Representing Data Structures

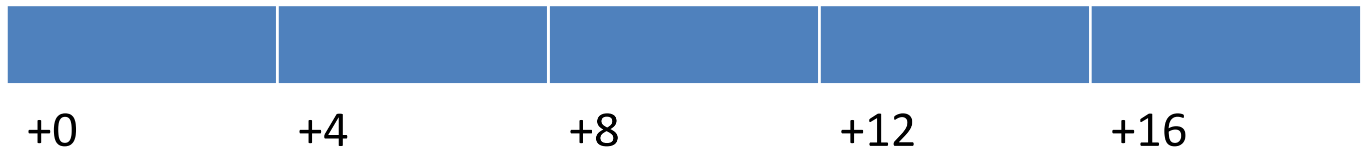
Multidimensional arrays

C structs

C: Array layout and indexing

ex

`int val[5];`



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

1. Assume:
   - Base address of val is in %rdi
   - i is in %rsi

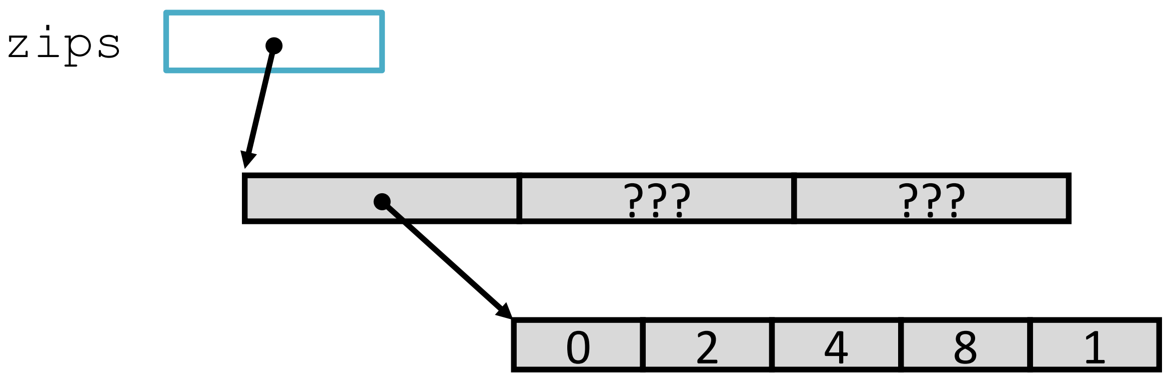
2. Assume:
   - Base address of val is 28(%rsp)
   - i is in %rcx

C: Arrays of pointers to arrays of …

reminder

C

```
int** zips = (int**)malloc(sizeof(int*)*3);
...
zips[0] = (int*)malloc(sizeof(int)*5);
...
int* zip0 = zips[0];
zip0[0] = 0;
zips[0][1] = 2;
zips[0][2] = 4;
zips[0][3] = 8;
zips[0][4] = 1;
```



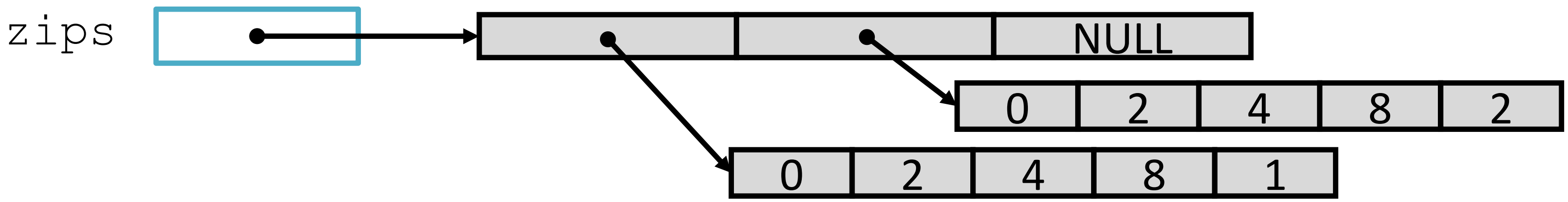
Java

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

C: Translate to x86

ex

```
void copyleft(int** zips, long i, long j) {
  zips[i][j] = zips[i][j - 1];
}
```



C: Row-major nested arrays

$$\begin{bmatrix} \texttt{a[0][0]} & \cdots & \texttt{a[0][C-1]} \\ \vdots & & \vdots \\ \texttt{a[R-1][0]} & \cdots & \texttt{a[R-1][C-1]} \end{bmatrix}$$

`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]

Suppose `a`'s base address is *A*.

