More data layout!

**Array Layout**

Array of data type T and length N
*Continuously* allocated memory region of N * (size of T in bytes)

```
char a[12]
```

```
int b[5]
```

```
double c[3]
```

```
short* d[3]
```

---

**Two-Dimensional Arrays**

**Declaration**

2D array of data type T  
R rows, C columns  
Type T element requires K bytes

**Array layout and size?**

```
A[0][0]  \cdots  A[0][C-1]
\vdots \ddots \vdots
\vdots   \vdots \vdots
A[R-1][0]  \cdots  A[R-1][C-1]
```

---

**Nested Arrays (C)**

**Declaration**

2D array of data type T  
R rows, C columns  
Type T element requires K bytes

**Layout**

Row-major ordering

```
A[0][0]  \cdots  A[0][C-1]
\vdots \ddots \vdots
\vdots   \vdots \vdots
A[R-1][0]  \cdots  A[R-1][C-1]
```

```
int A[R][C];
```
Uh, raise awareness day!

Seat = 4 bytes

Array tag!
Call out 2D index.
Stand up if that’s you.

Nested arrays: 4 rows x 8 columns. Element size: 4 bytes.

Strange Referencing Examples (C)

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>sea[3][3]</td>
<td>76+20<em>3+4</em>3</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>76+20<em>2+4</em>5</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>76+20<em>2+4</em>-1</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>76+20<em>4+4</em>-1</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>76+20<em>0+4</em>19</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td>76+20<em>0+4</em>-1</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

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

Nested Array Access (C)

Each row is an array.
T A[R][C]: A[i] is array of C elements
Each element of type T requires K bytes
Starting address: ________________

Array Elements
A[i][j] is element of type T, which requires K bytes
Address of A[i][j]: ________________

```
int A[R][C];
```

```
A [0] ... A [0] [C-1]
A [1] ... A [1] [C-1]
...    ...    ...
A [R-1] [0]  A [R-1] [C-1]
```

Multi-Level Arrays (Java, C)

```
int[] zips = new int[3][];
zips[0] = new int[5] [0, 2, 4, 8, 1];
int* zips[3];
zips[0] = (int*)malloc(sizeof(int)*5);
...
```

Java

```
... object info [later]
array length
```

Write x86 code to implement:
```
zips[i][j] = zips[i][j - 1];
```

Assume: zips = %eax, i = %ecx, j = %edx

Uh, raise awareness day!

Seat = 4 bytes

Array tag!
Call out 2D index.
Stand up if that's you.

Nested arrays: 4 rows x 8 columns.
Arrays of addresses of arrays:
4 rows x 8 columns
Variable width

Please return the addresses at the end.

Structures in C

Accessing Structure Member
Given a struct, use the . operator, just like Java:

```c
struct rec r1;
r1.i = val;
```

Given a pointer to a struct, use * and . operators or -> shorthand:

```c
struct rec* r = &r1;
(*r).i = val;
r->i = val;
```

```c
void set_i(struct rec* r, int val) {
    r->i = val;
}
```

Characteristics
Contiguously-allocated region of memory
Refer to members within structure by names
Members may be of different types

Like primitive version of Java objects

struct assignment

```c
struct rec a;
struct rec b;
a.i = 1;
a.a[0] = 2;
a.p = &(a.i);
b = a;  // Copies entire contents of a into b.
```

```c
struct rec* c;
c = &b;  // No copying
(*c).i++;
```

What is b.i? a.i?
typedef
struct rec { ... };

struct rec a;
typedef struct list_node {
  ...
} list_node;

typedef struct list_node {
  ...}
list_node;

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

IA32 Assembly
# %eax = val
# %edx = r
movl %eax,0(%edx)  # Mem[r+0] = val

void set_i(struct rec* r, int val)
{
  r->i = val;
}

Generating Pointer to Structure Member

Offset of each structure member determined at compile time

Generating Pointer to Structure Member

void set_i(struct rec* r, int val)
{
  r->i = val;
}

Generating Pointer to Array Element

Offset of each structure member determined at compile time

Generating Pointer to Array Element

int* find_address_of_elem
(struct rec* r, int idx)
{
  return &r->a[idx];
}

int* find_address_of_elem
(struct rec* r, int idx)
{
  return &r->a[idx];
}
Accessing Structure Member

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

```
int* find_address_of_elem (struct rec* r, int idx) {
    return &r->a[idx];
}
```

```
ecx = idx
edx = r
movl 4(%edx,%ecx,4),%eax  # Mov[r+4*idx+4]
```

Reading Array Element
Offset of each structure member still determined at compile time

```
# %ecx = idx
# %edx = r
movl 4(%edx,%ecx,4),%eax  # Mem[r+4*idx+4]
```

Structures & Alignment
Unaligned Data
- Primitive data type requires K bytes
- Address must be multiple of K

Aligned Data
- Treated differently by IA32 Linux, x86-64 Linux, Windows, Mac OS X, ...
- Lowest 2 bits of address must be 00

Specific Cases of Alignment (IA32)
1 byte: char, ...
   - no restrictions on address
2 bytes: short, ...
   - lowest 1 bit of address must be 0
4 bytes: int, float, char *, ...
   - lowest 2 bits of address must be 00
8 bytes: double, ...
   - Windows (and most other OSs & instruction sets): lowest 3 bits 000
   - Linux: lowest 2 bits of address must be 00
   - i.e., treated like 2 contiguous 4-byte primitive data items

Alignment Principles
**Aligned Data**
- Primitive data type requires K bytes
- Address must be multiple of K

**Aligned data is required on some machines; it is advised on IA32**
- Treated differently by IA32 Linux, x86-64 Linux, Windows, Mac OS X, ...

**Motivation for Aligning Data**
- Physical memory is accessed by aligned chunks of 4 or 8 bytes (system-dependent)
  - Inefficient to load or store datum that spans these boundaries
  - Also, virtual memory is very tricky when datum span two pages (later...)

**Compiler**
- Inserts padding in structure to ensure correct alignment of fields
- `sizeof()` should be used to get true size of structs
**Saving Space**

Put large data types first:

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

Effect (example ia32, both have K=8)

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

But actually...

**Struct Alignment Principles**

Size must be a multiple of the largest primitive type inside.

```
K = 8 so size mod 8 = 0
```

**Arrays of Structures**

Satisfy alignment requirement for every element

How would accessing an element work?

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