Data: Arrays
Array Allocation

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

![Diagram of array allocation](attachment://array_allocation_diagram.png)
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>3</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>0</td>
</tr>
</tbody>
</table>
Array Accessing Example

- Register %rdi contains starting address of array
- Register %rsi contains array index
- Desired digit at %rdi + 4*%rsi
- Use memory reference (%rdi,%rsi,4)

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

```assembly
# %rdi = z
# %rsi = digit
movq (%rdi,%rsi,4), %rax  # z[digit]
```
Array Loop Example

```c
void zincr(int[] z) {
    int i;
    for (i = 0; i < 5; i++)
        z[i]++;  
}
```

```
# %rdi = z, %rax = i
movl $0, %rax    #   i = 0
jmp .L3          #   goto middle
.L4:  
    addl $1, (%rdi,%rax,4) #   z[i]++
    addq $1, %rax          #   i++
.L3:  
    cmpq $4, %rax          #   i:4
    jbe .L4               #   if <=, goto loop
rep; ret
```
Multidimensional 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

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

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

```
\begin{array}{c|c|c|c}
A[0][0] & \cdots & A[0][C-1] \\
\vdots & \ddots & \vdots \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{array}
```

\[ 4*R*C \text{ bytes} \]
Nested Array Example

```c
int[4][5] zips =
    {{1, 5, 2, 0, 6},
     {1, 0, 0, 0, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

- Variable `zips`: array of 4 elements, allocated contiguously
- Each element is an array of 5 `int`'s, allocated contiguously
- “Row-Major” ordering of all elements in memory
Nested Array Row Access

- Declaration
  
  ```c
  T   A[R][C];
  ```

- Row Vectors
  
  - `A[i]` is an array of `C` elements, e.g. a “row”
  - Each element of type `T` requires `K` bytes
  - Therefore the starting address of row is `A + i * (C * K)`

- Example

  ```c
  int A[R][C];
  ```
Nested Array Row Access Code

- Row Vector
  - \texttt{zips[index]} is array of 5 \texttt{int}'s
  - Starting address \texttt{zips + 20 * index}

- Machine Code
  - Computes and returns address
  - Compute as \texttt{zips + 4 * (index + 4 * index)}

```c
int* get_a_zip(int index)
{
    return zips[index];
}
```

```asm
# %rdi = index
leaq (%rdi,%rdi,4), %rax  # 5 * index
leaq zips(,%rax,4), %rax  # zips + (20 * index)
```
 Nested Array Element Access

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

\[
\text{int } A[R][C];
\]
**Nested Array Element Access Code**

- **Array Elements**
  - \( \text{zips}[\text{index}][\text{dig}] \) is an int
  - Address: \( \text{zips} + 20 \times \text{index} + 4 \times \text{dig} \)
    - Expressed in assembly as \( \text{zips} + 4 \times (5 \times \text{index} + \text{dig}) \)

```c
int get_zip_digit(int index, int dig){
    return zips[index][dig];
}
```

```
leaq (%rdi,%rdi,4), %rax  # 5 * index (%rdi is index)
addl %rax, %rsi           # 5 * index + dig
movl zips(,%rsi,4), %rax  # M[zips + 4*(5 * index + dig)]
```
Multi-Level Array Example

- Variable `univ` denotes an array of 3 elements
- Each element is a pointer (8 bytes)
- Each pointer points to an array of int’s

```c
int* nyu = { 1, 0, 0, 0, 3 };  // nyu[0]
int* mit = { 0, 2, 1, 3, 9 };  // mit[0]
int* ucb = { 9, 4, 7, 2, 0 };  // ucb[0]

#define UCOUNT 3
int* univ[UCOUNT] = {mit, nyu, ucb};
```
Element Access in Multi-Level Array

int uni_digit(int idx, int digit) {
    return univ[idx][digit];
}

// %rdi = idx, %rsi = digit
leaq (%rsi,4), %rsi            # 4*digit
addq univ(,%rdi,8), %rsi        # p = univ[idx] + 4*digit
movl (%rsi), %rax               # return *p
ret

- Computation
  - Element access Mem[Mem[univ+8*idx]+4*digit]
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array
Array Element Accesses

Nested array

```c
int get_zip_digit(size_t index, size_t digit) {
    return zips[index][digit];
}
```

Multi-level array

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

Accesses looks similar in C, but address computations very different:

```
Mem[zips+20*index+4*digit]  
Mem[Mem[univ+8*index]+4*digit]
```
Data: Structs
Structure represented as block of memory
- Big enough to hold all of the fields

Fields ordered according to declaration
- Even if another ordering could yield a more compact representation

Compiler determines overall size + positions of fields
- Machine-level program has no understanding of the structures in the source code
Generating Pointer to Structure Member

- Generating pointer to array element
  - Offset of each structure member determined at compile time
  - Compute as \( r + 4 \times \text{idx} \)
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;
    int i[2];
    double v;
} *p;
```
Alignment Principles

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
  - Required on some machines; advised on x86-64

- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans word boundaries

- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment (x86-64)

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

- **2 bytes**: `short, ...`
  - lowest 1 bit of address must be $0_2$

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

- **8 bytes**: `double, long, char *, ...`
  - lowest 3 bits of address must be $000_2$
Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each 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:**
  - $K = 8$, due to `double` element
Meeting Overall Alignment Requirement

- For largest alignment requirement \( K \)
- Overall structure must be multiple of \( K \)

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

Multiple of \( K = 8 \)
Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Saving Space (Struct packing)

- Put large data types first

```c
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

- Effect (K=12 -> K=4)

```c
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

```plaintext
<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>c</td>
<td>d</td>
<td></td>
<td>2 bytes</td>
</tr>
</tbody>
</table>
```