`&a[i][j]` is *A + C×sizeof(int)×i + sizeof(int)×j*

(regular unscaled arithmetic)

```
int* b = (int*)a;  // Treat as larger 1D array

&a[i][j] == &b[ C*i + j ]
```

C: Strange array indexing examples

ex

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

9	8	1	9	5	9	8	1	0	5	9	8	1	0	3	9	8	1	1	5

76 96 116 136 156

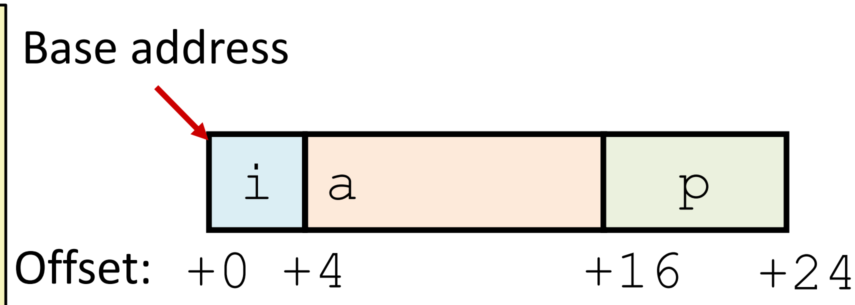
Reference	Address	Value
`sea[3][3]`	`76+20*3+4*3 = 148`	`1`
`sea[2][5]`		
`sea[2][-1]`		
`sea[4][-1]`		
`sea[0][19]`		
`sea[0][-1]`		

C does not do any bounds checking.

Row-major array layout is guaranteed.

C structs

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

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++
```

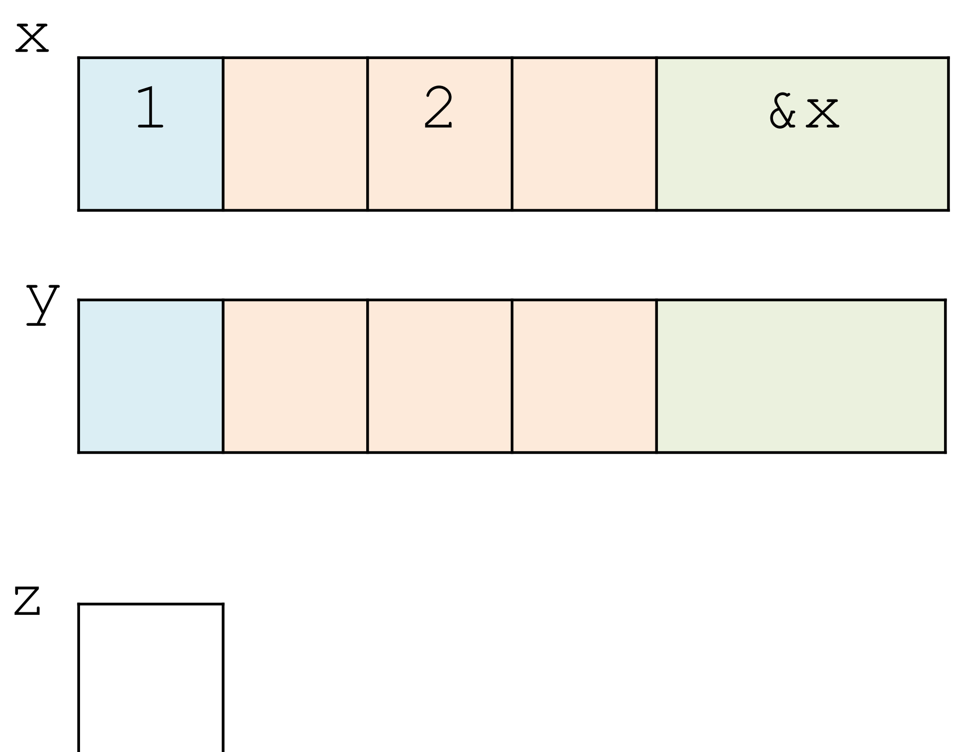


Memory Layout

Like Java class/object without methods.

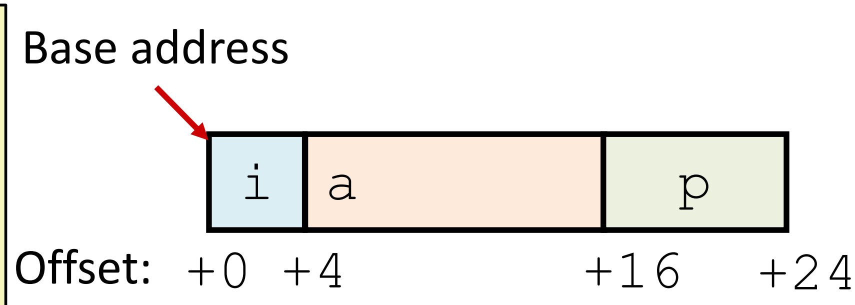
Compiler determines:

- Total size
- Offset of each field



C structs

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

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++
```

