Outline

Goal: understand how we represented structured data in C and x86

- Arrays in x86
- Array indexing
- Arrays of pointers to arrays
- 2-dimensional arrays
- C structs (simpler version of objects)
  - Overview and accessing fields
  - Alignment
  - LinkedList example

C: Array layout and indexing

Recall:
- Array layout will be contiguous block of memory
- The base address will be aligned based on the element type: here, a multiple of 4

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

```
int val[5];
+0 +4 +8 +12 +16
movl (%rdi, %rsi, 4), %eax
movl 28(%rsp, %rcx, 4), %eax
```

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;
```

```
int[] zips = new int[3][];
zips[0] = new int[5] {0, 2, 4, 8, 1};
```
### C: Arrays of pointers to arrays in x86

**Goal:** translate to x86, using two scratch registers

1. Put `zips[i]` in a reg
2. Access element `[j-1]`
3. Set element `[j]`
4. Return

```assembly
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
```

### Alternative: row-major nested arrays

Pros:
- Accessing nested elements now a single memory operation!
- Calculations can be done ahead of time, via arithmetic

Cons:
- Less space efficient depending on the shape of the data
- Need to be careful with our order of indexing!

### C: Arrays of pointers to arrays: Pros/Cons

**Pros:**
- Flexible array lengths
- Different elements can be different lengths
- Lengths can change as the program runs
- Representation of empty elements saves space

**Cons:**
- Accessing a nested element requires multiple memory operations

### C: Row-major nested arrays

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

Suppose `a`'s base address is `A`.

\[
&a[i][j] \text{ is } A + C \times \text{sizeof(int)} \times i + \text{sizeof(int)} \times j \\
\text{(regular unscaled arithmetic)}
\]

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

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

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sea[3][3]</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

int sea[4][5];

76 96 16 16 15
9 8 1 9 5 9 8 1 0 9 8 1 0 9 8 1 1 5

C structs

Like Java class/object, without methods.
Models structured, but not necessarily list-list, data.
Combines other, simpler types.

struct point {
  int xcoordinate;
  int ycoordinate;
};

struct student {
  int classyear;
  int id;
  char* name;
};

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;

Memory Layout

Base address

Offset: +0 +4 +16 +24

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;

Memory Layout

Base address

Offset: +0 +4 +16 +24

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;
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;
// copy full struct
y = x;
struct rec* z;
z = &y;

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;
// copy full struct
y = x;
struct rec* z;
z = &y;

// 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];
}

Example: traversing a list of structs

struct student {
    int classyear;
    int id;
    char* name;
};

// Given a null-terminated list of students,
// return the name of the student with a given ID, or null
// if there is no student with that ID.
char* getStudentNameWithId(struct student s[], int id) {
    int index;
    // ...
    return s[index].name;
}
C: Struct field alignment

Alignment is especially important for structs

Unaligned Data (not what C does)

<table>
<thead>
<tr>
<th>c</th>
<th>v</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+1</td>
<td>p+9</td>
<td>p+13</td>
</tr>
</tbody>
</table>

Aligned Data (what C does)

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

<table>
<thead>
<tr>
<th>c</th>
<th>v</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+8</td>
<td>p+16</td>
</tr>
</tbody>
</table>

multiple of 8

multiple of 4

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

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

C: Struct packing

Put large data types first:

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

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

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.

Array in struct

Struct in array
C: typedef

```c
// 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 = ...

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

Linked Lists

### 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;
}
```

Extra fun: try a recursive version too!