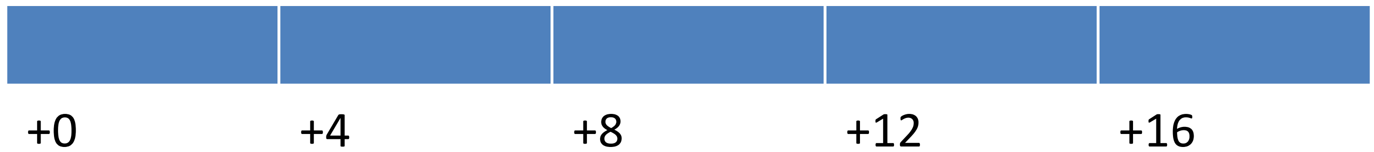
Representing Data Structures

Multidimensional arrays

Structs

Array Layout and Indexing

int val[5];



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

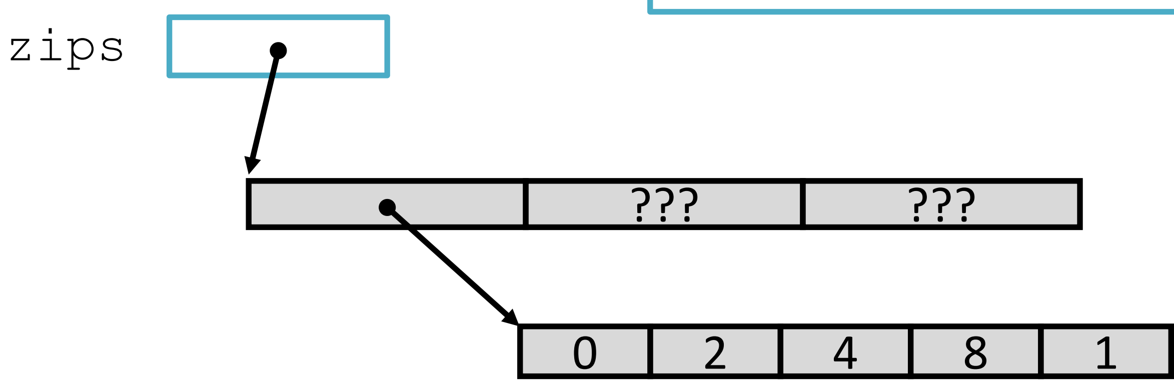
1. Assume:
    - Base address of val **is in** %rdi
    - i is in %rsi
2. Assume:
    - Base address of val **is** 28(%rsp)
    - i is in %rcx

C: Arrays of pointers to arrays of …

```
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;
```

Write x86 code to implement:

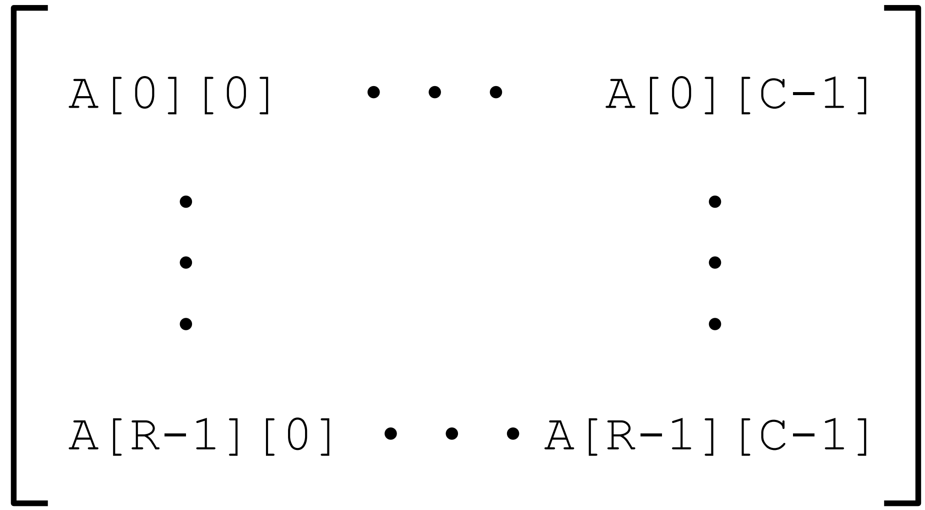
```
void copyleft(int** zips, long i, long j){
  zips[i][j] = zips[i][j - 1];
}
```