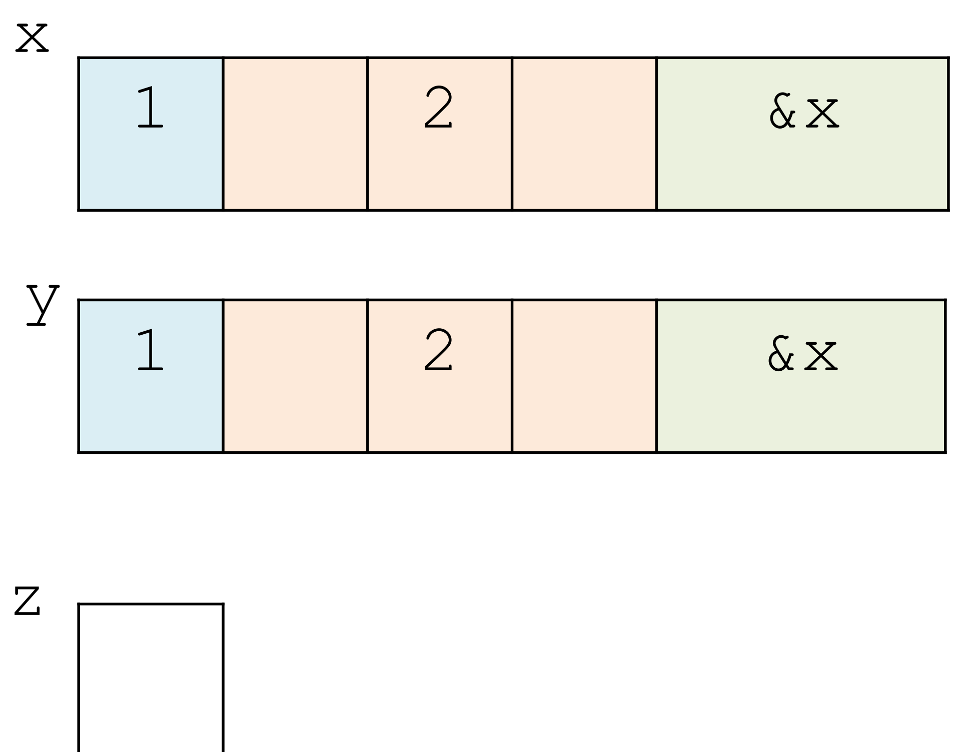


Memory Layout

Like Java class/object without methods.

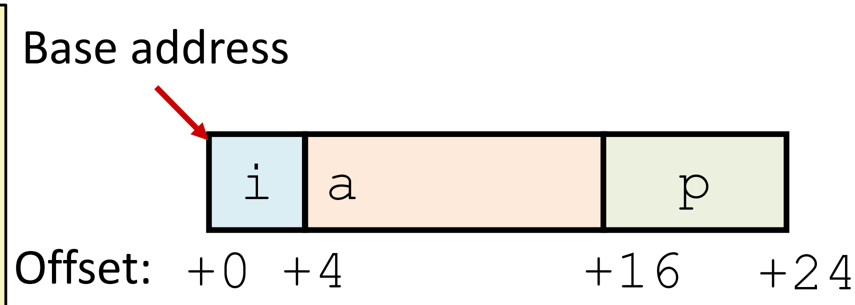
Compiler determines:

