Array Allocation

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

Array Access

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

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

![Array Example](image)

- Example arrays were allocated in successive 20-byte blocks
- Not guaranteed to happen in general

Array Accessing Example

- `int get_digit(int digit)`
  - Register `rdi` contains starting address of array
  - Register `rsi` contains array index
  - Desired digit at `rdi = digit`
  - Use memory reference (rdi, rsi)

```
Array Loop Example

void sum(int& digit)
{
    int i;
    for( i = 0; i < ZLEN; i++)
    {
        int val = A[i];
        int sum = 0;
        for( int k = 0; k < 5; k++)
        {
            sum += val[k];
        }
        digit = sum;
    }
}
```

Nested Array Row Access Code

- **Row Vector**
  - `pgh`: Variable `pgh`, array of 6 elements, allocated contiguously
  - Each element is an array of 5 `int's`, allocated contiguously
  - "Row-Major" ordering of all elements in memory

```
#define ZLEN 5
int A[5][5] = { {0, 1, 2, 3, 4},
                {5, 6, 7, 8, 9},
                {10, 11, 12, 13, 14},
                {15, 16, 17, 18, 19},
                {20, 21, 22, 23, 24}
};
```
Nested Array Element Access

- **Array Elements**
  - `A[i][j]` is an element of type `C`, which requires `K` bytes.
  - Address `A + i*C + j*K`.

- **Element Access**
  - Accesses look similar in C, but address computations very different:
    - `Mem[pgh + 2*index + 4*digit] + 4*j`.
  - Must perform integer multiplication
  - `n` is a pointer to row array
  - `n+i` is a pointer to array

- **Computations**
  - Element access requires `n+variable*index+4*digit`
  - Must do two memory reads
  - First get pointer to row array
  - Then access element within array

16 x 16 Matrix Access

- **Array Elements**
  - Address `A + i*C + j*K`.
  - `C=16; K=4`

- **Element Access**
  - Uses `pgh` and `index` to compute address
  - `A[pgh + 4*(i*index + digit)]

- **Nested Array Element Access**
  - Accesses looks similar in C, but address computations very different:
    - `Mem[pgh + 2*index + 4*digit] + 4*j`.
  - Must perform integer multiplication

- **Element Access**
  - Accesses looks similar in C, but address computations very different:
    - `Mem[pgh + 2*index + 4*digit] + 4*j`.
  - Must perform integer multiplication

N x N Matrix Access

- **Array Elements**
  - Address `A + i*C + j*K`.
  - `C=16; K=4`

- **Element Access**
  - Uses `pgh` and `index` to compute address
  - `A[pgh + 4*(i*index + digit)]

- **Nested Array Element Access**
  - Accesses looks similar in C, but address computations very different:
    - `Mem[pgh + 2*index + 4*digit] + 4*j`.
  - Must perform integer multiplication

- **Element Access**
  - Accesses looks similar in C, but address computations very different:
    - `Mem[pgh + 2*index + 4*digit] + 4*j`.
  - Must perform integer multiplication

Multi-Level Array Example

- **Variable `u,v` denotes array of `v` elements**
  - Each element is a pointer
  - `u,v` is a pointer points to array

16 x 16 Matrix Access Example

- **Array Elements**
  - Address `A + i*C + j*K`.
  - `C=16; K=4`

- **Element Access**
  - Uses `pgh` and `index` to compute address
  - `A[pgh + 4*(i*index + digit)]

- **Nested Array Element Access**
  - Accesses looks similar in C, but address computations very different:
    - `Mem[pgh + 2*index + 4*digit] + 4*j`.
  - Must perform integer multiplication

- **Element Access**
  - Accesses looks similar in C, but address computations very different:
    - `Mem[pgh + 2*index + 4*digit] + 4*j`.
  - Must perform integer multiplication

Structure Representation

- **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

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

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

### Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute \( r + 4 \times \text{idx} \)

### Register Value

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdi</td>
<td>i</td>
</tr>
<tr>
<td>rsi</td>
<td>val</td>
</tr>
</tbody>
</table>

### Element i

```
int i = r->i;
```

### Following Linked List

- **C Code**
  ```c
  void set_val(struct rec *r, int val)
  {
      while (r)
      {
          int i = r->i;
          r->a[i] = val;
          r = r->next;
      }
  }
  ```

- **Register Value**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdi</td>
<td>i</td>
</tr>
<tr>
<td>rsi</td>
<td>val</td>
</tr>
</tbody>
</table>

### Element i

- While loop
  ```c
  loop: # loop
      movl 16(%rdi), rdx      # i = M[r+16]
      movl rsi, (%rdi, rdx, 4) # M[r+4*i] = val
      movq 24(%rdi), rdi      # r = M[r+24]
      testq %rdi, %rdi          # Test r
      jne loop                 # if (r != 0) goto loop
  ```