Data Structures in Memory!

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structs
  - Alignment
- Unions

Data Structures in Assembly...

- Arrays?
- Strings?
- Structs?
**Array Allocation**

- **Basic Principle**
  - `T A[N];`
  - Array of data type `T` and length `N`
  - Contiguously allocated region of `N * sizeof(T)` bytes

- `char string[12]:`  
- `int val[5]:`  
- `double a[3]:`  
- `char *p[3]:`

**Array Access**

- **Basic Principle**
  - `T A[N];`
  - Array of data type `T` and length `N`
  - Identifier `A` can be used as a pointer to array element 0: Type `T*`

- `int val[5]:`

- **Reference** | **Type** | **Value**
  - `val[4]` int
  - `val` int *
  - `val+1` int *
  - `&val[2]` int *
  - `val[5]` int
  - `*(val+1)` int
  - `val + i` int *
Array Access

- Basic Principle
  - T A[N];
  - Array of data type T and length N
  - Identifier A can be used as a pointer to array element 0: Type T*

```
int val[5];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>x</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>x + 4</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>x + 8</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>8</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>x + 4 i</td>
</tr>
</tbody>
</table>

Array Example

```
typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```
Array Example

typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

- Declaration “zip_dig uw” equivalent to “int uw[5]”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

int get_digit(zip_dig z, int dig)
{
    return z[dig];
}

# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax  # z[dig]

- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at 4*%eax + %edx
- Use memory reference (%edx,%eax,4)
### Referencing Examples

<table>
<thead>
<tr>
<th>zip_dig cmu;</th>
<th>1</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
</tr>
<tr>
<td>zip_dig uw ;</td>
<td>9</td>
<td>8</td>
<td>1</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td>52</td>
</tr>
<tr>
<td>zip_dig ucb;</td>
<td>9</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>60</td>
<td>64</td>
<td>68</td>
<td>72</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
</table>

- Reference Address Value Guaranteed?

- No bound checking
- Out-of-range behavior implementation-dependent
- No guaranteed relative allocation of different arrays
Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>uw[3]</td>
<td>36 + 4* 3 = 48</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>uw[6]</td>
<td>36 + 4* 6 = 60</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>uw[-1]</td>
<td>36 + 4*-1 = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15 = 76</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- No bound checking
- Out-of-range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

Array Loop Example

```c
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```
Array Loop Example

**Original**

```c
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

**Transformed**

- Eliminate loop variable `i`
- Convert array code to pointer code
- Express in do-while form (no test at entrance)

```c
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z <= zend);
    return zi;
}
```

Array Loop Implementation (IA32)

```assembly
.L59:
    leal (%eax,%edx,2),%eax
    cmpl %ebx,%ecx
    jle .L59
```

**Translation?**
Array Loop Implementation (IA32)

- Registers
  - %ecx = z
  - %eax = zi
  - %ebx = zend

- Computations
  1. \(10 \times zi + *z\) implemented as 
     \(*z + 2 \times (zi+4 \times zi)\)
  2. \(z++\) increments by 4

```c
int zd2int(zip_dig z) {
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

```
# %ecx = z
xorl %eax,%eax     # zi = 0
leal 16(%ecx),%ebx # zend = z+4
.L59:
    leal (%eax,%eax,4),%edx # 5*zi
    movl (%ecx),%eax       # *z
    addl $4,%ecx           # z++
    leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
    cmpl %ebx,%ecx         # z : zend
    jle .L59               # if <= goto loop
```

Nested Array Example

```c
#define PCOUNT 4
zip_dig sea[PCOUNT] =
    { { 9, 8, 1, 9, 5 },
      { 9, 8, 1, 0, 5 },
      { 9, 8, 1, 0, 3 },
      { 9, 8, 1, 1, 5 } };```

```c
```
Nested Array Example

```c
#define PCOUNT 4
zip_dig sea[PCOUNT] =
    {{ 9, 8, 1, 9, 5 },
     { 9, 8, 1, 0, 5 },
     { 9, 8, 1, 0, 3 },
     { 9, 8, 1, 1, 5 }};
```

- “row-major” ordering of all elements
- Guaranteed?
Multidimensional (Nested) Arrays

- **Declaration**
  - \( T \ A[R][C]; \)
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Type \( T \) element requires \( K \) bytes

- **Array size?**

- **Arrangement**
  - Row-major ordering

```plaintext
int A[R][C];
```
### Nested Array Row Access

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

- **Row vectors**
  - $A[i]$ is array of $C$ elements
  - Each element of type $T$ requires $K$ bytes
  - Starting address $A + i \times (C \times K)$

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

- **Row vectors**
  - $A[i]$ is array of $C$ elements
  - Each element of type $T$ requires $K$ bytes
  - Starting address $A + i \times (C \times K)$
Nested Array Row Access Code

```c
int *get_sea_zip(int index) {
    return sea[index];
}
```

```c
#define PCOUNT 4
zip_dig sea[PCOUNT] = {
    { 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 }};
```

