Roadmap

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();
```

Assembly language:

```
get_mpg:
pushq   %rbp
movq    %rsp, %rbp
...
popq    %rbp
ret
```

Machine code:

```
0111010000011000
10001110100000010
1000100111000010
110000011111100100011111
```

Computer system:

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

Autumn 2014

Arrays & structs
Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structs**
  - Alignment

- **Unions**
Array Allocation

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

```
char mesg[12];
```

```x```

```x + 12```

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

```x```

```x + 4```

```x + 8```

```x + 12```

```x + 16```

```x + 20```

```double a[3];```

```x```

```x + 8```

```x + 16```

```x + 24```

```char* p[3];```

```x```

```x + 4```

```x + 8```

```x + 12```

```IA32```

```x86-64```
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></td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td></td>
</tr>
</tbody>
</table>

Autumn 2014
Arrays & structs
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*

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

- **Reference**
  - | Type   | Value |
  - |-------|-------|
  - | val    | int * | x     |
  - | val+1  | int * | x + 4 |
  - | &val[2]| int * | x + 8 |
  - | val[5] | int   | ?? (whatever is in memory at address x + 20) |
  - | *(val+1) | int | 8    |
  - | val + i | int * | x + 4*i |
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 };  

int uw[5] ...
Array Example

```c
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 happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array Accessing Example

zip_dig uw;

<table>
<thead>
<tr>
<th>36</th>
<th>40</th>
<th>44</th>
<th>48</th>
<th>52</th>
<th>56</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

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

**IA32**

```asm
movl 8(%ebp), edx # %edx = z
movl 12(%ebp), eax # %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```
Referencing Examples

\[ \text{zip\_dig \ cmu;} \]
\[ \begin{array}{cccccc}
1 & 5 & 2 & 1 & 3 \\
16 & 20 & 24 & 28 & 32 & 36
\end{array} \]

\[ \text{zip\_dig \ uw;} \]
\[ \begin{array}{cccccc}
9 & 8 & 1 & 9 & 5 \\
36 & 40 & 44 & 48 & 52 & 56
\end{array} \]

\[ \text{zip\_dig \ ucb;} \]
\[ \begin{array}{cccccc}
9 & 4 & 7 & 2 & 0 \\
56 & 60 & 64 & 68 & 72 & 76
\end{array} \]

- Reference
- Address
- Value
- Guaranteed?

\text{uw[3]}
\text{uw[6]}
\text{uw[-1]}
\text{cmu[15]}
Referencing Examples

- **zip_dig cmu;**
  - 1  5  2  1  3
  - 16 20 24 28 32 36

- **zip_dig uw;**
  - 9  8  1  9  5
  - 36 40 44 48 52 56

- **zip_dig ucb;**
  - 9  4  7  2  0
  - 56 60 64 68 72 76

<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 bounds checking
- Location of each separate array in memory is not guaranteed
Array Loop Example

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

zi = 10*0 + 9 = 9

zi = 10*9 + 8 = 98

zi = 10*98 + 1 = 981

zi = 10*981 + 9 = 9819

zi = 10*9819 + 5 = 98195
Array Loop Example

**Original**

```cpp
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`, use pointer `zend` instead
- Convert array code to pointer code
  - Pointer arithmetic on `z`
- Express in do-while form (no test at entrance)

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

<table>
<thead>
<tr>
<th>zip_dig</th>
<th>uw</th>
<th>9</th>
<th>8</th>
<th>1</th>
<th>9</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td>52</td>
</tr>
</tbody>
</table>
| address of 5th digit | | | | | | | Increments by 4 (size of int)
Array Loop Implementation (IA32)

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

Computations
- $10*zi + *z$ implemented as: $*z + 2*(5*zi)$
- $z++$ increments by 4 (size of int=4)

```asm
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 #  zi  +  4*zi  =  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
zip_dig sea[4] =
    {{{ 9, 8, 1, 9, 5 }},
     { 9, 8, 1, 0, 5 },{ 9, 8, 1, 0, 3 },{ 9, 8, 1, 1, 5 }};
```

