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 ...

```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: Translate to x86

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

Representing Data Structures
C: Row-major nested arrays

\[
\begin{bmatrix}
    a[0][0] & \cdots & a[0][C-1] \\
    \vdots & \ddots & \vdots \\
    a[R-1][0] & \cdots & a[R-1][C-1]
\end{bmatrix}
\]

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

\[
\text{int* } b = (\text{int*})a; \quad // \text{Treat as larger 1D array}
\]

\&a[i][j] == \&b[C\times 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<em>3+4</em>3 = 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.
```c
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 structs**

Like Java class/object without methods.

Compiler determines:
- Total size
- Offset of each field

### Memory Layout

<table>
<thead>
<tr>
<th>Offset</th>
<th>Base address</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>

---

Representing Data Structures  7
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++

Like Java class/object without methods.

Compiler determines:
• Total size
• Offset of each field

C structs

Representing Data Structures
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 structs

Like Java class/object without methods.

Compiler determines:
• Total size
• Offset of each field

Offset:

Base address

Memory Layout

C structs

Representing Data Structures
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];
}

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

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

Aligned Data

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

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

internal fragmentation
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...
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;
```

Representing Data Structures
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];
```
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 = ...;
typedef 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

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

```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;
}
```

**Implement append in x86:**

```assembly
append:
    pushq %rbp
    movl %esi, %ebp
    pushq %rbx
    movq %rdi, %rbx
    subq $8, %rsp
    jmp .L3
.L6:
    movq %rax, %rbx
.L3:
    movq (%rbx), %rax
    testq %rax, %rax
    jne .L6
    movl $16, %edi
    call malloc
    movq %rax, (%rbx)
    movq $0, (%rax)
    movl %ebp, 8(%rax)
    addq $8, %rsp
    popq %rbx
    popq %rbp
    ret
```

Try a recursive version too.