- What data type is `sea[index]`?
- What is its starting address?

```assembly
# %eax = index
leal (%eax,%eax,4),%eax
leal sea(,%eax,4),%eax
```

Translation?
Nested Array Row Access Code

```c
int *get_sea_zip(int index)
{
    return sea[index];
}
```

```c
#define PCOUNT 4
zip_dig sea[PCOUNT] =
    {{ 9, 8, 1, 9, 5 },
     { 9, 8, 1, 0, 5 },
     { 9, 8, 1, 0, 3 },
     { 9, 8, 1, 1, 5 }};
```

- **Row Vector**
  - `sea[index]` is array of 5 ints
  - Starting address `sea+20*index`

- **IA32 Code**
  - Computes and returns address
  - Compute as `sea+4*(index+4*index)=sea+20*index`

```
# %eax = index
   leal (%eax,%eax,4),%eax  # 5 * index
   leal sea(,%eax,4),%eax  # sea + (20 * index)
```

### Nested Array Row Access

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

- `A` is a 2D array of integers
- `A[i][j]` is an element
- `A[R-1][C-1]` is the last element
- `A + i*C*4` is the address of `A[i][j]`
- `A + (R-1)*C*4` is the address of `A[R-1][C]`

```
  ⋮      ⋮      ⋮
  ⋮      ⋮      ⋮
```

Autumn 2012  Data Structures
Nested Array Row Access

- **Array Elements**
  - $A[i][j]$ is element of type $T$, which requires $K$ bytes
  - Address: $A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K$

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

```plaintext
[0]  [0]  [R-1]
[0]  [C-1] [C-1]

\[ A \] \rightarrow \[ A + i \times C \times 4 \] \rightarrow \[ A + (R-1) \times C \times 4 + j \times 4 \]
```

Nested Array Element Access Code

```c
int get_sea_digit(int index, int dig)
{
    return sea[index][dig];
}
```

```plaintext
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx    # 4*dig
leal (%eax,%eax,4),%eax  # 5*index
movl sea(%edx,%eax,4),%eax   # *(sea + 4*dig + 20*index)
```

- **Array Elements**
  - sea[index][dig] is int
  - Address: sea + 20*index + 4*dig

- **IA32 Code**
  - Computes address sea + 4*dig + 4*(index+4*index)
  - `movl` performs memory reference
Strange Referencing Examples

```
zip_dig
sea[4];
```

<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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[2][5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[0][19]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
76 96 116 136 156
```

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
Strange Referencing Examples

<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>148</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>76+20<em>2+4</em>5</td>
<td>136</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>76+20<em>2+4</em>-1</td>
<td>112</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>76+20<em>4+4</em>-1</td>
<td>152</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>76+20<em>0+4</em>19</td>
<td>152</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>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed

Multi-Level Array Example

```c
#define UCOUNT 3
int *univ[UCOUNT] = {uw, cmu, ucb};
```

Same thing as a 2D array?
**Multi-Level Array Example**

```c
#define UCOUNT 3
int *univ[UCOUNT] = {uw, cmu, ucb};
```

NB: This is how Java represents multi-dimensional arrays.

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of ints
Element Access in Multi-Level Array

```c
int get_univ_digit
(int index, int dig)
{
    return univ[index][dig];
}
```

Computation (IA32)
- Element access `Mem[Mem[univ+4*index]+4*dig]`
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

```c
# %ecx = index
# %eax = dig
leal 0(%ecx,4),%edx  # 4*index
movl univ(%edx),%edx  # Mem[univ+4*index]
movl (%edx,%eax,4),%eax  # Mem[...+4*dig]
```
Array Element Accesses

Nested array

```c
int get_sea_digit(int index, int dig)
{
    return sea[index][dig];
}
```

Multi-level array

```c
int get_univ_digit(int index, int dig)
{
    return univ[index][dig];
}
```

Access looks similar, but it isn’t:

```c
```

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>160</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>univ[1][5]</td>
<td>164</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>univ[2][-1]</td>
<td>168</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>univ[3][-1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>univ[1][12]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What values go here?
## Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>univ[2][3]</code></td>
<td><code>56+4*3 = 68</code></td>
<td><code>2</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>univ[1][5]</code></td>
<td><code>16+4*5 = 36</code></td>
<td><code>9</code></td>
<td>No</td>
</tr>
<tr>
<td><code>univ[2][-1]</code></td>
<td><code>56+4*-1 = 52</code></td>
<td><code>5</code></td>
<td>No</td>
</tr>
<tr>
<td><code>univ[3][-1]</code></td>
<td><code>??</code></td>
<td><code>??</code></td>
<td>No</td>
</tr>
<tr>
<td><code>univ[1][12]</code></td>
<td><code>16+4*12 = 64</code></td>
<td><code>7</code></td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed
Using Nested Arrays