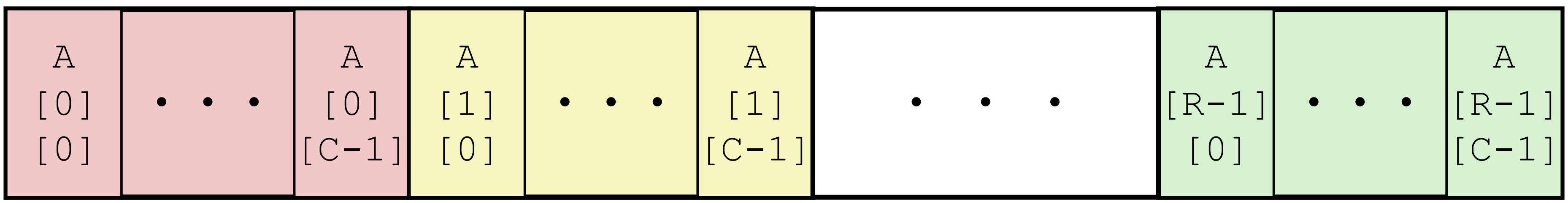

```
int[][] zips = new int[3][];
zips[0] = new int[5] {0, 2, 4, 8, 1};
```

Java

Row-Major Nested Arrays




```
int a[R][C];
```

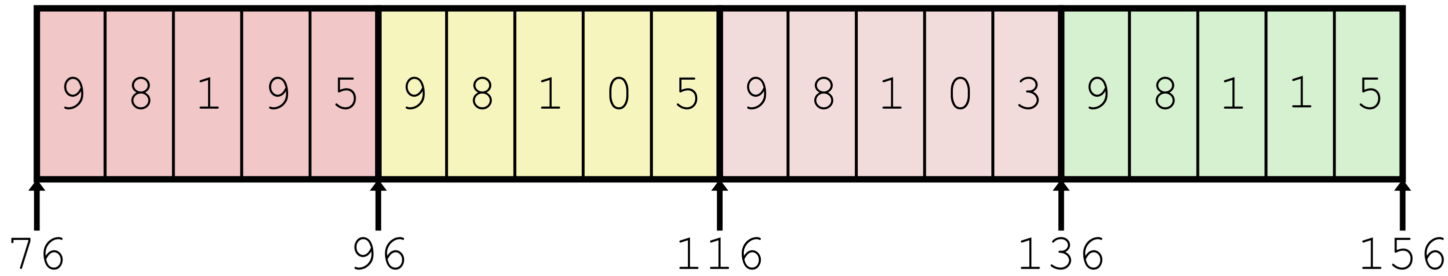


`&a[i][j]` is **`a +`** ________________________

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

&a[i][j] == &b[________________________________]
```

Strange Referencing Examples

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



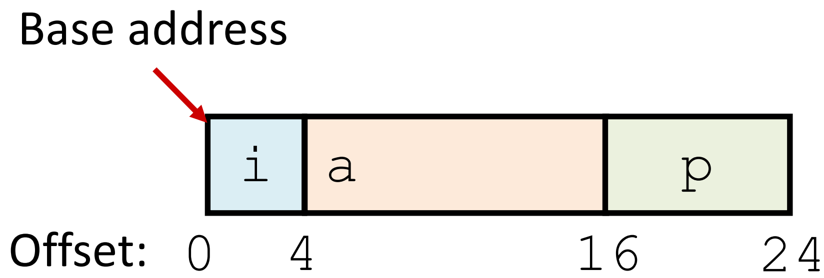
Reference	Address	Value	Guaranteed?
`sea[3][3]`	76+20*3+4*3 = 148	1	Yes
`sea[2][5]`			
`sea[2][-1]`			
`sea[4][-1]`			
`sea[0][19]`			
