Representing Data Structures

Multidimensional arrays
C structs
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 (defer details to video)
- C structs (simpler version of objects)
  - Overview and accessing fields
  - Alignment
  - LinkedList example
C: Array layout and indexing

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 \texttt{28(\%rsp)}
   - \texttt{i} is in \%rcx

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

For: \texttt{T a[N]}

**Address of \texttt{a[i]} is:**

\texttt{a + i * sizeof(T)}
Which expression correctly loads val[i] into %rax? Assume val is in %rdi and i is in %rsi.

- `movq (%rsi, %rdi, 4), %rax` (0th option)
- `movq (%rdi, %rsi, 4), %rax` (1st option)
- `movq (%rdi, %rsi, 8), %rax` (2nd option)
- `movq (%rsi, %rdi, 8), %rax` (3rd option)
- None of the above (4th option)
Which expression correctly loads val[i] into %rax? Assume val is in %rdi and i is in %rsi.

- movq (%rsi,%rdi,4), %rax
- movq (%rdi,%rsi,4), %rax
- movq (%rdi,%rsi,8), %rax
- movq (%rsi,%rdi,8), %rax
- None of the above
Which expression correctly loads val[i] into %rax? Assume val is in %rdi and i is in %rsi.

long val[4];

- `movq (%rsi,%rdi,4), %rax` 0%
- `movq (%rdi,%rsi,4), %rax` 0%
- `movq (%rdi,%rsi,8), %rax` 0%
- `movq (%rsi,%rdi,8), %rax` 0%
- None of the above 0%
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: Arrays of pointers to arrays in x86

Goal: translate to x86, using two scratch registers
%rax, %ecx (why 32 bits?)

1. Put \texttt{zips[i]} in a reg
2. Access element \texttt{[j-1]}
3. Set element \texttt{[j]}
4. Return (nothing)
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
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!

Single contiguous block of memory
C: Row-major nested arrays

int a[R][C];

Suppose a's base address is $A$.

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

(regular unscaled arithmetic)

int* b = (int*)a;  // 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></td>
<td></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[2][4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[4][-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 structs

Like Java class/object, without methods.

Models structured, but not necessarily list-like, data.

Combines other, simpler types.

```c
struct point {
    int xcoordinate;
    int ycoordinate;
};

struct student {
    int classyear;
    int id;
    char* name;
};
```
C structs

like Java class/object without methods.

Compiler determines:
- Total size
- Offset of each field

```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);
```
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;
C structs

Like Java class/object without methods.

Compiler determines:
- Total size
- Offset of each field

```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;
```
C structs

Like Java class/object without methods.

Compiler determines:
- Total size
- Offset of each field
C structs

Like Java class/object without methods.

Compiler determines:
• Total size
• Offset of each field
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.

```c
char* getStudentNameWithId(struct student s[], int id) {
    // Implementation...
}
```
C: Accessing struct fields

Example: traversing a list of structs

```c
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) {
    struct student *cursor = s;
    while (*cursor) {
        if (cursor->id == id)
            return cursor->name;
        cursor++;
    }
    return NULL;
}
```
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

Alignment is especially important for structs.

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

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

#### Unaligned Data (not what C does)

```
  c     v     i
p       p+1   p+9    p+13
```

#### Aligned Data (what C does)

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

```
  c  [7 bytes] v     i
p+0  p+8  p+16    p+20
```

**Internal fragmentation**

- Multiple of 8
- Multiple of 4
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;
```

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;
```

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

“external fragmentation”

“internal fragmentation”
**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;

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;
}
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;
}
```

Extra fun: try a recursive version too!
2-D array practice problem

1. Draw a picture of how this array is laid out in memory, labeling the indices and byte offset of each element (starting with `array[0][0]` at offset +0);

   Recall: \( \text{index} = C \times r + c \)
   scale by element size

2. Write x86 assembly code to implement this function.

```c
long array[2][3];

long get_elem_1_2(long array[2][3]){
    return array[1][2];
}
```
struct practice problem (similar to CSAPP 3.45)

```
struct s {
    char *a;
    short b;
    int *c;
    char d;
    long e;
    char f;
};
```

1. Draw a picture of how this struct is laid out in memory, labeling the byte offset of each field (starting with `a` at offset +0);

2. Modify your picture to show how much space a single element of this struct would take if used as an element of an array (e.g., the total size).

Recall: a short is 2 bytes in C

3. Rearrange the fields of the struct to minimize wasted space. Draw the new offsets and the total size.