Machine-Level Programming IV: Data

Introduction to Computer Systems
10th Lecture, Oct. 8, 2015

Instructor:
Younghoon Kim
Today

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

- **Structures**
  - Allocation
  - Access
  - Alignment

- **Union**
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];
  ```

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

  ```
  double a[3];
  ```

  ```
  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 \( T^* \)

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

### Reference

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

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

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

IA32

```assembly
movl (%rdi,%rsi,4), %eax # z[digit]
```

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

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

```assembly
# %rdi = z
movl  $0, %eax               # i = 0
jmp   .L3                     # goto middle
.L4:                            # loop:
    addl $1, (%rdi,%rax,4)     # z[i]++
    addq $1, %rax              # i++
.L3:                            # middle
    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

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

![Diagram showing 2D array arrangement]

\[ A[0][0] \ldots \ldots A[0][C-1] \]
\[ \quad \ldots \quad \quad \quad \quad \quad \quad \quad \quad \]
\[ A[R-1][0] \ldots \ldots A[R-1][C-1] \]

\( 4 \times R \times C \) Bytes
Nested Array Example

```
#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 }};
```

- **zip_dig pgh[4]**;
- **“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 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 $A + i \times (C \times K)$

- **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[0][0]  ...  A[0][C-1]
|       |       |
A[1][0]  ...  A[1][C-1]
|       |       |
A[R-1][0]  ...  A[R-1][C-1]
|       |       |
```

$A + (i \times C \times 4)$

$A + ( (R-1) \times C \times 4)$
### Multi-Level Array Example

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

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

#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```
Element Access in Multi-Level Array

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

```
salq $2, %rsi           # 4*digit
addq univ(%rdi,8), %rsi # p = univ[index] + 4*digit
movl (%rsi), %eax      # return *p
ret
```

**Computation**

- Element access \texttt{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_pgh_digit(size_t index, size_t digit)
{
    return pgh[index][digit];
}
```

Multi-level array

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

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

\[
\text{Mem}[\text{pgh}+20*\text{index}+4*\text{digit}] \quad \text{Mem}[\text{Mem}[\text{univ}+8*\text{index}]+4*\text{digit}]
\]
Today

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

- **Structures**
  - Allocation
  - Access
  - Alignment

- **Union**
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

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```
Generating Pointer to Structure Member

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

**Generating Pointer to Array Element**

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

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

```
# 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) {
      int i = r->i;
      r->a[i] = val;
      r = r->next;
   }
}
```

### Element i

<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>

<table>
<thead>
<tr>
<th>Register Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>movslq 16(%rdi), %rax # i = M[r+16]</td>
</tr>
<tr>
<td>movl %esi, (%rdi,%rax,4) # M[r+4*i] = val</td>
</tr>
<tr>
<td>movq 24(%rdi), %rdi # r = M[r+24]</td>
</tr>
<tr>
<td>testq %rdi, %rdi # Test r</td>
</tr>
<tr>
<td>jne .L11 # if !=0 goto loop</td>
</tr>
</tbody>
</table>

struct rec {
  int a[3];
  int i;
  struct rec *next;
};
Structures & Alignment

- **Unaligned Data**

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
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 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 \( K \)
    - \( K \) = Largest alignment of any element
  - Initial address & structure length must be multiples of \( K \)

- **Example:**
  - \( K = 8 \), due to `double` element

```
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;
```

Diagram showing memory allocation and alignment:
- $v$ at $p+0$
- $i[0]$ at $p+8$
- $i[1]$ at $p+16$
- $c$ at $p+24$

Multiple of $K=8$
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*idx
  - `sizeof(S3)`, including alignment spacers
- Element j is at offset 8 within structure
- Assembler gives offset a+8
  - Resolved during linking

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

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

```
# %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;
```

- Effect (K=4)

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

```plaintext
<table>
<thead>
<tr>
<th></th>
<th>3 bytes</th>
<th></th>
<th></th>
<th>3 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
<td>i</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

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

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

- **Structures**
  - Allocation
  - Access
  - Alignment

- **Union**
Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```c
union U1 {
    char c;
    int i[2];
    double v;
} *up;

struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```
Using Union to Access Bit Patterns

```c
typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}
```

Same as (float) u?  
Same as (unsigned) f?
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

- **Unions**
  - Overlay declarations
  - Way to circumvent type system