`sea[0][-1]`			

C does not do any bounds checking.

Row-major array layout is guaranteed.

C structs

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



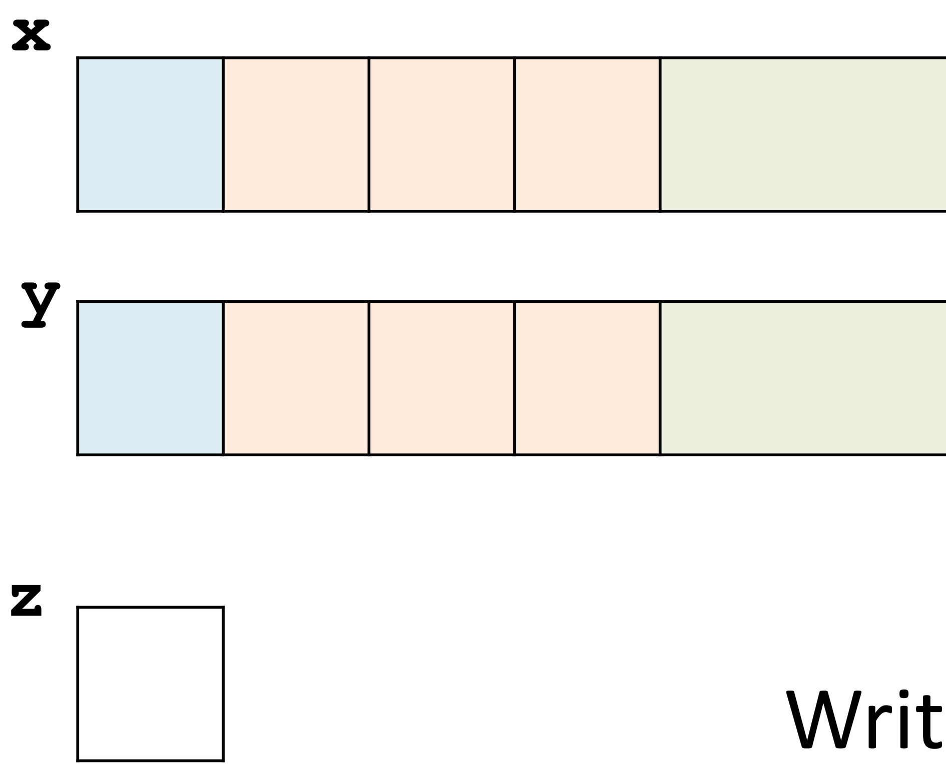
Memory Layout

Like Java class/object without methods.

Compiler determines:

- Total size
- Offset of each field

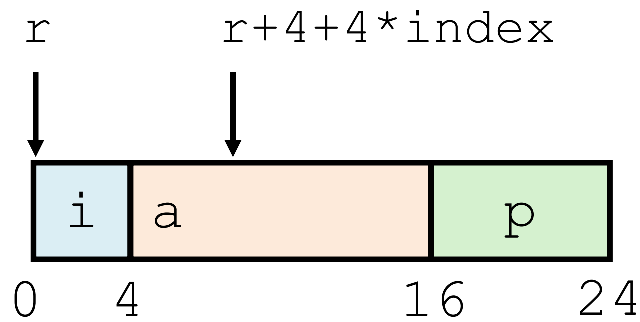
```
struct rec x;
struct rec y;
x.i = 1;
x.a[1] = 2;
x.p = &(x.i);
// copy full struct:
y = x;

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];
}
```

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

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;
```

