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 val is in \%rdi
   - \texttt{i} is in \%rsi

2. Assume:
   - Base address of val is 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: \( \text{T a[N]} \)
Address of \( a[i] \) is:
\( a + i \times \text{sizeof(T)} \)

Which expression correctly loads \texttt{val[i]} into \%rax? Assume \texttt{val} is in \%rdi and \texttt{i} is in \%rsi.

\begin{itemize}
  \item \texttt{movq (rdi, rsi, 4), rax}
  \item \texttt{movq (rdi, rsi, 4), rdx}
  \item \texttt{movq (rdi, rsi, 8), rax}
  \item \texttt{movq (rdi, rsi, 8), rdx}
  \item None of the above
\end{itemize}
C: Arrays of pointers to arrays in x86

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

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

1. Put zips[i] in a reg
2. Access element [j-1] (why)
3. Set element [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!

C: Row-major nested arrays

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

```
\begin{array}{cccc}
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| a[0][0] & \cdots & a[0][C-1] & \vdots \\
| \vdots & \ddots & \vdots & \vdots \\
| a[R-1][0] & \cdots & a[R-1][C-1] & \\
\end{array}
```

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>sea[3][3]</td>
<td>96</td>
<td>76</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>116</td>
<td>96</td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>136</td>
<td>5</td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>76</td>
<td>9</td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>156</td>
<td>8</td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td>156</td>
<td>5</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.

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

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

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;
```

```
struct rec* z;
z = &y;
```

```c
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;
```

```
struct rec* z;
z = &y;
```

```c
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;
```

```
struct rec* z;
z = &y;
```
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;

(*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;

(*z).i++;

// same as:
// z->i++

C: Accessing struct fields

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

Example: traversing a list of structs

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

C: Struct field alignment

Alignment is especially important for structs

Unaligned Data (not what C does)

Aligned Data (what C does)

C: Struct packing

Put large data types first:

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

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.

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

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

**Offset:** 0 4 16 24

---

**Struct in array**

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

- `v` : 3 bytes

---

**Linked Lists**

```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;
}
```

**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 = ...;`

---

**Linked Lists**

```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, (%rbx)
    movl %ebp, 8(%rbx)
    addq $8, %rsp
    popq %rbx
    popq %rbp
    ret
```

- 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 get_elem_1_2(long array[2][3]) {
    return array[1][2];
}
```

Struct practice problem (similar to CSAPP 3.45)

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.