- Total size
- Offset of each field



C structs

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

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++
```

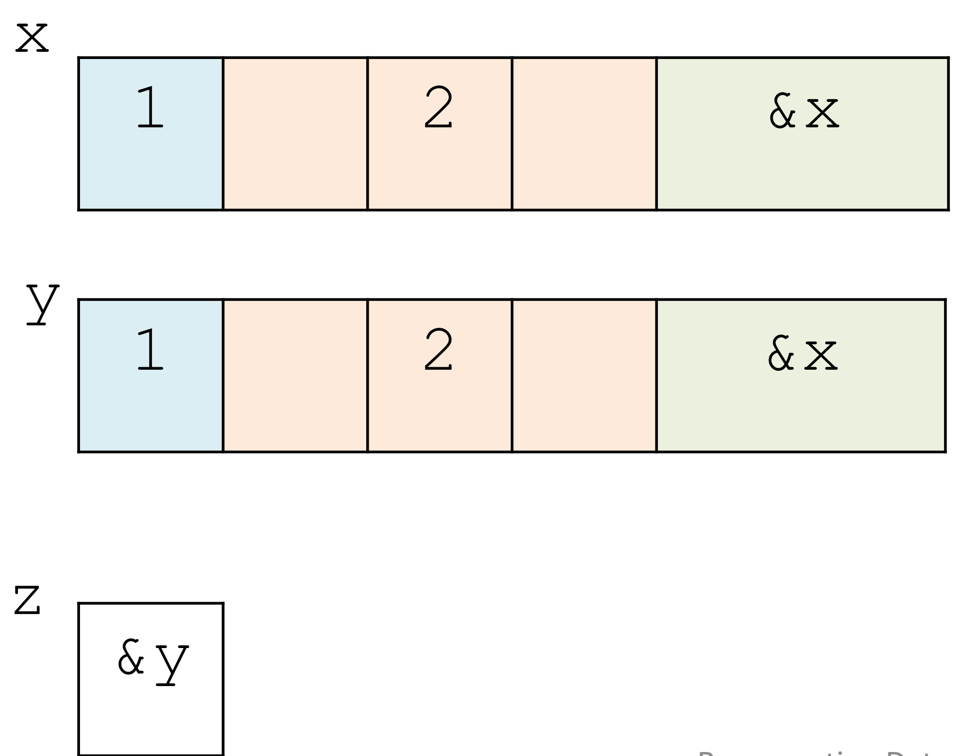


Memory Layout

Like Java class/object without methods.

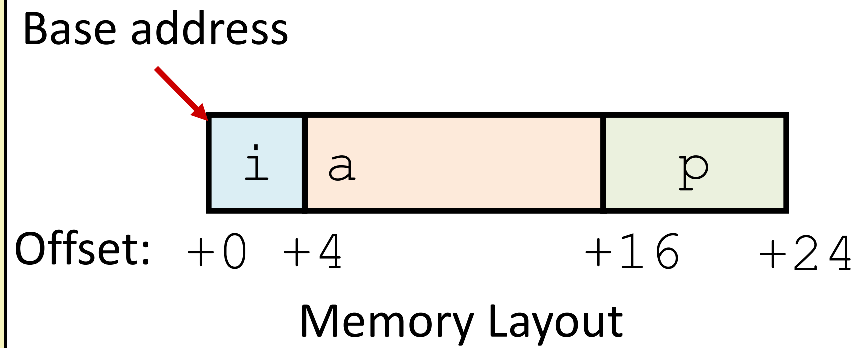
Compiler determines:

- Total size
- Offset of each field



C structs

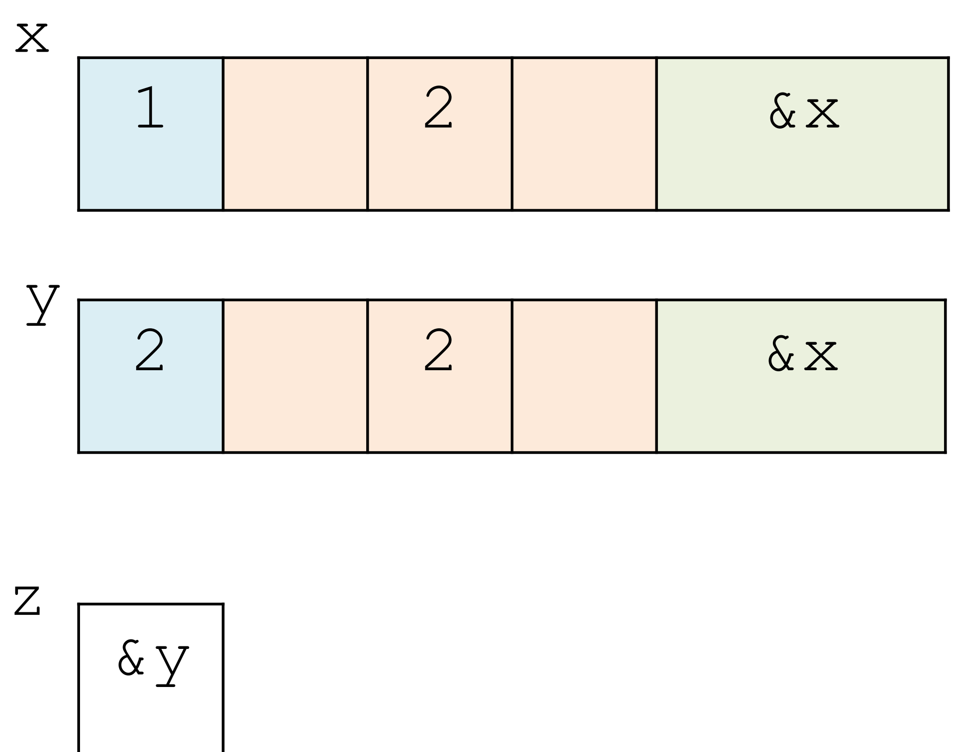
```
struct rec {
  int i;
  int a[3];
  int* p;
};