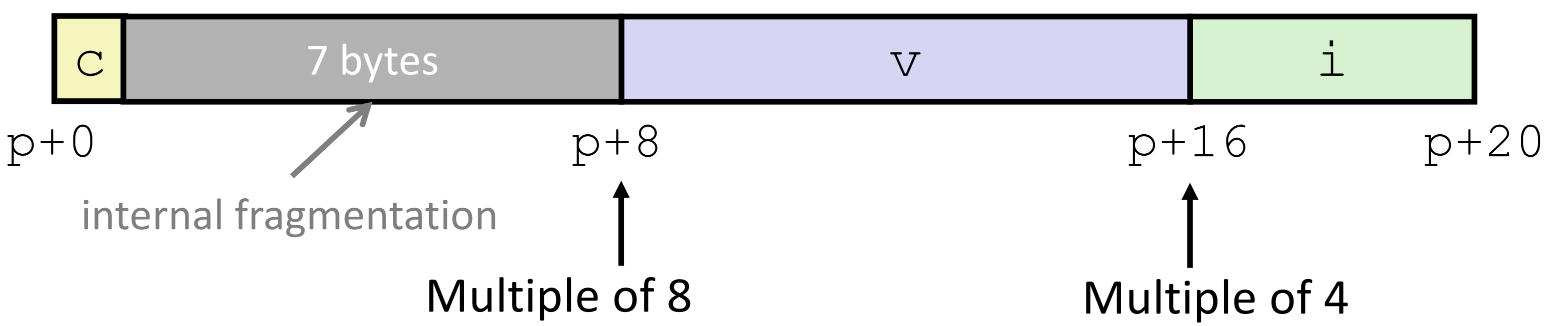
Defines new struct type and declares variable p of type struct S1*

Aligned Data

Primitive data type requires K bytes

Address must be multiple of K

C: align every struct field accordingly.



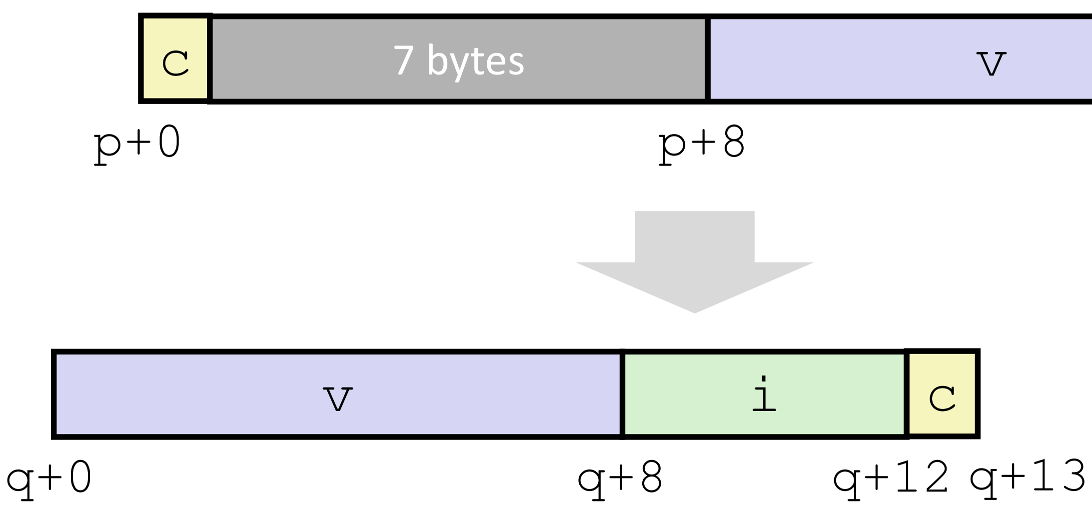
Struct packing

Put large data types first:

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

programmer

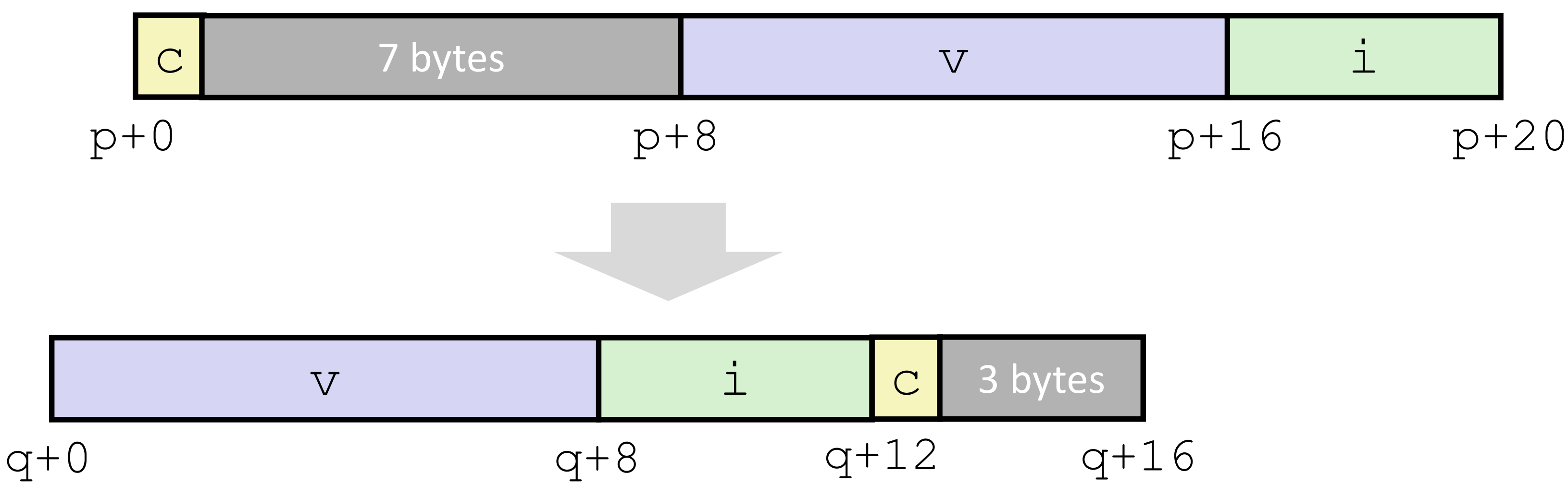
```
struct S2 {
   double v;
   int i;
   char c;
} * q;
```



But actually…

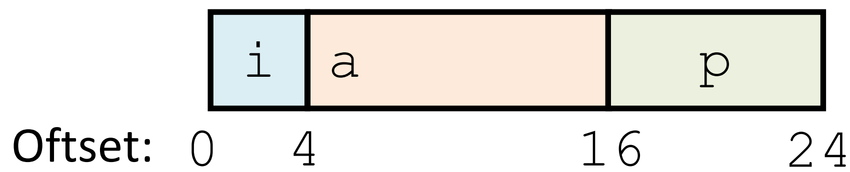
Struct alignment (full)

- Base *and total size* must align largest internal primitive type.
- Fields must align their type's largest alignment requirement.



