Machine-Level Programming IV: Data Structures

Kai Shen

Basic Data Types

- Integer
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on software interpretation and manipulation

<table>
<thead>
<tr>
<th>Intel</th>
<th>ASM</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>b</td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>l</td>
<td>4</td>
<td>[unsigned] int</td>
</tr>
<tr>
<td>quad word</td>
<td>q</td>
<td>8</td>
<td>[unsigned] long int (x86-64)</td>
</tr>
</tbody>
</table>

- Floating Point
  - Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Intel</th>
<th>ASM</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>s</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12/16</td>
<td>long double</td>
</tr>
</tbody>
</table>

Outline: Data Structures

- Basic data types
- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Optimizations
- Structures

Array Allocation

T A[l];
- Array of data type T and length L
- Contiguously allocated region of L * sizeof (T) bytes

char string[12];

int val[5];

double a[3];

char *p[3];

Send message to Kai Shen
Array Access

```c
int val[5];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val[4]</code></td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td><code>val</code></td>
<td>int*</td>
<td><code>x</code></td>
</tr>
<tr>
<td><code>val+1</code></td>
<td>int*</td>
<td><code>x+4</code></td>
</tr>
<tr>
<td><code>&amp;val[2]</code></td>
<td>int*</td>
<td><code>x+8</code></td>
</tr>
<tr>
<td><code>val[5]</code></td>
<td>int</td>
<td><code>??</code></td>
</tr>
<tr>
<td><code>*(val+1)</code></td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td><code>val+i</code></td>
<td>int*</td>
<td><code>x+4i</code></td>
</tr>
</tbody>
</table>

Array Accessing Example

```c
zip_dig z:
```

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

IA32

```asm
# edx = z
movl $0, %eax  # %eax = i
.L4:  # loop:
    addl $1, (%edx,%eax,4) # z[i]++
    addl $1, %eax        # i++
    cmpl $5, %eax        # i:5
    jne .L4              # if !=, goto loop
```

Array Loop Example (IA32)

```c
void zincr(zip_dig z) {
    int i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

Nested Array Data Layout

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

IA32

```asm
.zip_dig pgh[4]:
```

```asm
# “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 int’s, allocated contiguously
# “Row-Major” ordering of all elements guaranteed
```
**Nested Array Row Access Code**

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

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

- **Machine code**
  - Computes and returns address
  - Using `lea`, compute as `pgh + 4*(index+4*index)`

```c
#define PCOUNT 4
zip_dig pgh[PCOUNT] = {
    {1, 5, 2, 1, 3},
    {1, 5, 1, 2, 7},
    {1, 5, 2, 2, 1}};
```

```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`

- **Machine code**
  - Computes address `pgh + 4*((index+4*index)+dig)`

```c
movl 8(%ebp), %eax # index
leal (%eax,%eax,4),%eax # 5 * index
addl 12(%ebp), %eax # 5*index+dig
movl pgh(,%eax,4), %eax # offset 4*(5*index+dig)
```

**16 X 16 Matrix Access**

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

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

```c
movl 12(%ebp), %edx # i
sall $6, %edx # i*64
movl 16(%ebp), %eax # j
sall $2, %eax # j*4
addl 8(%ebp), %eax # a + j*4
movl (%eax,%edx), %eax # *(a + j*4 + i*64)
```

**n X n Matrix Access**

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

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

```c
movl 8(%ebp), %eax # n
sall $2, %eax # n*4
movl %eax, %edx # n*4
imull 16(%ebp), %edx # i*n*4
movl 20(%ebp), %eax # j
sall $2, %eax # j*4
addl 12(%ebp), %eax # a + j*4
movl (%eax,%edx), %eax # *(a + j*4 + i*n*4)
```
Outline: Data Structures

- Basic data types
- Arrays
- Structures
  - Allocation
  - Access

Structure Allocation

```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```

- Concept
  - Contiguously-allocated region of memory
  - Refer to members within structure by names
  - Members may be of different types

Structure Access

```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```

- Accessing Structure Member
  - Pointer indicates first byte of structure
  - Access elements with offsets

```
void set_i(struct rec *r, int val) {
  r->i = val;
}
```

IA32 Assembly

```
# %edx = val
# %eax = r
movl %edx, 12(%eax) # Mem[r+12] = val
```