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++
```



Like Java class/object without methods.

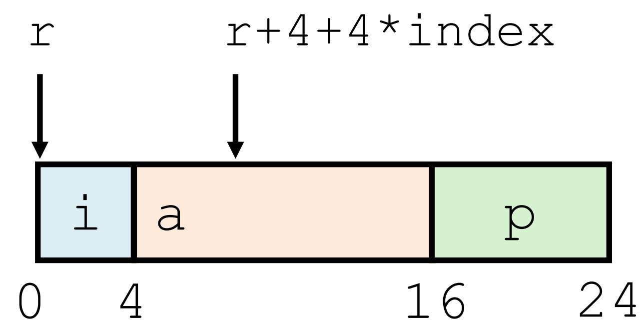
Compiler determines:

- Total size
- Offset of each field



C: 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];
}
```

```
movl 0(%rdi),%eax         # Mem[r+0]
addl 4(%rdi,%rsi,4),%eax    # Mem[r+4*index+4]
retq
```

C: Struct field alignment

Unaligned Data


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

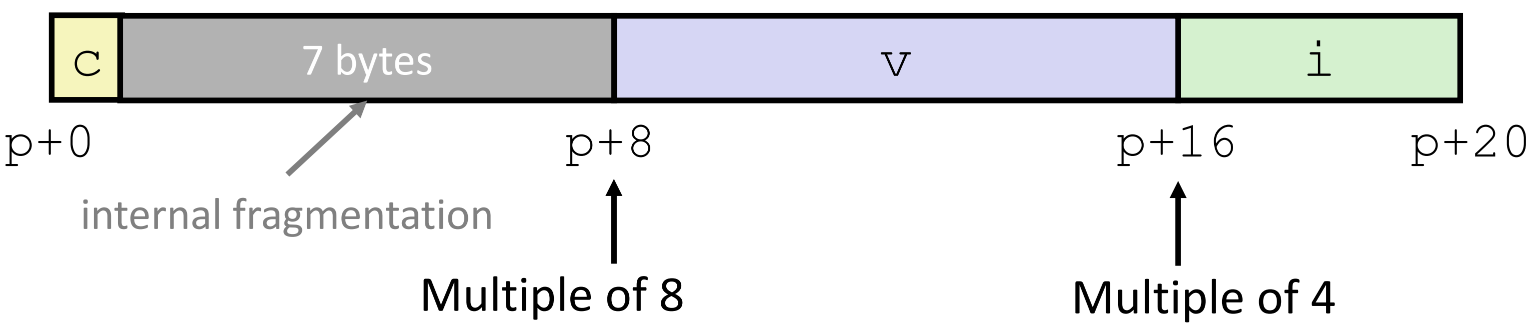
Defines new struct type and declares variable `p` of type `struct S1*`

Aligned Data

Primitive data type requires K bytes

Address must be multiple of K

C: align every struct field accordingly.