```c
#define N 16
typedef int fix_matrix[N][N];

/* Compute element i,k of fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b, int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```

- **Strengths**
  - C compiler handles doubly subscripted arrays
  - Generates very efficient code
  - Avoids multiply in index computation

- **Limitation**
  - Only works for fixed array size

![Diagram of matrix multiplication with nested arrays]
Dynamic Nested Arrays

- **Strength**
  - Can create matrix of any size

- **Programming**
  - Must do index computation explicitly

- **Performance**
  - Accessing single element costly
  - Must do multiplication

```c
int * new_var_matrix(int n) {
    return (int *)
        calloc(sizeof(int), n*n);
}
```

```c
int var_ele
    (int *a, int i, int j, int n)
    {
        return a[i*n+j];
    }
```

```asm
movl 12(%ebp),%eax # i
movl 8(%ebp),%edx # a
imull 20(%ebp),%eax # n*i
addl 16(%ebp),%eax # n*i+j
movl (%edx,%eax,4),%eax # Mem[a+4*(i*n+j)]
```

Structures

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

```c
struct rec {
    int i;
    int a[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

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

In java: `r.i = val;`

IA32 Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx) # Mem[r] = val
```

Generating Pointer to Structure Member

```c
struct rec {
    int i;
    int a[3];
    int *p;
};
```
Generating Pointer to Structure Member

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

### Generating Pointer to Array Element
- Offset of each structure member determined at compile time

```c
int *find_a // r.a[idx]
(struct rec *r, int idx)
{
    return &r->a[idx];
    // return &(*((*r).a + idx));
}
```

```c
# %ecx = idx
# %edx = r
leal 0(%ecx,4),%eax     # 4*idx
leal 4(%eax,%edx),%eax   # r+4*idx+4
```

Structure Referencing (Cont.)

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

```c
void set_p(struct rec *r)
{
    r->p = &r->a[r->i];
    // (*r).p = &(*((*r).a+(*r).i));
}
```
Structure Referencing (Cont.)

**C Code**

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

void set_p(struct rec *r) {
    r->p = &r->a[r->i];
    // (*r).p = &(*(r->a+(*r).i));
}
```

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!

**What is the motivation for alignment?**
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_2

- **4 bytes: int, float, char *, ...**
  - lowest 2 bits of address must be 00_2

- **8 bytes: double, ...**
  - Windows (and most other OS’s & instruction sets): lowest 3 bits 000_2
  - Linux: lowest 2 bits of address must be 00_2
    - i.e., treated the same as a 4-byte primitive data type

- **12 bytes: long double**
  - Windows, Linux: (same as Linux double)
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 double element

  ```c
  struct S1 {
      char c;
      int i[2];
      double v;
  } *p1;
  ```

Different Alignment Conventions

- **IA32 Windows or x86-64:**
  - K = 8, due to double element

  ```c
  struct S1 {
      char c;
      int i[2];
      double v;
  } *p1;
  ```

- **IA32 Linux**
  - K = 4; double treated like a 4-byte data type
Saving Space

- Put large data types first

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p1;

struct S2 {
    double v;
    int i[2];
    char c;
} *p2;
```

- Effect (example x86-64, both have K=8)
Arrays of Structures

- Satisfy alignment requirement for every element

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```

```
a[0]   a[1]   a[2]   ...
```

```
a+0    a+24   a+48   a+72
```

```
v  i[0]  i[1]  c
```

7 bytes

```
a+24   a+32   a+40   a+48
```
Accessing Array Elements

- Compute array offset 12i
- Compute offset 8 with structure
- Assembler gives offset a+8
  - Resolved during linking

```
| a[0] |   *   | a[i] |   *   |
  +-----+-------+------+
  | v    | i bytes | j bytes |
  +-----+-------+------+
```

```c
short get_j(int idx)
{
    return a[idx].j;
    // return (a + idx)->j;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(%eax,4),%eax
```

Unions

- Concept
  - Allow same regions of memory to be referenced as different types
  - Aliases for the same memory location

```
struct rec {
    int i;
    int a[3];
    int *p;
};
union urec {
    int i;
    int a[3];
    int *p;
};
```
Unions

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

union urec {
    int i;
    int a[3];
    int *p;
};
```

- **Concept**
  - Allow same regions of memory to be referenced as different types
  - Aliases for the same memory location

Structure Layout

```
<table>
<thead>
<tr>
<th>i</th>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Union Layout

```
<table>
<thead>
<tr>
<th>i</th>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12</td>
</tr>
</tbody>
</table>
```
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;
} *up;
```

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```

Using Union to Access Bit Patterns

```c
typedef 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