Machine-Level Programming IV: Data
Next ...

- 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 $L \times \text{sizeof}(T)$ bytes in memory

char string[12];

char *p[3];
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];
```

<table>
<thead>
<tr>
<th>C 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>5</td>
</tr>
<tr>
<td>val+i</td>
<td>int *</td>
<td>x+4i</td>
</tr>
</tbody>
</table>
Array Example

```c
#define ALEN 5
typedef int int_array[ALEN];

int_array cmu = { 1, 5, 2, 1, 3 };
int_array mit = { 0, 2, 1, 3, 9 };
int_array bu = { 0, 2, 2, 1, 5 };
```

- Declaration “int_array cmu” equivalent to “int cmu[5]”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array Accessing Example

**Register%rdi** contains starting address of array

**Register%rsi** contains array index

Desired digit at %rdi + 4*%rsi

Use memory reference (%rdi,%rsi,4)

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

```assembly
# %rdi = a
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # a[digit]
```
Array Loop Example

```c
void aincr(int_array a) {
    size_t i;
    for (i = 0; i < ALEN; i++)
        a[i]++;
}
```

`size_t` type defined in stdio.h to be unsigned long

```assembly
# %rdi = a
movl $0, %eax # i = 0
jmp .L3 # goto middle
.L4:
    addl $1, (%rdi,%rax,4) # a[i]++
    addq $1, %rax # i++
.L3:
    cmpq $4, %rax # i:4
    jbe .L4 # if <=, goto loop
rep; ret
```
Multidimensional (Nested) 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

\[
\begin{array}{cccc}
A[0][0] & \cdots & A[0][C-1] \\
\vdots & & \vdots \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{array}
\]

\[
\begin{array}{cccc}
A[0] & \cdots & A[0][C-1] \\
A[R-1] & \cdots & A[R-1][C-1]
\end{array}
\]

\[
\begin{array}{cccc}
A[0][0] & \cdots & A[0][C-1] \\
\vdots & & \vdots \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{array}
\]

\[
\begin{array}{cccc}
A[0] & \cdots & A[0][C-1] \\
A[R-1] & \cdots & A[R-1][C-1]
\end{array}
\]
Nested Array Example

- "int_array a[4]" equivalent to "int a[4][5]"
  - Variable a: 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
Nested Array Row Access

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

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

```plaintext
\[ \begin{array}{cccc}
& | & | & |
A & A[0] & \cdots & A \[C-1] \\
A[0] & & & A[0] \\
& \cdots & \cdots & \cdots \\
\end{array} \]
\[ \begin{array}{cccc}
& | & | & |
A & A[i] & \cdots & A \[C-1] \\
& \cdots & \cdots & \cdots \\
\end{array} \]
\[ \begin{array}{cccc}
& | & | & |
A & A[R-1] & \cdots & A \[C-1] \\
& \cdots & \cdots & \cdots \\
\end{array} \]
\[ A + (i \cdot C \cdot 4) \]
\[ A + ((R-1) \cdot C \cdot 4) \]
Nested Array Row Access Code

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

- **Machine Code**
  - Computes and returns address
  - Compute as `a + 4*(index+4*index)`

```c
int *get_a_array(int index) {
    return a[index];
}
```

# %rdi = index
leaq (%rdi,%rdi,4),%rax # 5 * index
leaq a(,%rax,4),%rax # a + (20 * index)
Nested Array Element Access

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

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

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

$A + (i \times C \times 4) + (j \times 4)$
Nested Array Element Access Code

### Array Elements

- **Array Elements**
  - \(a[\text{index}][\text{dig}]\) is int
  - Address: \(a + 20*\text{index} + 4*\text{dig}\)
    - \(= a + 4*(5*\text{index} + \text{dig})\)

```c
int get_a_digit(int index, int dig)
{
    return a[index][dig];
}
```

```assembly
leaq (%rdi,%rdi,4), %rax  # (4+1)*index = 5*index
addl %rax, %rsi          # 5*index+dig
movl a(,%rsi,4), %eax    # M[a + 4*(5*index+dig)]
```
Multi-Level Array Example

```c
int_array cmu = { 1, 5, 2, 1, 3 };  
int_array mit = { 0, 2, 1, 3, 9 };  
int_array bu = { 0, 2, 2, 1, 5 };  

#define UCOUNT 3  
int *univ[UCOUNT] = {mit, cmu, bu};
```

- Variable `univ` denotes an array of 3 elements.
- Each element is a pointer – 8 bytes.
- Each pointer points to an array of `int`'s.
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
Array Element Accesses

Nested array

```c
int get_a_digit(size_t index, size_t digit)
{
    return a[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[a+20*index+4*digit]  Mem[Mem[univ+8*index]+4*digit]
```
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

```c
#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];
}
```
16 X 16 Matrix Access

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

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

```assembly
    # a in %rdi, i in %rsi, j in %rdx
    salq   $6, %rsi       # 64*i
    addq   %rsi, %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 + 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];
}
```

```assembly
# 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
```
Next ...

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

• Structures
  – Allocation
  – Access
  – Alignment
• 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

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time
  - Compute as \( r + 4 \times \text{idx} \)

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

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

```assembly
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```
Following Linked List

• C Code

```c
void set_val
  (struct rec *r, int val)
{
  while (r) {
    size_t 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>r</td>
</tr>
<tr>
<td>%rsi</td>
<td>val</td>
</tr>
</tbody>
</table>

```asm
.L11:  # loop:
  movq  16(%rdi), %rax
  movl  %esi, (%rdi,%rax,4) # M[r+4*i] = val
  movq  24(%rdi), %rdi  # r = M[r+24]
  testq %rdi, %rdi  # Test r
  jne   .L11  # if !=0 goto loop
```

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

### Element i

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>next</td>
<td></td>
</tr>
</tbody>
</table>

Element i

Following Linked List
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 $K$

```c
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

• 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 0₂
- 4 bytes: int, float, ...
  - lowest 2 bits of address must be 00₂
- 8 bytes: double, long, char *, ...
  - lowest 3 bits of address must be 000₂
- 16 bytes: long double (GCC on Linux)
  - lowest 4 bits of address must be 0000₂
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

```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

```c
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 Array Elements

- Compute array offset $12 \times \text{idx}$
  - 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];
```

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

```assembly
  # %rdi = idx
  leaq (%rdi,%rdi,2),%rax  # 3*idx
  movzwl a+8(%rax,4),%eax
```
Saving Space

• 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=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>c</td>
<td>d</td>
<td></td>
<td>2 bytes</td>
</tr>
</tbody>
</table>
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
  – Possibly require internal and external padding to ensure alignment

• Combinations
  – Can nest structure and array code arbitrarily