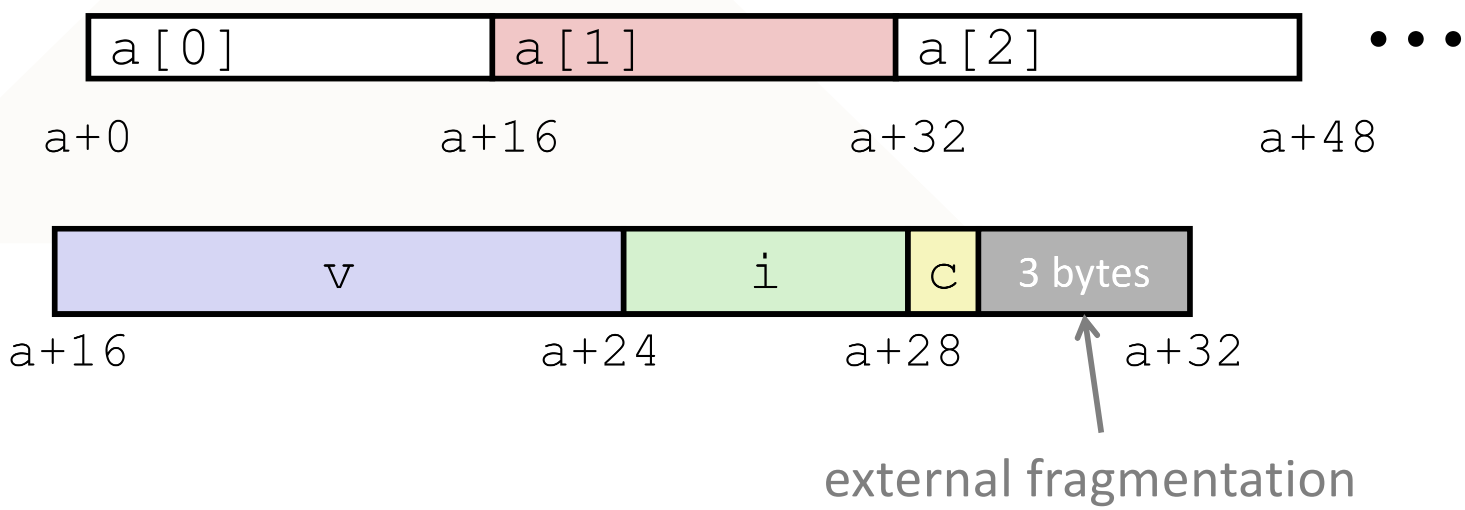
Array in struct

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



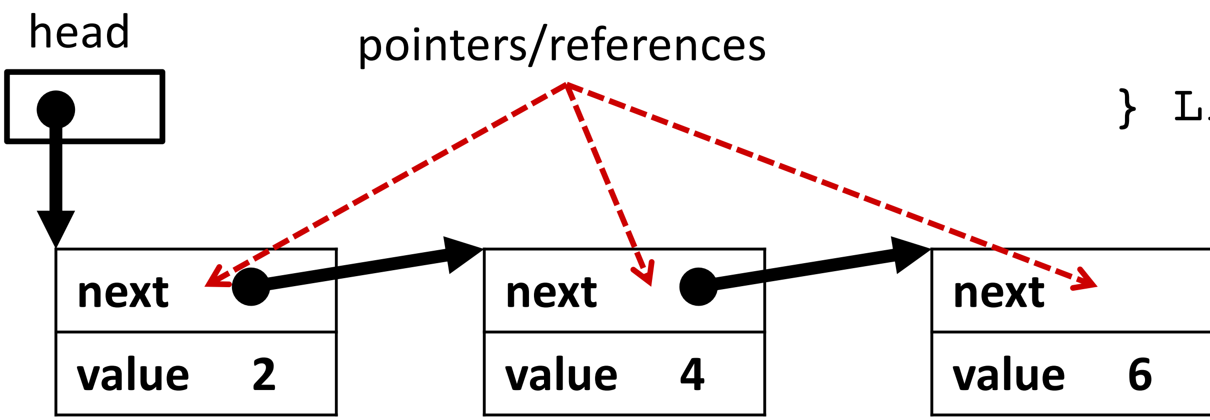
Struct in array

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



Linked Lists

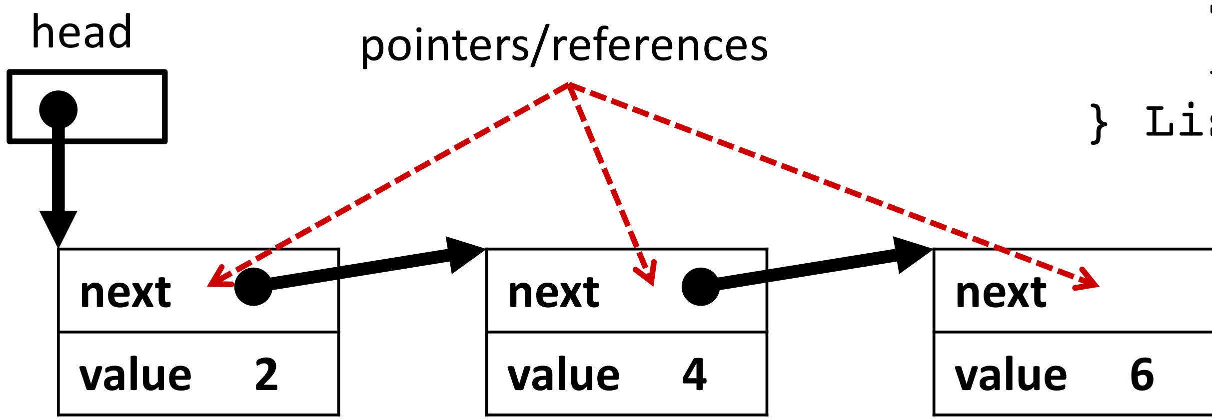
```
typedef
struct ListNode {
    struct ListNode* next;
    int value;
} ListNode;
```



1. Choose memory layout for ListNode

Linked Lists

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



2. Implement append in x86:

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

Try a recursive version too.