Today

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- Structures
  - Allocation
  - Access
  - Alignment

- Floating Point

Array Allocation

**Basic Principle**

\[
T A[i];
\]

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

Array Access

**Basic Principle**

\[
T A[i];
\]

- 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_digit[ZLEN];

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

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

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

Array Accessing Example

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

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

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

```c
void zip_dig(char *cp) {
    size_t i = 0;
    for (; i < ELEN; i++) {
        *cp = i++;
    }
}
```

**Multidimensional (Nested) Arrays**

- **Declaration**
  
  - 2D array of data type `T`
  - `R` rows, `C` columns
  - Type `T` element requires `K` bytes

- **Array Size**
  
  - `R` * `C` * `K` bytes

- **Arrangement**
  
  - Row-Major Ordering

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

**Nested Array Row Access**

- **Row Vectors**
  - `A[i]` is array of `C` elements
  - Each element of type `T` requires `K` bytes
  - Starting address `A + /*C*K`

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

**Exercise: Assembly Code Matching**

```c
/* 1 */
char *cp;
incl (%rax)
if (!*cp) ...
/* 2 */
int i, ary[20];
addw $2, 4(%rbx)
return &ary[i];
/* 3 */
int *p;
cmpb $0x0, (%rdx)
*p++;
/* 4 */
short a2[10];
a2[2] += 2;
```

**Nested Array Example**

```c
#define PCOUNT 4

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

**Nested Array Row Access Code**

```c
int *get_pgh_zip(int index) {
    return pgh[index];
}
```
Nested Array Element Access

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

**Code Example**

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

A[0][0] ... A[0][(C-1)]
A[1][0] ... A[1][(C-1)]

A = (i*4) + (j*4)
```

Multi-Level Array Example

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

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

Element Access in Multi-Level Array

- **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

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

Array Element Accesses

- **Nested array**
  - `int get_pgh_digit( size_t index, size_t digit )`
  - `int get_univ_digit( size_t index, size_t digit )`

- **Multi-level array**
  - `int get_pgh_digit( size_t index, size_t digit )`
  - `int get_univ_digit( size_t index, size_t digit )`

```
 /* 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 Code

- **Fixed dimensions**
  - Know value of $N$ at compile time

```
#define N 16
typedef int fix_matrix[N][N];
```

- **Variable dimensions, explicit indexing**
  - Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+j)
```

- **Variable dimensions, implicit indexing**
  - Now supported by gcc

```
/* 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];
}
16 X 16 Matrix Access

**Array Elements**
- Address: \( A + j \times (C \times K) + i \times K \)
- \( C = 16, K = 4 \)

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

```asm
# a in rdi, i in rsi, j in rdx
salq $6, rsi  # 64*i
addq trsi, rdi  # a + 64*i
movl (rdi,rdx,4), eax  # M[a + 64*i + 4*j]
ret
```

---

n X n Matrix Access

**Array Elements**
- Address: \( A + j \times (C \times K) + i \times K \)
- \( C = n, K = 4 \)

```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];
}
```

```asm
# 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

- 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;
};
```

```asm
# r in rdi, i in rsi, next in rdx
lea r (rsi,rdi,4), rax  # a + 4*i
movq r (rdx,rdi,4), rax  # M[r + 4*i]
leaq (rdi,rdx,4), rax  # r + 4*i
ret
```

---

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;
    }
}
```

```asm
.L1:
    # loop
    movslq 16(%rdi), trax  # i = M[r+16]
    movl %esi, (rdi,rdax,4)  # M[r+4*i] = val
    movq 24(%rdi), trdi  # r = M[r+24]
    intestq trdi, trdi  # Test r
    jne .L1  # if !=0 goto loop
```
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 $K$

```
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 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:

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

```
struct S2 {
    double v;
    int i[2];
    char c;
} *a[10];
```
### Accessing Array Elements
- Compute array offset 12*idx
  - sizeof(S3), including alignment spacers
- Element j is at offset 8 within structure
-Assembler gives offset a+8
  - Resolved during linking

```c
struct S3 {
    short i;
    float v;
    short j;
    a[10];
};
```

```asm
short get_j(int idx)
{
    return a[idx].j;
}
```

```asm
leaq (%rdi,%rdi,2),%rax # 3*idx
movzl %rax,a+8(%rax,4),%eax
```

### Saving Space
- Put large data types first

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

### 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**
  - `addss %xmm0, %xmm1`  
  - `%xmm0`  
  - `%xmm1`

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

- **Scalar Operations: Double Precision**
  - `addsd %xmm0, %xmm1`  
  - `%xmm0`  
  - `%xmm1`
**FP Basics**
- Arguments passed in %xmm0, %xmm1, ...
- Result returned in %xmm0
- All XMM registers caller-saved

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

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

- # x in %xmm0, y in %xmm1
  - addss %xmm1, %xmm0
  - ret

- # x in %xmm0, y in %xmm1
  - added %xmm1, %xmm0
  - ret

**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

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

- # p in rdi, v in %xmm0
  - movapd %xmm0, %xmm1 # Copy v
  - movsd (%rdi), %xmm0 # x = *p
  - added %xmm0, %xmm1 # t = x + v
  - movsd %xmm1, (%rdi) # *p = t
  - ret

**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