Representing Data Structures

Multidimensional arrays
C structs

C: Array layout and indexing

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

1. Assume:
   - Base address of `val` is in `%rdi`
   - `i` is in `%rsi`

   ```
   movl (%rdi, %rsi, 4), %eax
   movl 28(%rsp, %rcx, 4), %eax
   ```

2. Assume:
   - Base address of `val` is `28(%rsp)`
   - `i` is in `%rcx`

   ```
   movq (%rdi,%rsi,8), %rax # %rax ← zips[i]
   movl -4(%rax,%rdx,4), %ecx # %ecx ← %rax[j - 1]
   movl %ecx, (%rax,%rdx,4) # %rax[j] ← %ecx
   retq
   ```

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;
zip0[1] = 2;
zip0[2] = 4;
zip0[3] = 8;
zip0[4] = 1;
```

Java

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

C: Translate to x86

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

```c
copyleft(zips, 1, 3)
```
C: Row-major nested arrays

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

Suppose a's base address is A.

```
&a[i][j] is 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];
```

Reference | Address | Value
---|---|---
sea[3][3] | 76+20*3+4*3 = 148 | 1
sea[2][5] | 76+20*2+4*5 = 136 | 9
sea[2][-1] | 76+20*2+4*-1 = 112 | 5
sea[4][-1] | 76+20*4+4*-1 = 152 | 5
sea[0][19] | 76+20*0+4*19 = 152 | 5
sea[0][-1] | 76+20*0+4*-1 = 72 | ??

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

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>+0</td>
</tr>
<tr>
<td>a[0]</td>
<td>+4</td>
</tr>
<tr>
<td>a[1]</td>
<td>+16</td>
</tr>
<tr>
<td>a[2]</td>
<td>+24</td>
</tr>
</tbody>
</table>
```

Memory Layout

```
x = 1

y = 2

&x
```

```
y = 6

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

```
<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>+0</td>
</tr>
<tr>
<td>a[0]</td>
<td>+4</td>
</tr>
<tr>
<td>a[1]</td>
<td>+16</td>
</tr>
<tr>
<td>a[2]</td>
<td>+24</td>
</tr>
</tbody>
</table>
```

Memory Layout

```
x = 1

y = 2

&x
```

```
y = 6

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;
};

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

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

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;
```

```c
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.
```

```
C: 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;
```

Try a recursive version too.

```
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;
}
```

Implement append in x86:

```
append:
    pushq %rbp
    movl %esi, %ebp
    pushq %rbx
    movq %rdi, %rbx
    subq $8, %rsp
    jmp .L3
.L6:
    movq (%rax), %rbx
.L3:
    movq %rax, (%rbx)
    movq $0, (%rax)
    movl %ebp, 8(%rax)
    addq $8, %rsp
    popq %rbx
    popq %rbp
    ret
```

Try a recursive version too.