Representing Data Structures

Multidimensional arrays
C structs

https://cs.wellesley.edu/~cs240/
C: Array layout and indexing

int val[5];

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+4</td>
<td>+8</td>
<td>+12</td>
<td>+16</td>
</tr>
</tbody>
</table>

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

```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];
}
```

copyleft(zips, 1, 3)

---

Representing Data Structures
C: Row-major nested arrays

int a[R][C];

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

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

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

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

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sea[3][3]</code></td>
<td>76 + 20<em>3 + 4</em>3</td>
<td>148, 1</td>
</tr>
<tr>
<td><code>sea[2][5]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sea[2][-1]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sea[4][-1]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sea[0][19]</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>sea[0][-1]</code></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

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

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
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;
};

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

![Diagram showing 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*

Representing Data Structures
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…

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.

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

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

Representing Data Structures 14
Array in struct

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

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

---

Struct in array

```
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 = ...;
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.
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.