Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment

---

### Array Allocation

- **Basic Principle**
  - $T A\{l\}$
  - Array of data type $T$ and length $l$
  - Contiguously allocated region of $l \cdot \text{sizeof}(T)$ bytes in memory

```
char string[12];  // 12 bytes
int val[5];      // 5 \times 4 = 20 bytes
double a[3];    // 3 \times 8 = 24 bytes
char *p[3];     // 3 \times 4 = 12 bytes
```

---

### Array Access

- **Basic Principle**
  - $T A\{l\}$
  - Array of data type $T$ and length $l$
  - Identifier $A$ can be used as a pointer to array element $0$: Type $T^*$

```
int val[5];

int &val[2];
*(val+1)
int *(val+1)
```

---

### Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = {1, 5, 2, 1, 3};
zip_dig mit = {0, 2, 1, 3, 9};
zip_dig ucb = {9, 4, 7, 2, 0};
```

---

### Array Accessing Example

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

```
x86:

movl (%rdi,%rsi,4), %eax
```

---
**Array Loop Example**

void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ELEN; i++)
    z[i]++;
}

```c
#define PCOUNT 4

int pgh[PCOUNT] = {
  (1, 5, 2, 0, 6),
  (1, 5, 2, 1, 3),
  (1, 5, 2, 1, 7),
  (1, 5, 2, 1, 0)};

int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

**Nested Array Example**

```c
#define PCOUNT 4

int pgh[PCOUNT] = {
  (1, 5, 2, 0, 6),
  (1, 5, 2, 1, 3),
  (1, 5, 2, 1, 7),
  (1, 5, 2, 1, 0)};
```

- “**zip_dig pgh[4]**” equivalent to “**int pgh[4][5]**”
  - Variable pgh: array of 4 elements, allocated contiguously
  - Each element is an array of 5 ints, allocated contiguously
- “**Row-Major**” ordering of all elements in memory

**Multidimensional (Nested) Arrays**

- **Declaration**
  - \(A[R][C]\)
  - 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
- **Row** \(A[R][C]\)
  \[ \begin{array}{c}
  A[0][0] \cdots \ A[0][C-1] \\
  \vdots \\
  A[R-1][0] \cdots \ A[R-1][C-1] 
  \end{array} \]

**Nested Array Row Access**

- **Row Vectors**
  - \(A[i]\) is array of \(C\) elements
  - Each element of type \(T\) requires \(K\) bytes
  - Starting address \(A + i \times (C \times K)\)

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

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

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

**Nested Array Element Access**

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

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

```c
int get_pgh_digit(int index, int dig)
    return pgh[index][dig];
```

**Array Elements**

- `pgh[index][dig]` is int
- Address: `pgh + 20*index + 4*dig`
- `pgh[5*index + dig]`

**Nested Array Example**

```c
#define UCOUNT 3
int *univ[UCOUNT] = {mit, unm, ucb};
```

**Element Access in Multi-Level Array**

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

**Computation**

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

**16 X 16 Matrix Access**

- Address `A + j*(C*K) + j*K`
- `C = 16, K = 4`

```c
/* Get element a[i][j] */
int fix_matrix(size_t n, size_t i, size_t j)
    return a[i][j];
```

**Fixed dimensions**

- Know value of `N` at compile time

**Variable dimensions, explicit indexing**

- Traditional way to implement dynamic arrays

**Variable dimensions, implicit indexing**

- *Now supported by gcc*
n X n Matrix Access

- Array Elements
  - Address A + i*[C*K] + j*K
  - C = n, k = 4
  - Must perform integer multiplication

```c
/* Get element a[i][j] */
int var_ele (size_t n, int a[n][n], size_t i, size_t j)
{
    return a[i][j];
}
```

Today

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment

Structure Representation

```
struct rec {
    int a[4];
    size_t 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*idx

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

Following Linked List

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

Unaligned & Alignment

- Unaligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
- Aligned Data
  - Multiple of 4
  - Multiple of 8
  - Multiple of 16
  - Multiple of 32
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 quad word boundaries
  - Virtual memory trickier when datum spans 2 pages
- **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:
- **4 bytes: int, float, ...**
  - lowest 2 bits of address must be 00:
- **8 bytes: double, long, char *, ...**
  - lowest 3 bits of address must be 000:
- **16 bytes: long double (GCC on Linux)**
  - lowest 4 bits of address must be 0000:

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement $K$
  - $K = \text{largest alignment of any element}$
  - 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 $K$

Arrays of Structures

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

Accessing Array Elements

- **Compute array offset $12*\text{idx}$**
- **sizeof(S3), including alignment spacers**
- **Element j is at offset 8 within structure**
- **Assembler gives offset a+b**
  - Resolved during linking
Saving Space

- Put large data types first

\[
\text{struct S4} \begin{cases}
\text{char c; } \\
\text{int i; } \\
\text{char d; }
\end{cases} *p;
\]

\[
\text{struct S5} \begin{cases}
\text{int i; } \\
\text{char c; } \\
\text{char d; }
\end{cases} *p;
\]

- Effect (K=4)

Summary

- Arrays
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements

- Structures
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possible require internal and external padding to ensure alignment

- Combinations
  - Can nest structure and array code arbitrarily