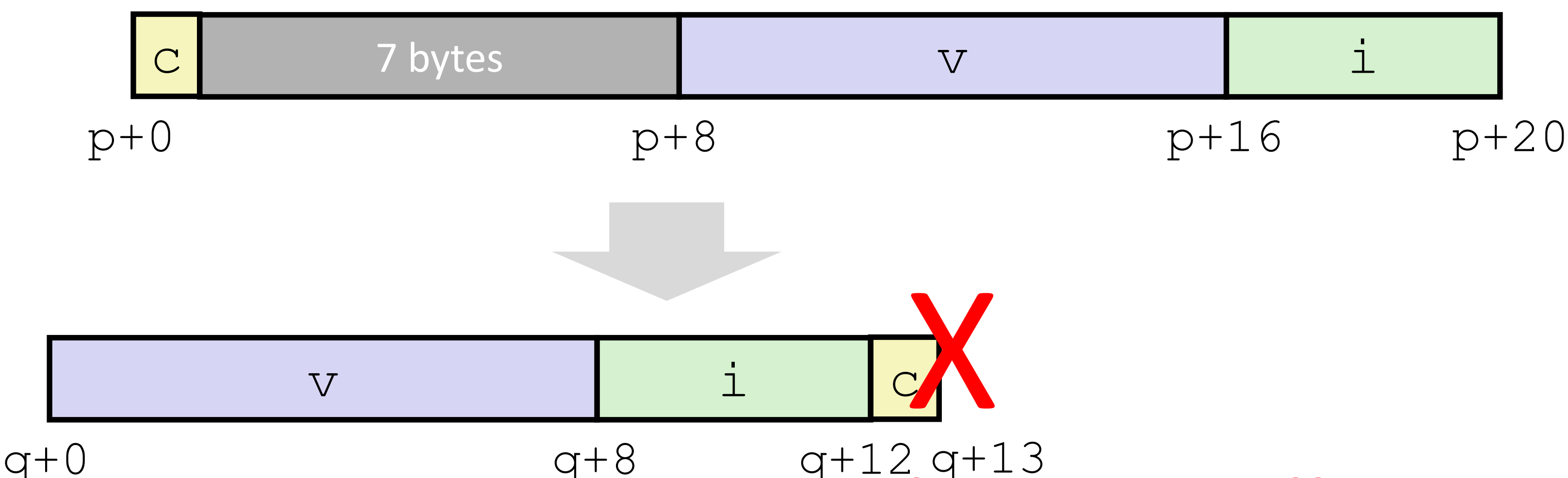
C: Struct packing

Put large data types first:

