Representing Data Structures

Multidimensional arrays
C structs

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;
```

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

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

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

C: Translate to x86

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

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

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

```
```
```
C: Row-major nested arrays

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

Suppose a's base address is A.

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

```
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<em>3+4</em>3 = 148</td>
<td>1</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>9</td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>76+20<em>2+4</em>(5-1) = 112</td>
<td>5</td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>76+20<em>4+4</em>(5-1) = 152</td>
<td>5</td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>5</td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td>76+20<em>0+4</em>(19-1) = 72</td>
<td>96</td>
</tr>
</tbody>
</table>

C does not do any bounds checking.
Row-major array layout is guaranteed.

C structs

Like Java class/object without methods.

Compiler determines:
- Total size
- Offset of each field

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

Base address

```
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 structs

Like Java class/object without methods.

Compiler determines:
- Total size
- Offset of each field

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

Base address

```
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++
```
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++]

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

C: Accessing struct field

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.
C: Struct packing

Put large data types first:

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

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

But actually...

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

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

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

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

Struct in array

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

**typedef**

```c
struct 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.

### Linked Lists

**typedef**

```c
struct 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.