Representing Data Structures

Multidimensional arrays
C structs
C: Array layout and indexing

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

Write x86 code to load \texttt{val[i]} into \%eax.

1. Assume:
   - Base address of \texttt{val} is in \%rdi
   - \texttt{i} is in \%rsi

```
movl (\%rdi, \%rsi, 4), \%eax
```

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

```
movl 28(\%rsp, \%rcx, 4), \%eax
```
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;
C: Translate to x86

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

```
C: Translate to x86

ex
```
C: Row-major nested arrays

int a[R][C];

Suppose a's base address is A.
&a[i][j] is $A + C \times \text{sizeof}(\text{int}) \times i + \text{sizeof}(\text{int}) \times j$
(regular unscaled arithmetic)

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

&a[i][j] == &b[ C*i + j ]
C: Strange array indexing examples

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

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sea[3][3]</td>
<td>$76+20\times3+4\times3 = 148$</td>
<td>1</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[0][19]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[0][-1]</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++
C: Accessing struct field

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

```c
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
```
C: Struct field alignment

Unaligned Data

Aligned Data

Primitive data type requires $K$ bytes
Address must be multiple of $K$

C: align every struct field accordingly.

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

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

internal fragmentation

Multiple of 8
Multiple of 4

Representing Data Structures
C: Struct packing

Put large data types first:

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

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

but actually...

Representing Data Structures
C: Struct alignment (full)

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

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

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

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

Struct in array

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

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
class Node {
    struct Node* next;
    int value;
} Node;

Implement append in x86:

```c
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.