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

programmer

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



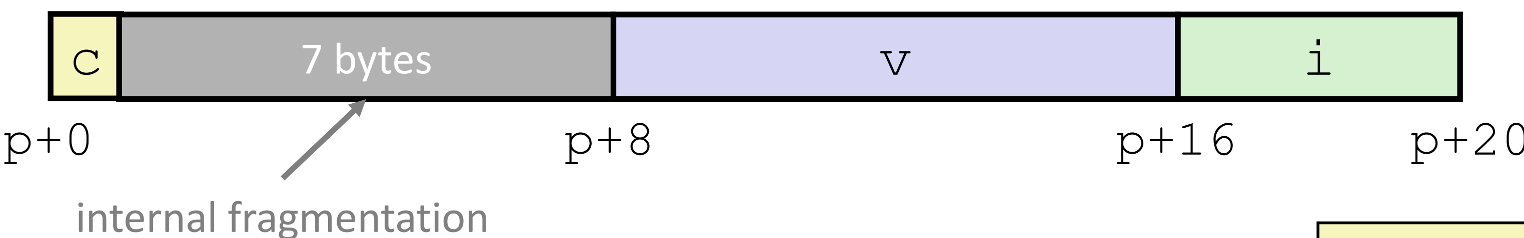
but actually…

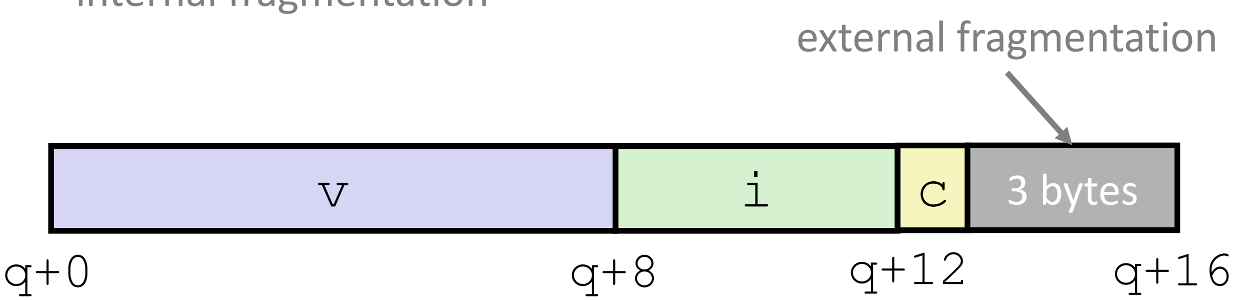
C: Struct alignment (full)

Base *and total size* must align largest internal primitive type.

Fields must align their type's largest alignment requirement.

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

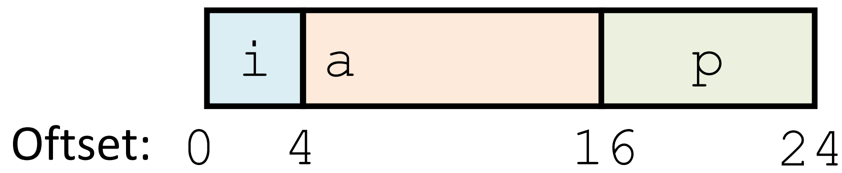





```
struct S2 {
  double v;
  int i;
  char c;
} * q;
```

