Today

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

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

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 \( T^* \)

Array Example

```c
#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

```x86
int get_digit (int z, int digit) {
  return z[digit];
}
```

```c
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+1</td>
<td>int*</td>
<td>x</td>
</tr>
<tr>
<td>awal[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>5</td>
</tr>
<tr>
<td>val+i</td>
<td>int*</td>
<td>x+4</td>
</tr>
</tbody>
</table>

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

```c
void zip_func(char *zip_str) {
    size_t i = 0;
    for (i = 0; i < ELEN; i++)
        z[i] +=;
}
```

```c
#define PCOUNT 4
int zip_func(pgh[PCOUNT] =
{(1, 5, 2, 0, 6),
(1, 5, 2, 1, 3),
(1, 5, 2, 1, 7),
(1, 5, 2, 1)});
```

```c
“zip_func” is 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
  ```c
  T A[R][C];
  ```
  - 2D array of data type T
  - R rows, C columns
  - Type T element requires K bytes
- Array Size
  ```c
  * R * C * K bytes
  ```
- Arrangement
  ```c
  Row-Major Ordering
  ```

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

```c
A[i][j]
```

```c
A[0][0] ... A[0][C-1] ... A[R-1][0] ... A[R-1][C-1]
```

```c
Nested Array Example
```

```c
#define PCOUNT 4
zip_func(pgh[PCOUNT] =
{(1, 5, 2, 0, 6),
(1, 5, 2, 1, 3),
(1, 5, 2, 1, 7),
(1, 5, 2, 1)});
```

```c
“zip_func” 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
```

```c
Nested Array Row Access Code
```

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

```c
# trdi = index
lea32 (trdi, rdi, 4), trax # S * index
lea32 pgh,[trax,4], trax # pgh + (20 * index)
```

```c
Row Vector
- pgh[index] is array of 5 int's
- Starting address pgh+20*index
```

```c
Machine Code
- Computes and returns address
- Computes as pgh + 4*(index+4*index)
```

```
Nested Array Row Access
```

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

```c
A[i][j]
```

```c
A[0][0] ... A[0][C-1] ... A[R-1][0] ... A[R-1][C-1]
```

```c
Nested Array Element Access
```

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

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

```c
A[i][j]
```

```c
A[0][0] ... A[0][C-1] ... A[R-1][0] ... A[R-1][C-1]
```

```c
A + (i * C * K) + (j * K)
```

```c
A + (i * C * K) + (j * K)
```

```c
A + (i * C * K) + (j * K)
```
N X N Matrix Code

- 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

#define N 16
typedef int fix_matrix[N][N]; /* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}

#define IDX(n, i, j) {{i}*n+j}
/* Get element a[i][j] */
int vec_ele(size_t n, int *a, size_t i, size_t j) {
  return a[IDX(n, i, j)];
}

/* 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];

Multi-Level Array Example

- Variable univ denotes array of 3 elements
- Each element is a pointer
- Each pointer points to array of int's

#define UCOUNT 3
typedef size_t univ[UCOUNT];

int *univ[UCOUNT] = {mit, unm, ucb};

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

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

/* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}

16 X 16 Matrix Access

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

/* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}

#define N 16
typedef int fix_matrix[N][N]; /* Get element a[i][j] */
int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}
**n x n Matrix Access**

- **Array Elements**
  - Address A + i*K + j*K
  - C = n^2
  - Must perform integer multiplication

/* Get element a[i][j] */

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

# n in %rdi, a in %rsi, i in %rdx, j in %rcx

```
imulq %rdx, %rdi # n*i
leaq (%rsi,%rdi,4), %rax # a + 4*n*i
movl (%rax,%rcx,4), %eax # a + 4*n*i + 4*j
ret
```

---

**Today**

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

---

**Structure Representation**

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

- 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**

```
int *get_ap (struct rec *r, size_t idx)
{
    return &r + 4*idx;
}
```

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as r + 4*idx

**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**
  - Multiple of 4
  - Multiple of 8

---

**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;
    }
}
```

---

**Unaligned & Alignment**

```
c [ 0 ] [ 1 [ ] ] v
```

- Multiple of 4
- Multiple of 8
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 0

- **8 bytes: double, long, char *, ...**
  - lowest 3 bits of address must be 0

- **16 bytes: long double (GCC on Linux)**
  - lowest 4 bits of address must be 0

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

Arrays of Structures

- **Overall structure length**
  - multiple of $K$

- **Satisfy alignment requirement**
  - for every element

Meeting Overall Alignment Requirement

- **For largest alignment requirement $K$**

Accessing Array Elements

- **Compute array offset 12*idx**
  - size of $\langle S3 \rangle$, 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

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

**Effect (K=4)**

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

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

**Today**

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

**Background**

- History
  - x87 FP
    - Legacy, many weird features
  - SSE FP
    - Special case use of vector instructions
  - AVX FP
    - Newest version
    - Similar to SSE
    - Documented in book

**Programming with SSE3**

- **XMM Registers**
  - 16 total, each 16 bytes
  - 16 single-byte integers
  - 8 16-bit integers
  - 4 32-bit integers
  - 4 single-precision floats
  - 2 double-precision floats
  - 1 single-precision float
  - 1 double-precision float

**Scalar & SIMD Operations**

- **Scalar Operations: Single Precision**
  - addps %xmm0, %xmm1

- **SIMD Operations: Single Precision**
  - addps %xmm0, %xmm1

- **Scalar Operations: Double Precision**
  - addsd %xmm0, %xmm1

**FP Basics**

- Arguments passed in %xmm0, %xmm1, ...
- Result returned in %xmm0
- All XMM registers caller-saved

```c
float fadd(float x, float y) {
    return x + y;
}
```

```c
double dadd(double x, double y) {
    return x + y;
}
```

```c
# x in %xmm0, y in %xmm1
addps %xmm0, %xmm1
```

```c
# x in %xmm0, y in %xmm1
addsd %xmm0, %xmm1
```
FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers
- FP values passed in XMM registers
- Different mov instructions to move between XMM registers, and between memory and XMM registers

```c
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

Other Aspects of FP Code

- Lots of instructions
  - Different operations, different formats, ...
- Floating-point comparisons
  - Instructions ucomiss and ucomisd
  - Set condition codes CF, ZF, and PF
- Using constant values
  - Set XMM0 register to 0 with instruction xorpd %xmm0, %xmm0
  - Others loaded from memory

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
- Floating Point
  - Data held and operated on in XMM registers