Representing Data Structures

Multidimensional arrays

C structs

C: 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;

C: Translate to x86

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

zips

Java

int[][] zips = new int[3][];
zips[0] = new int[5] {0, 2, 4, 8, 1};
C: Row-major nested arrays

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

Suppose a's base address is $A$.

$$\&a[i][j] = A + C \times sizeof(int) \times i + sizeof(int) \times j$$

(regular unscaled arithmetic)

```c
int* b = (int*)a; // Treat as larger 1D array
&a[i][j] == &b[C*i + j]
```

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

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<em>3+4</em>3 = 148</td>
<td>1</td>
</tr>
<tr>
<td>sea{2}[5]</td>
<td>96+20<em>2+4</em>5 = 136</td>
<td>9</td>
</tr>
<tr>
<td>sea{2}[-1]</td>
<td>96+20<em>2+4</em>(-1) = 112</td>
<td>8</td>
</tr>
<tr>
<td>sea{4}[-1]</td>
<td>76+20<em>4+4</em>(-1) = 152</td>
<td>5</td>
</tr>
<tr>
<td>sea{0}[19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>9</td>
</tr>
<tr>
<td>sea{0}[-1]</td>
<td>76+20<em>0+4</em>(-1) = 72</td>
<td>8</td>
</tr>
</tbody>
</table>

C: Accessing struct field

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

Like Java class/object without methods.

Compiler determines:
• Total size
• Offset of each field

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

C structs

<table>
<thead>
<tr>
<th>Offset</th>
<th>Memory Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>i</td>
</tr>
<tr>
<td>+4</td>
<td>a</td>
</tr>
<tr>
<td>+16</td>
<td>p</td>
</tr>
</tbody>
</table>

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

```c
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.

C: Struct packing
Put large data types first:

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

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

C: 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

Offset: 0 4 16 24

Struct in array

But actually...

Oftset: 0 4 16 24

A partition in array

A partition in array

A partition in array

A partition in array
C: typedef

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

// struct types can be verbose
struct Node {
    ...
} Node;

// typedef can help
typedef struct Node {
    ...
} Node;

Node* n = ...

LinkedLists

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

Node* n = ...

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.