Assembly IV: Complex Data Types

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Basic Data Types

- **Integer**
  - Stored & operated on in general registers
  - Signed vs. unsigned depends on instructions used

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

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

<table>
<thead>
<tr>
<th></th>
<th>Intel</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>single</td>
<td>s</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>l</td>
<td>8</td>
<td>double</td>
</tr>
</tbody>
</table>
Complex Data Types

- Complex data types in C
  - Pointers
  - Arrays
  - Structures
  - Unions
  - …

- Can be combined
  - Pointer to pointer, pointer to array, …
  - Array of array, array of structure, array of pointer, …
  - Structure in structure, pointer in structure, array in structure, …
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

**Examples:**

- `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

```
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</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 + 4 * i</td>
</tr>
</tbody>
</table>
```
Array Example

typedef int zip_dig[5];

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

- Notes
  - Example arrays were allocated in successive 20 byte blocks
    - Not guaranteed to happen in general
Array Access Example

- Code does not do any bounds checking!
  - Out of range behavior implementation-dependent
  - No guaranteed relative allocation of different arrays

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit[3]</td>
<td>36 + 4 * 3 = 48</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit[5]</td>
<td>36 + 4 * 5 = 56</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>mit[-1]</td>
<td>36 * 4 * (-1) = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4 * 15 = 76</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>
Array Access Code

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

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

**Memory Reference Code**

```assembly
# %rdi = z
# %esi = digit

movslq  %esi, %rsi
movl   (%rdi,%rsi,4),%eax
```
Array Loop Example

- Registers
  - `%rdi` z
  - `%rax` i

```c
void zincr(zip_dig z) {
    int i;
    for (i = 0; i < 5; i++)
        z[i]++;
}
```

```
# %rdi = z
movl $0, %eax  # i = 0
jmp .L3
.L4
    addl $1, (%rdi,%rax,4)  # z[i]++
    addq $1, %rax  # i++
.L3
    cmpq $4, %rax  # compare i:4
    jle .L4  # if <=, goto loop
ret
```
### Nested Array

**Declaration:** \( T \ A[R][C] \);
- 2D array of data type \( T \)
- \( R \) rows, \( C \) columns
- Array size = \( R \times C \times \text{sizeof}(T) \)

**Arrangement**
- Row-major ordering

<table>
<thead>
<tr>
<th>( A[0][0] )</th>
<th>( \cdots )</th>
<th>( A[0][C-1] )</th>
<th>( A[1][0] )</th>
<th>( \cdots )</th>
<th>( A[1][C-1] )</th>
<th>( \cdots )</th>
<th>( A[R-1][0] )</th>
<th>( \cdots )</th>
<th>( A[R-1][C-1] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4<em>R</em>C Bytes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Nested Array Example

- C code
  - Variable `pgh` denotes array of 4 elements
    - Allocated contiguously
  - Each element is an array of 5 int’s
    - Allocated contiguously

- Row-major ordering of all elements guaranteed

```c
int pgh[4][5] = {
  {1, 5, 2, 0, 6},
  {1, 5, 2, 1, 3 },
  {1, 5, 2, 1, 7 }},
  {1, 5, 2, 2, 1 }};
```
Nested Array Access Example

- Code does not do any bounds checking! (again)
  - Ordering of elements within array guaranteed

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pgh[3][3]</code></td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][5]</code></td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][-1]</code></td>
<td>76+20<em>2+4</em>(-1) = 112</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[4][-1]</code></td>
<td>76+20<em>4+4</em>(-1) = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][19]</code></td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][-1]</code></td>
<td>76+20<em>0+4</em>(-1) = 72</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>
Nested Array Row Access

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

int \( A[R][C] \);

```
A
```

```
A[\theta]
```

```
A[i]
```

```
A[R-1]
```

```
A[\theta][\theta]
```

```
A[i][i]
```

```
A[R-1][R-1]
```

```
A[\theta][C-1]
```

```
A[i][C-1]
```

```
A[R-1][C-1]
```

```
A+(i*C*4)
```

```
A+((R-1)*C*4)
```
Nested Array Row Access Code

