### Roadmap

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

**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`
- `100011010000010000000010`
- `1000100111000010`
- `110000011111101000011111`

**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

### 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

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

**Examples:**
- `int val[5];`
- `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*

<table>
<thead>
<tr>
<th>T, A[N];</th>
</tr>
</thead>
<tbody>
<tr>
<td>x  x + 4  x + 8  x + 12  x + 16  x + 20</td>
</tr>
</tbody>
</table>

<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>?? (whatever is in memory at address x + 20)</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

```c
typedef int zip_dig[5];

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

<table>
<thead>
<tr>
<th>zip_dig cmu</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>zip_dig uw</td>
<td>9</td>
<td>8</td>
<td>1</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>zip_dig ucb</td>
<td>9</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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

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

```asm
# Register %edx contains starting address of array
# Register %eax contains array index
# Desired digit at 4*%eax + %edx
# Use memory reference (%edx, %eax, 4)
```
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;
}
```

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 Implementation (IA32)

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

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

- **Computations**
  - 10*zi + *z implemented as *z + 2*(5*zi)
  - z++ increments by 4

```
# %ecx = z
xorl %eax,%eax
lea 16(%ecx),%ebx
    # zi = 0
.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
```

References
- No bounds checking
- Location of each separate array in memory is not guaranteed

### Array Loop Example

- **Original**
- **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)
### 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`

What is the layout in memory?

```c
int sea[4][5];
```

### 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**

```c
A[0][0] \cdots A[0][C-1]
\cdots \cdots \cdots
A[R-1][0] \cdots A[R-1][C-1]
```

- **Arrangement**
  - Row-major ordering

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

- **Array size**
  - `R \times C \times K` bytes

### 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`

```c
int sea[3][2];
```

- “Row-major” ordering of all elements
- Guaranteed (in C)
Nested Array Row Access

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

- \( A[R][C] \):
  - \( A[0][0] \)
  - \( A[0][C-1] \)
  - \( A[i][0] \)
  - \( A[i][C-1] \)
  - \( A[R-1][0] \)
  - \( A[R-1][C-1] \)

- \( A+i \times C \times 4 \)
- \( A+(R-1) \times C \times 4 \)

### Nested Array Row Access Code

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

- int \( \text{sea}[4][5] = \{ \{ 9, 8, 1, 9, 5 \}, \{ 9, 8, 1, 0, 5 \}, \{ 9, 8, 1, 0, 3 \}, \{ 9, 8, 1, 1, 5 \} \}; \)

- What data type is \( \text{sea}[\text{index}] \)?
- What is its starting address?

### Nested Array Row Access Code

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

- int \( \text{sea}[4][5] = \{ \{ 9, 8, 1, 9, 5 \}, \{ 9, 8, 1, 0, 5 \}, \{ 9, 8, 1, 0, 3 \}, \{ 9, 8, 1, 1, 5 \} \}; \)

- What data type is \( \text{sea}[\text{index}] \)?
- What is its starting address?

```c
leal (%eax,%eax,4),%eax
leal sea(,%eax,4),%eax
```

- Translation?

### Nested Array Row Access Code

- \# %eax = index
- leal (%eax,%eax,4),%eax
- leal sea(,%eax,4),%eax

- What data type is \( \text{sea}[\text{index}] \)?
- What is its starting address?

```c
leal (%eax,%eax,4),%eax
leal sea(,%eax,4),%eax
```

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

- Row Vector
  - \( \text{sea}[\text{index}] \) is array of 5 ints
  - Starting address \( \text{sea}+20\times \text{index} \)

- IA32 Code
  - Computes and returns address
  - Compute as \( \text{sea}+4\times (\text{index}+4\times \text{index}) = \text{sea}+20\times \text{index} \)
Nested Array Row Access

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

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

```
+---------------------+---------------------+---------------------+---------------------+
+---------------------+---------------------+---------------------+---------------------+
| 0                   | [i]                 | [R-1]               | 0                   |
+---------------------+---------------------+---------------------+---------------------+
| A                   | A + i*C*4           | A + (R-1)*C*4       | A + j*4             |
+---------------------+---------------------+---------------------+---------------------+
```

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}};
```

Strange Referencing Examples

- **Reference**
- **Address**
- **Value**
- **Guaranteed?**

<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>72</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?

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

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

in bytes: base + 300 * 1024 * 1024 * 8 + 800 * 1024 * 8 + 2 * 8
= base + 8 * (2 + 1024 * (800 + 1024 * (300)))
= base + 2,523,136,016

Multi-Level Array Example

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};

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

Same thing as a 2D array?

No. 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

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};

Element Access in Multi-Level Array

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*4+dig]

- Computation (IA32)
  - Element access Mem[univ+4*index]+4*8
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

Note: this is how Java represents multi-dimensional arrays.
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>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>

- Code does not do any bounds checking
- Location of each lower-level array in memory is not guaranteed

Using Nested Arrays

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

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
Dynamic Nested Arrays: arrays of pointers to 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)]
```

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
    - parts anywhere in memory

```c
int a[4][5];
int *b[4];
```

Structures

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

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

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

```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:
    ```c
    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

- **Generating Pointer to Structure Member**
  - Offset of each structure member determined at compile time

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

- **Access to Structure Member**
  - 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];
}
```

**IA32 Assembly**

- **Accessing Structure Member**
  - Offset of each structure member determined at compile time

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

- **Generaing Pointer to Structure Member**
  - Offset of each structure member determined at compile time

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

- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time
Structures & Alignment

- Unaligned Data

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

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

![Diagram showing aligned data]

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?

- 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
  - Inserts padding in structure to ensure correct alignment of fields
  - `sizeof()` should be used to get true size of structs
### 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 OSs & instruction sets): lowest 3 bits 000_2
  - Linux: lowest 2 bits of address must be 00_2
  - i.e., treated liked 2 contiguous 4-byte primitive data items

### Struct Alignment Principles

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

\[
K = 8 \quad \text{so} \quad \text{size \ mod \ } 8 = 0
\]

### Saving Space

- **Put large data types first:**

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

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

**Effect (example x86-64, both have K=8)**

```
\begin{array}{c}
p+0 \quad p+8 \quad p+16 \quad p+20 \\
\hline
\text{c} \quad \text{v} \quad \text{i} \quad \text{c}
\end{array}
```

**But actually...**

### Arrays of Structures

- Satisfy alignment requirement for every element
- How would accessing an element work?

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

```
\begin{array}{c}
a[0] \quad a[1] \quad a[2] \quad \ldots
\end{array}
```

```
\begin{array}{c}
a+0 \quad a+16 \quad a+32 \quad a+48
\end{array}
```

```
\begin{array}{c}
\text{v} \quad \text{i} \quad \text{c}
\end{array}
```

```
\begin{array}{c}
a+16 \quad a+24 \quad a+28 \quad a+32
\end{array}
```

external fragmentation
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;
```

```c
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

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

```c
hw_register reg;
reg.byte = 0x3F; // 00111111
reg.bits.b2 = 0; // 00110111
reg.bits.b3 = 0; // 00110011
unsigned short a = reg.byte;
printf("0x%X\n", a); // output: 0x33
```

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