Remember, \( \text{T A[N]} \) is an array with elements of type \( \text{T} \), with length \( \text{N} \)

What is the layout in memory?

```c
int sea[4][5];
```
Nested Array Example

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

Remember, `T A[N]` is an array with elements of type `T`, with length `N`.

```
sea[3][2];
```

- “Row-major” ordering of all elements
- Guaranteed (in C)
Two-Dimensional (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?**
# Two-Dimensional (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:
- \( R \times C \times K \) bytes

## Arrangement
- **Row-major** ordering

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

<table>
<thead>
<tr>
<th>A[0][0]</th>
<th>( \ldots )</th>
<th>A[R][0]</th>
<th>( \ldots )</th>
<th>A[R][C-1]</th>
<th>( \ldots )</th>
<th>A[R-1][0]</th>
<th>( \ldots )</th>
<th>A[R-1][C-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>( \ldots )</td>
<td>[1]</td>
<td>( \ldots )</td>
<td>[R-1]</td>
<td>( \ldots )</td>
<td>[0]</td>
<td>( \ldots )</td>
<td>[R-1]</td>
</tr>
<tr>
<td>[0]</td>
<td>( \ldots )</td>
<td>[0]</td>
<td>( \ldots )</td>
<td>[0]</td>
<td>( \ldots )</td>
<td>[C-1]</td>
<td>( \ldots )</td>
<td>[C-1]</td>
</tr>
<tr>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

\[ 4 \times R \times C \text{ Bytes} \]
Nested Array Row Access

Row vectors

- Given: \( T \ A[R][C] \):
  - \( A[i] \) is an array of \( C \) elements, “row \( i \)”
  - Each element of type \( T \) requires \( K \) bytes
  - \( A \) is starting address of array
  - Starting address of row \( i = A + i \times (C \times K) \)

\[
\text{int } A[R][C];
\]

```
\begin{array}{c}
A[0] \hspace{2cm} A[i] \hspace{2cm} A[R-1] \\
\begin{array}{c}
\begin{array}{c}
A[0] \hspace{2cm} \cdots \hspace{2cm} A[C-1] \\
A[0] \hspace{2cm} \cdots \hspace{2cm} A[C-1] \\
A[0] \hspace{2cm} \cdots \hspace{2cm} A[C-1]
\end{array}
\end{array}
\end{array}
```

\( A[i][0] \) \hspace{2cm} \( A[i][C-1] \) \hspace{2cm} \( A[R-1][0] \) \hspace{2cm} \( A[R-1][C-1] \)

\( A[0] \) \hspace{2cm} \( A+i*4 \) \hspace{2cm} \( A+(R-1)*4 \)
Nested Array Row Access Code

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

int sea[4][5] =
    {{ 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?
Nested Array Row Access Code

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

```c
int sea[4][5] = {
    { 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?

```c
# %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
int sea[4][5] = {
    { 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 }
};
```

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

- **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`
Nested Array Element Access

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

```
A[0]
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
A

A[0]
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

A[i]
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

A[R-1]
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

A[0]
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
A

A*4

A + i*C*4

A + (R-1)*C*4

?
Nested Array Element Access

Array Elements

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

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

Diagram showing the memory layout of a 2D array $A[R][C]$ with elements $A[i][j]$. The address calculation is visualized with arrows pointing to the corresponding memory locations.
Nested Array Element Access Code

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

```c
int sea[4][5] =
    {{ 9, 8, 1, 9, 5 },
     { 9, 8, 1, 0, 5 },
     { 9, 8, 1, 0, 3 },
     { 9, 8, 1, 1, 5 }};
```

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

- **Array Elements**
  - `sea[index][dig]` is an 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>9 8 1 9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>9 8 1 0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>9 8 1 0</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>9 8 1 0</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>9 8 1 0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td>9 8 1 0</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

