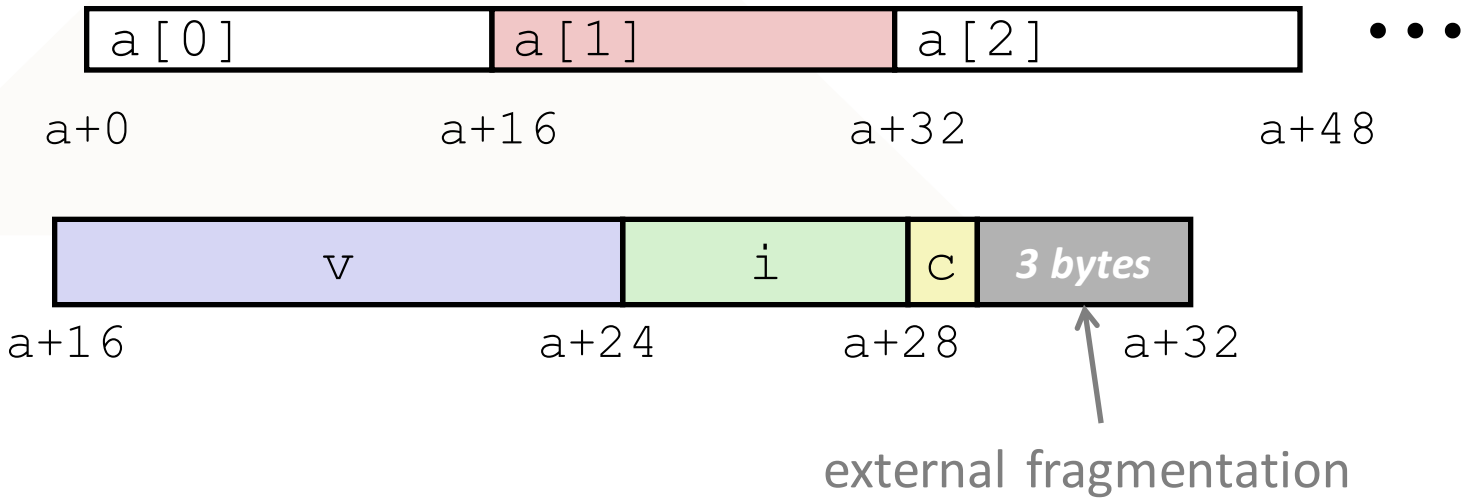
Array in struct

```
struct rec {
  int i;
  int a[3];
  int* p;
};
```

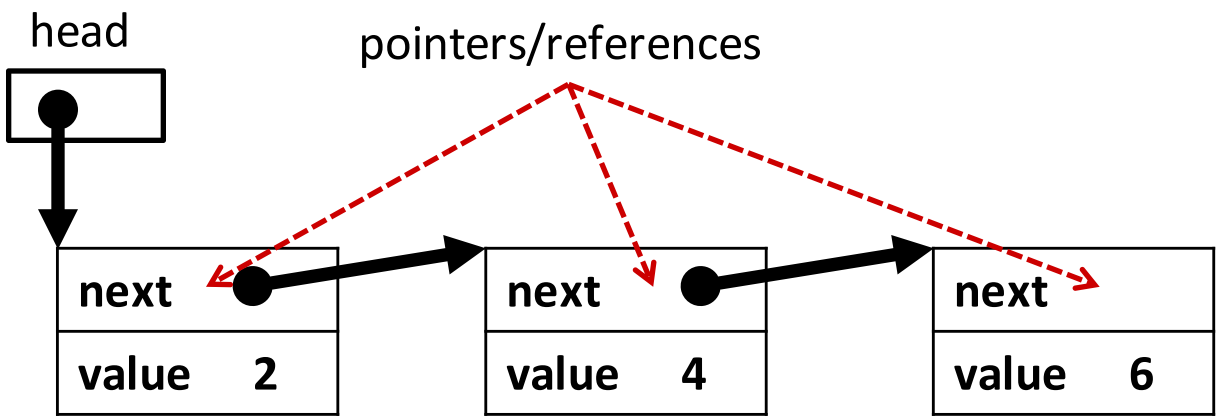


Struct in array

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



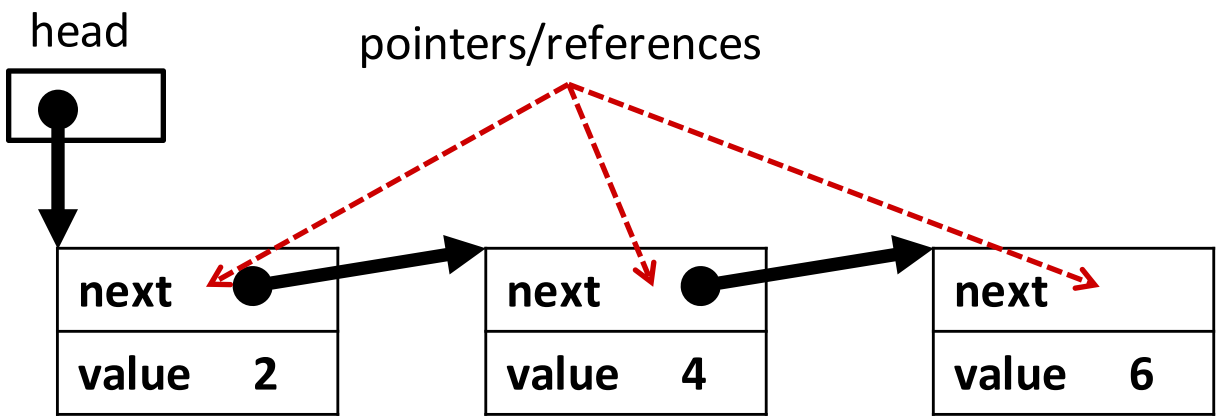
Linked Lists




```
typedef
struct ListNode {
    ListNode* next;
    int value;
} ListNode;
```

1. Decide on memory layout for all ListNodes.

Linked Lists




```
typedef
struct ListNode {
    ListNode* next;
    int value;
} ListNode;
```

2. Implement add_at_end:

```
void add_at_end(ListNode* head, int x) {
    ListNode* cursor = head;
    while (cursor->next != null) {
      cursor = cursor->next;
    }
    ListNode* n = (ListNode*)malloc(sizeof(ListNode));
    cursor->next = n;
    n->next = NULL;
    n->value = x;
}
```

Try a recursive version too.