Assembly IV: Complex Data Types

Jinkyu Jeong (jinkyu@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>Intel</th>
<th>GAS</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 (x86-64)</td>
</tr>
</tbody>
</table>

• Floating point
  – Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Intel</th>
<th>GAS</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>
</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: $T \text{ A}[L]$;
  – Array of data type $T$ and length $L$
  – Contiguously allocated region of $L \times \text{sizeof}(T)$ bytes
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

```plaintext
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

```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 }; 
```

- **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

| Reference | Address       | Value | Guaranteed?
|-----------|---------------|-------|--------------
| mit[3]    | 36 + 4 * 3 = 48 | 3     | Yes          
| mit[5]    | 36 + 4 * 5 = 56 | 9     | No           
| mit[-1]   | 36 * 4 * (-1) = 32 | 3     | No           
| cmu[15]   | 16 + 4 * 15 = 76 | ??    | No           

Reference Address Value Guaranteed?
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_dig z, int digit)
{
    return z[digit];
}
```

```c
# %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
.L3
addl $1, (%rdi,%rax,4) # z[i]++
addq  $1, %rax     # i++
.L4
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

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

\( 4 \times R \times C \) Bytes
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

```
int pgh[4][5] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
```

```
int pgh[4][5];
```
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>pgh[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td>76+20<em>2+4</em>(-1) = 112</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td>76+20<em>4+4</em>(-1) = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][-1]</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: $A + i \times (C \times K)$

```c
int A[R][C];
```
Nested Array Row Access Code

• Row vector
  – \texttt{pgh[index]} is array of 5 int’s
  – Starting address: \texttt{pgh + 20 * index}

• Code
  – Compute and return address
  – Compute as \texttt{pgh + 4 * (index + 4 * index)}

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

```assembly
# %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: \( A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K \)

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

```
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*C*4)

A+(i*C+j)*4

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

• Array elements
  – pgh[index][digit] is int
  – Address: pgh + 20 * index + 4 * digit

• Code
  – Compute address pgh + 4 * ((index + 4 * index) + digit)
  – movl performs memory reference

```
int get_pgh_digit
    (long index, long digit) {
        return pgh[index][digit];
    }
```

```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)]
```
Multi-Level Array

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

```
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

```c
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 \( r + 4 \times \text{id}x \)

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

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

![Diagram showing structure referencing example](image)
Following Linked List Example

```c
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;
    }
}
```

# rdi = r
# esi = val

.L11:
    movq 16(%rdi),%rax
    movl %esi,(%rdi,%rax,4)
    movq 24(%rdi),%rdi
    testq %rdi,%rdi
    jne .L11
Structure & Alignment

- Unaligned data

- 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
    • 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 $K$
    • $K = \text{Largest alignment of any element}$
  – Initial address & structure length must be multiple of $K$

• Example: $K = 8$ due to double element

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 bytes</td>
<td>4 bytes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
p+0 \quad p+4 \quad p+8 \quad p+16 \quad p+24
\]

- Multiple of 4
- Multiple of 8
- Multiple of 8
- Multiple of 8
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;
```

$v$ $i[0]$ $i[1]$ $c$ 7 bytes

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

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

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

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

• Structures
  – Allocate bytes in order declared
  – Pad in middle and at end to satisfy alignment

• Unions
  – Overlay declarations
  – Way to circumvent type system