- 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 = 148</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
N-dimensional arrays...

double heatMap3D[1024][1024][1024];

total size in bytes?

\[ 1024 \times 1024 \times 1024 \times 8 = 8,589,934,592 = \text{roughly 8GB} \]

&heapMap3D[300][800][2] = ?

in bytes: base + \[ 300 \times 1024 \times 1024 \times 8 + 800 \times 1024 \times 8 + 2 \times 8 \]

\[ = \text{base} + 8 \times (2 + 1024 \times (800 + 1024 \times (300))) \]

\[ = \text{base} + 2,523,136,016 \]
Multi-Level Array Example

Multi-Level Array Declaration(s):

```c
int cmu[5] = {1, 5, 2, 1, 3};
int uw[5] = {9, 8, 1, 9, 5};
int ucb[5] = {9, 4, 7, 2, 0};
int* univ[3] = {uw, cmu, ucb};
```

Is a multi-level array the same thing as a 2D array? NO

2D Array Declaration:

```c
zip_dig univ2D[3] = {
{9, 8, 1, 9, 5},
{1, 5, 2, 1, 3},
{9, 4, 7, 2, 0}
};
```

One array declaration = one contiguous block of memory
Multi-Level Array Example

Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of ints

```java
int cmu[5] = { 1, 5, 2, 1, 3 };
int uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };

int* univ[3] = {uw, cmu, ucb};
```

Note: this is how Java represents multi-dimensional arrays.
Element Access in Multi-Level Array

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

- `# %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]

**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
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:

```
```
Strange Referencing Examples

- C Code does not do any bounds checking
- Location of each lower-level array in memory is 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>60+4*3 = 72</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>univ[1][5]</td>
<td>16+4*5 = 36</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>univ[2][-2]</td>
<td>60+4*-2 = 52</td>
<td>5</td>
<td>No</td>
</tr>
<tr>
<td>univ[3][-1]</td>
<td>#@$%^??</td>
<td>??</td>
<td>No</td>
</tr>
<tr>
<td>univ[1][12]</td>
<td>16+4*12 = 64</td>
<td>4</td>
<td>No</td>
</tr>
</tbody>
</table>

- C Code does not do any bounds checking
- Location of each lower-level array in memory is not guaranteed
#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;
}
Using Nested Arrays: arrays of arrays

**Strengths**
- Generates very efficient assembly code
- Avoids multiply in index computation

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

```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;
}
```

- **Autumn 2014**
- **Arrays & structs**
Dynamic Nested Arrays: pointer to a 2D array

- **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);
}

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

```
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)]
```
Summary: Arrays in C

- Contiguous allocations of memory
- No bounds checking
- Can usually be treated like a pointer to first element
- `int a[4][5]` => array of arrays
  - all levels in one contiguous block of memory
- `int* b[4]` => array of pointers to arrays
  - first level in one contiguous block of memory
  - Each element in the first level points to another “sub” array
  - parts anywhere in memory
Structures

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

Memory Layout

```
0 4 16 20
```

 Characteristics

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

- Given an instance of the struct, we can use the . operator, just like Java:
  
  ```c
  struct rec r1;
  r1.i = val;
  ```

- What if we have a pointer to a struct:
  
  ```c
  struct rec* r = &r1;
  ```
Structures

- Accessing Structure Member
  - Given an instance of the struct, we can use the . operator, just like Java:
    ```c
    struct rec r1;
    r1.i = val;
    ```
  - What if we have a pointer to a struct: `struct rec* r = &r1;`
    - Using * and . operators:
      ```c
      (*r).i = val;
      ```
    - Or, use -> operator for short:
      ```c
      r->i = val;
      ```
  - Pointer indicates first byte of structure; access members with offsets

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

IA32 Assembly

```assembly
# %eax = val
# %edx = r
movl %eax,0(%edx)  # Mem[r+0] = val
```
Generating a **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_address_of_elem  
(struct rec* r, int idx)  
{  
    return &r->a[idx];  
}
```