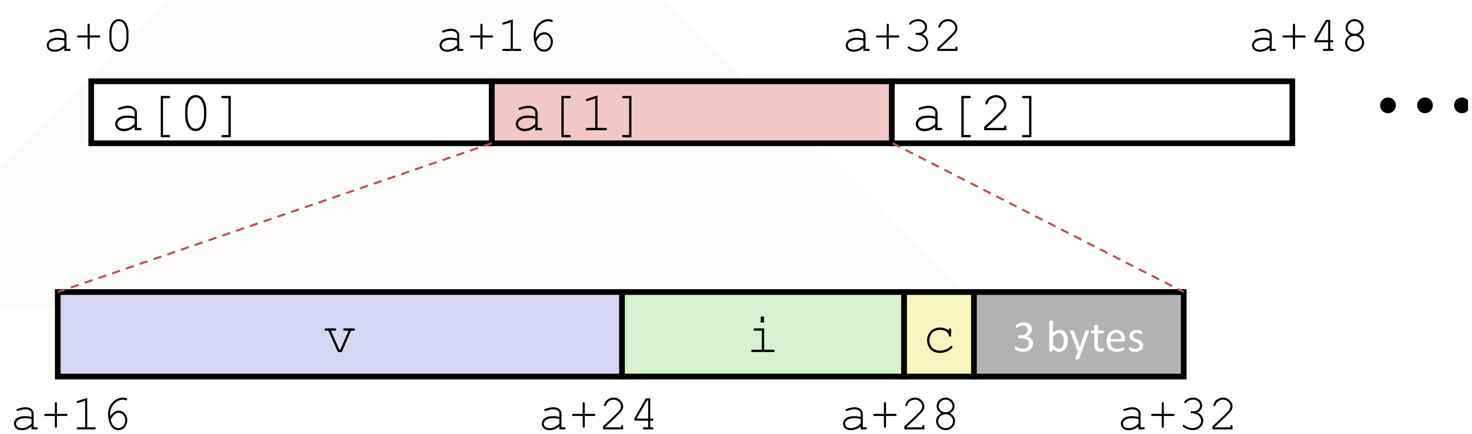
Array in struct

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



Struct in array

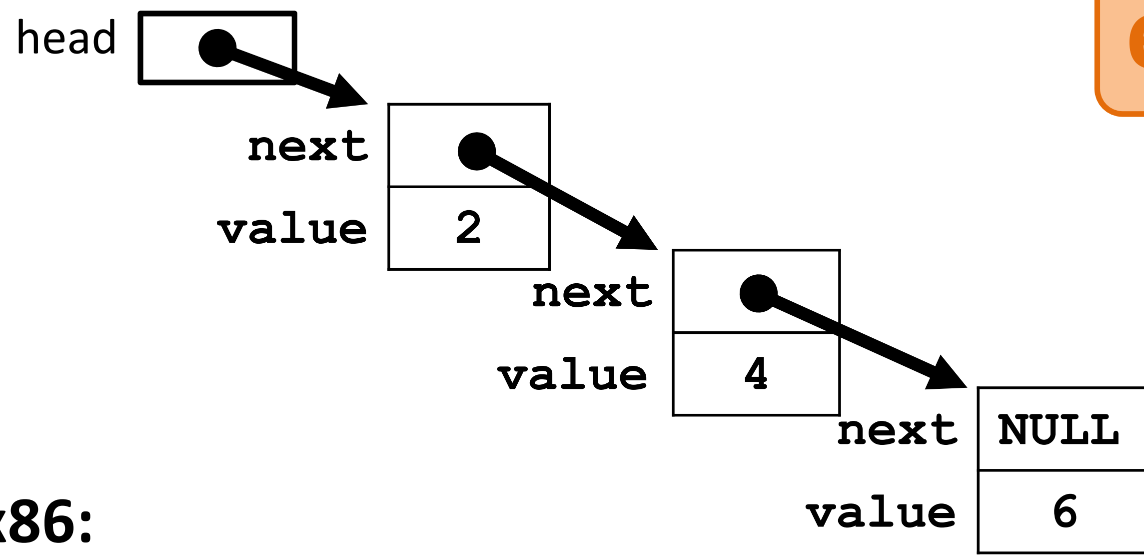
```
struct S2 {
  double v;
  int i;
  char c;
} a[10];
```



C: `typedef`

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

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

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

Linked Lists



ex

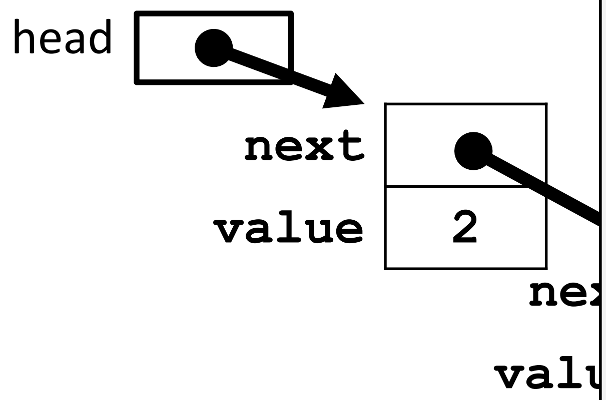
```
typedef
struct Node {
  struct Node* next;
  int value;
} Node;
```

Implement append in x86:

```
void append(Node* head, int x) {
  // assume head != NULL
  Node* cursor = head;
  // find tail
  while (cursor->next != NULL) {
    cursor = cursor->next;
  }
  Node* n = (Node*)malloc(sizeof(Node));
  // error checking omitted
  // for x86 simplicity
  cursor->next = n;
  n->next = NULL;
  n->value = x;
}
```

Try a recursive version too.

Linked Lists




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

Implement append in x86:

```
void append(Node* head, int x) {
  // assume head != NULL
  Node* cursor = head;
  // find tail
  while (cursor->next != NULL) {
    cursor = cursor->next;
  }
  Node* n = (Node*)malloc(sizeof(Node));
  // error checking omitted
  // for x86 simplicity
  cursor->next = n;
  n->next = NULL;
  n->value = x;
}
```

Try a recursive version too.

```
append:
  pushq  %rbp
  movl   %esi, %ebp
  pushq  %rbx
  movq   %rdi, %rbx
  subq   $8, %rsp
  jmp    .L3
.L6:
  movq   %rax, %rbx
.L3:
  movq   (%rbx), %rax
  testq  %rax, %rax
  jne    .L6
  movl   $16, %edi
  call   malloc
  movq   %rax, (%rbx)
  movq   $0, (%rax)
  movl   %ebp, 8(%rax)
  addq   $8, %rsp
  popq   %rbx
  popq   %rbp
  ret
```