Representing Data Structures

Multidimensional arrays
Structs
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 ...

```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;
```

Write x86 code to implement:
```c
void copyleft(int** zips, long i, long j){
    zips[i][j] = zips[i][j - 1];
}
```

```java
int[][] zips = new int[3][];
zips[0] = new int[5] {0, 2, 4, 8, 1};
```
Row-Major Nested Arrays

```c
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];

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>sea[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[2][−1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[4][−1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[0][19]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[0][−1]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C does not do any bounds checking.
Row-major array layout is guaranteed.
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++
### Accessing Struct Field

**C Code**

```c
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];
}
```

```assembly
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

Aligned Data

Primitive data type requires $K$ bytes
Address must be multiple of $K$
C: align every struct field accordingly.

internal fragmentation

Multiple of 8
Multiple of 4

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

Defines new struct type and declares variable $p$ of type struct S1*
Struct packing

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)

- 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**

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

1. Choose memory layout for ListNode
Linked Lists

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

2. Implement append in x86:

```c
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.