```
# %ecx = idx  
# %edx = r  
leal 0(%ecx,4),%eax  # 4*idx  
leal 4(%eax,%edx),%eax  # r+4*idx+4
```
Generating a **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_address_of_elem  
  (struct rec* r, int idx)  
{  
  return &r->a[idx];  
}
```

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

OR

```assembly
leal 4(%edx,%ecx,4),%eax  # r+4*idx+4
```
Accessing a Structure Member

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

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

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

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

- Unaligned Data

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

- What would it look like if data items were **aligned** (address multiple of type size)?

Create one instance of the struct and assign p to point to it.
Structures & Alignment

- **Unaligned Data**

  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

- **Aligned Data**

  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

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

Internal fragmentation

(This is not quite accurate yet...)

Arrays & structs
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, ...

- **What is the motivation for alignment?**
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 spans two pages (later...)

- Compiler
  - Maintains declared *ordering* of fields in struct
  - Inserts padding in structure to ensure correct *alignment* of fields
  - `sizeof()` should be used to get true size of structs
  - `offsetof(struct, field)` can be used to find the actual offset of a field
Specific Cases of Alignment (IA32)

- 1 byte: char, ...
  - no restrictions on address

- 2 bytes: short, ...
  - lowest 1 bit of address must be 0₂

- 4 bytes: int, float, pointers, ...
  - lowest 2 bits of address must be 00₂

- 8 bytes: double, ...
  - Windows (and most other OSs & instruction sets): lowest 3 bits 000₂
  - Linux: lowest 2 bits of address must be 00₂
    - i.e., treated liked 2 contiguous 4-byte primitive data items
Saving Space by putting large data types first

- **Original Struct:**

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

- **Programmer re-writes as:**

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

- `double` is 8 bytes, so for x86-64 we get:

  - For `p`:
    - `c` at `p+0`
    - `7 bytes`
    - `v` at `p+8`
    - `i` at `p+16`

  - For `q`:
    - `v` at `q+0`
    - `i` at `q+8`
    - `c` at `q+12`
    - `c` at `q+13`

But this is still not quite accurate...
Overall Struct Alignment

- **Overall size of struct** must be:
  - a multiple of the largest primitive type inside.
- **Both have largest primitive=8 bytes, so for x86-64 must have:**
  - overall size mod 8 = 0

(This is what gcc on x86-64 does)
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];

Create an array of ten S2 structs called “a”
Unions

- Allocated according to largest element
- Can only use one member at a time

```c
union U {
    char c;
    int i[2];
    double v;
} *up;

struct S {
    char c;
    int i[2];
    double v;
} *sp;
```
What Are Unions Good For?

- Unions allow the same region of memory to be referenced as different types
  - Different “views” of the same memory location
  - Can be used to circumvent C’s type system (bad idea)
- Better idea: use a struct inside a union to access some memory location either as a whole or by its parts
- But watch out for endianness at a small scale...
- Layout details are implementation/machine-specific...

```c
union int_or_bytes {
    int i;
    struct bytes {
        char b0, b1, b2, b3;
    }
}
```
Unions For Embedded Programming

typedef union
{
    unsigned char byte;
    struct {
        unsigned char reserved:4;
        unsigned char b3:1;
        unsigned char b2:1;
        unsigned char b1:1;
        unsigned char b0:1;
    } bits;
} hw_register;

hw_register reg;
reg.byte = 0x3F; // 00111111
reg.bits.b2 = 0; // 00111011
reg.bits.b3 = 0; // 00110011

unsigned short a = reg.byte;
printf("0x%X\n", a); // output: 0x33

(Note: the placement of these fields and other parts of this example are implementation-dependent)
Summary

- **Arrays in C**
  - Contiguous allocations of memory
  - No bounds checking
  - Can usually be treated like a pointer to first element

- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment

- **Unions**
  - Provide different views of the same memory location