Data: Arrays
### Array Allocation

- **Basic Principle**

  ```
  T A[N];
  ```

- Array of data type $T$ and length $N$
- Contiguously allocated region of $N \times \text{sizeof}(T)$ bytes in memory

```plaintext
char string[12]; // x to x + 12

int val[5]; // x to x + 20

double a[3]; // x to x + 24

char *p[3]; // x to x + 24
```
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>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>
<tr>
<td>val + i</td>
<td>int*</td>
<td>x + 4 i</td>
</tr>
</tbody>
</table>
Array Accessing Example

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

int[] nyu;
```

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

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

```assembly
# %rdi = z, %rax = i
movl $0, %rax  # i = 0
jmp .L3        # goto middle
.L4:           # loop:
    addl $1, (%rdi,%rax,4) # z[i]++
    addq $1, %rax          # i++
.L3:            # middle
    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

---

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

| \( A[0][0] \) | \( \cdots \) | \( A[0][C-1] \) |
|\( \vdots \) | \( \vdots \) | \( \vdots \) |
|\( A[R-1][0] \) | \( \cdots \) | \( A[R-1][C-1] \) |

4*\( R \times C \) Bytes
Nested Array Example

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

```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 }};
```
Nested Array Row Access

- **Declaration**

  ```
  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**

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

![Diagram showing how to access row vectors in a nested array](image)
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 \times (index + 4 \times index)}
Nested Array Element Access

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

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

$A+(i*C*4)$

$A+(i*C*4)+(j*4)$

$A+(R-1)*C*4)$
**Nested Array Element Access Code**

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

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

```assembly
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 array of int’s

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

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

### Computation

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

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

```assembly
salq $2, %rsi          # 4*digit
addq univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
movl (%rsi), %rax      # return *p
ret
```
### 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 look 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

```c
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```
Generating Pointer to Structure Member

- Generating pointer to array element
  - Offset of each structure member determined at compile time
  - Compute as \( r + 4 \times \text{id} \)

```c
struct rec {  
    int a[4];  
    int i;  
    struct rec* next;  
};

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

leaq (%rdi,%rsi,4), %rax  # r in %rdi, idx in %rsi  
ret  # move ptr into %rax, return
```
Structures & Alignment

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

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

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
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
    - Virtual memory trickier when you allow unaligned data.

- **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

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Meeting Overall Alignment Requirement

- For largest alignment requirement $K$
- Overall structure must be multiple of $K$

```
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```
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];
```
Accessing Members in Array of Struct

- **Compute array offset** \(12 \times \text{idx} \)
  - \(\text{sizeof}(i) + \text{sizeof}(v) + \text{sizeof}(j) + 4\) bytes  
    \(== \text{sizeof}(S3) == 12\) bytes (4 bytes is the padding)

- **Element j is at an offset of 8 bytes within structure**
  - \(\text{sizeof}(i) + 2\) bytes + \(\text{sizeof}(\text{float})\)  
    \(== 8\) bytes

```c
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```
Saving Space (Struct packing)

- Put large data types first

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

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

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

<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></td>
<td>c</td>
<td>d</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>