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

- **Code**
  - Compute and return address
  - Compute as `pgh + 4 * (index + 4 * index)`

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

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

- Array elements
  - $A[i][j]$ is element of type $T$
  - Address: $A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K$

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

![Diagram showing nested array element access]
Nested Array Element Access Code

- **Array elements**
  - \( pgh[index][digit] \) is int
  - Address: \( pgh + 20 \times index + 4 \times digit \)

- **Code**
  - Compute address \( pgh + 4 \times ((index + 4 \times index) + digit) \)
  - `movl` performs memory reference

```assembly
  # %rdi = index
  # %rsi = digit
  leaq (%rdi,%rdi,4),%rax     # 5 * index
  addl %rax, %rsi             # 5 * index + digit
  movl pgh(,%rsi,4),%eax     # M[pgh+4*(5*index+digit)]
```

```c
  int get_pgh_digit
  (long index, long digit) {
    return pgh[index][digit];
  }
```
Multi-Level Array

**Example**

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

```c
typedef int zip_dig[5];

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

int *univ[3] = {mit, cmu, ucb};
```
Summary

**Arrays in C**
- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

**Compiler optimizations**
- Compiler often turns array code into pointer code
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops
Structure

- Structure represented as block of memory
  - Refer to members within structure by names
  - Members may be of different type

- Fields ordered according to declaration
  - Even if another ordering could be more compact

- Compiler determines overall size + positions of fields
  - Machine-level program has no understanding of the structure

```
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```
Structure Referencing Example

- 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];
    long i;
    struct rec *next;
};

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

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

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

void set_val (struct rec *r, int val)
{
    while (r) {
        long i = r->i;
        r->a[i] = val;
        r = r->next;
    }
}
```

```assembly
.L11:
    movq 16(%rdi),%rax
    movl %esi,(%rdi,%rax,4)
    movq 24(%rdi),%rdi
    testq %rdi,%rdi
    jne .L11
```

# rdi = r
# esi = val
Structure & 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
    - Inefficient to load or store data that spans quad word boundaries
    - Virtual memory trickier when data spans 2 pages

- **Compiler**
  - Inserts gaps (or “pads”) in structure to ensure correct alignment of fields
Specific Alignment Cases (x86-64)

- **1 byte**: char, …
  - No restrictions on address

- **2 bytes**: short, …
  - Lowest 1 bit of address must be $0_2$

- **4 bytes**: int, float, …
  - Lowest 2 bits of address must be $00_2$

- **8 bytes**: long, double, char *, …
  - Lowest 3 bits of address must be $000_2$
Satisfying Alignment in Structures

- Offsets 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 multiple of $K$

- Example: $K = 8$ due to `double` element

```
<table>
<thead>
<tr>
<th></th>
<th>3 bytes</th>
<th>i[0]</th>
<th>i[1]</th>
<th>4 bytes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
<td>p+24</td>
<td></td>
</tr>
</tbody>
</table>
```

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

---

SSE2030: Introduction to Computer Systems | Fall 2017 | Jin-Soo Kim (jinsookim@skku.edu)

25
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;
```

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;
} *p;
```
Saving Spaces

- Put large data types first

```c
struct S3 {
    char c;
    int i;
    char d;
} *p;
```

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

```plaintext

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>c</th>
<th>d</th>
<th>2 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td></td>
</tr>
</tbody>
</table>
```
Union Allocation

- **Principles**
  - Overlay union elements
  - Allocate according to largest element
  - Can only use one field at a time

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

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

Diagram:
- `c` at `p+0`
- `i[0]` at `p+4`
- `i[1]` at `p+8`
- `v` at `p+16`

- 3 bytes for `c`
- 4 bytes for `v`
- 16 bytes for `i[0]` and `i[1]`
Summary

- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment

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