Course Speed
- 65%!

Multi-Level Array Example
```c
define UCCOUNT
int *univ[UCCOUNT] = {wm, cmw, ucb};
```

2-level vs nested arrays
- `int A[10][10] = {1,...,10, ..., (20,...,29)}`;
  - What do you get when you write: `A[i]`?

- `int *pA[2]; pA[0] = {1, 2, 3}; pA[1] = {5, 6, 7};`
- How many memory accesses in:
  - A[3][4]
  - pA[1][4]
- Btw, what is: `int **parr;`

Nested Array Example
```c
#define PCOUNT 4
int d[PCOUNT][PCOUNT] =
               { 9, 8, 1, 0.5, },
               { 9, 0, 1, 1.5, },
               { 9, 8, 1, 0.5, },
               { 9, 0, 1, 1.5, },
```

- “row-major” ordering of all elements
- Guaranteed?

Multi-Level Array Example
```c
define UCCOUNT
int *univ[UCCOUNT] = {wm, cmw, ucb};
```

Array Element Accesses
```c
int get_univ_digit
int get_uuniv_digit
```
Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>univ[2][3]</td>
<td>56+i+3</td>
<td>68</td>
<td>Yes</td>
</tr>
<tr>
<td>univ[1][5]</td>
<td>16+i+5</td>
<td>36</td>
<td>No</td>
</tr>
<tr>
<td>univ[2][6]</td>
<td>56+i+1</td>
<td>52</td>
<td>No</td>
</tr>
<tr>
<td>univ[3][7]</td>
<td>??</td>
<td>??</td>
<td>No</td>
</tr>
<tr>
<td>univ[1][12]</td>
<td>16+i+12</td>
<td>64</td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking.
- Ordering of elements is different across arrays, not guaranteed.

Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>univ[2][3]</td>
<td>56+i+3 = 68</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>univ[1][5]</td>
<td>16+i+5 = 36</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>univ[2][6]</td>
<td>56+i+1 = 52</td>
<td>5</td>
<td>No</td>
</tr>
<tr>
<td>univ[3][7]</td>
<td>??</td>
<td>??</td>
<td>No</td>
</tr>
<tr>
<td>univ[1][12]</td>
<td>16+i+12 = 64</td>
<td>7</td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking.
- Ordering of elements is different across arrays, not guaranteed.

Structures

Struct rec {
    int i;
    int g[3]:
    int p;
};

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

- Accessing structure member
  - In Java: r.i = val;
  - IA32 Assembly
    ```assembly
    movl %eax, %esi
    addl %edi, %esi
    movl %esi, %edi
    addl %edi, %eax
    ```
Generating Pointer to Structure Member

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

```
int *find_a // r + 4 * idx
    (struct vec *v, int idm)
    { 
      return &v->a[idm]; 
      // return &v->a[((i)*a + id0)]; 
    }
```

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

```
int *find_a // r + 4 * idx
    (struct vec *v, int idm)
    { 
      return &v->a[idm]; 
      // return &v->a[((i)*a + id0)]; 
    }
```

```
void set_p(struct vec *t)
    { 
      *p = *p + 2; 
      // *(p) = *(p) + 2; 
    }
```

Alignment

**Aligned Data**
- Primitive data type requires K bytes
- Address must be multiple of K
- Required on some machines; advised on IA32
  - treated differently by IA32 Linux, x86-64 Linux, and Windows

**Motivation for Aligning Data**
- Memory accessed by (aligned) chunks of 4 or 8 bytes (system-dependent)
  - Inefficient to load or store datum that spans quad word boundaries
  - Virtual memory very tricky when datum spans two pages (later...)

**Compiler**
- Inserts gaps in structure to ensure correct alignment of fields

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 0

**8 bytes: double, ...**
- Windows (and most other OS’s & instruction sets): lowest 3 bits 000
  - Linux: lowest 2 bits of address must be 00
  - i.e., treated the same as a 4-byte primitive data type

**12 bytes: long double**
- Windows, Linux: (same as Linux double)
Alignment and structs

- Hmm, how would you satisfy alignments in structs?

Satisfying Alignment with Structures

- Within structure:
  - Must satisfy element’s alignment requirement

- Overall structure placement:
  - Each structure has alignment requirement \( K \)
  - \( K \) = largest alignment of any element
  - Initial address & structure length must be multiples of \( K \)

- Example (under Windows or x86-64):
  - \( K = 8 \), due to \texttt{double} element

<table>
<thead>
<tr>
<th>Offset</th>
<th>( i[0] )</th>
<th>( i[1] )</th>
<th>( \text{char} )</th>
<th>( \text{int} )</th>
<th>( \text{double} )</th>
<th>( \text{p} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>p=0</td>
<td>Multiple of 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p=4</td>
<td>Multiple of 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p=8</td>
<td>Multiple of 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p=16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p=24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Arrays of structs?

Unions

- Concept:
  - Allow same regions of memory to be referenced as different types
  - Aliases for the same memory location
Union Allocation
- Allocate according to largest element
- Can only use one field at a time

```c
union U1 {
  char c;
  int i[2];
  double v;
} *sp;
```

```c
union U1 {
  char c;
  int i[2];
  double v;
} *sp;
```

Using Union to Access Bit Patterns
```c
typeset union {
  float f;
  unsigned u;
  bit_float_t;
}
```
```c
float bit2float(unsigned u) {
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```
```c
unsigned float2bit(float f) {
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```
Same as `float` u? Same as `(unsigned) f`?

Summary
- Arrays in C
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
  - Pointer to first element
  - No bounds checking
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Overlay declarations
  - Way to circumvent type system