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
    - Intel  
    - GAS  
    - Bytes  
    - C  
    - byte  
    - b  
    - 1  
    - [unsigned] char  
    - word  
    - w  
    - 2  
    - [unsigned] short  
    - double word  
    - l  
    - 4  
    - [unsigned] int

- **Floating point**
  - Stored & operated on in floating point registers
    - Intel  
    - GAS  
    - Bytes  
    - C  
    - Single  
    - s  
    - 4  
    - float  
    - Double  
    - l  
    - 8  
    - double  
    - Extended  
    - t  
    - 10/12  
    - long double
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

```
char string[12];
int val[5];
double a[4];
char *p[3];
```

- char string[12]
- int val[5]
- double a[4]
- char *p[3]

Diagram showing allocation and indexing of array elements.
Array Access

- **Basic principle:** $T \ A[L]$;
  - Array of data type $T$ and length $L$
  - Identifier $A$ can be used as a pointer to 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

\[
\text{typedef int zip_dig[5];}
\]

\[
\text{zip_dig cmu = \{ 1, 5, 2, 1, 3 \};}
\]

\[
\text{zip_dig mit = \{ 0, 2, 1, 3, 9 \};}
\]

\[
\text{zip_dig ucb = \{ 9, 4, 7, 2, 0 \};}
\]

- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array Accessing Example (1)

- Computation
  - Register %edx contains starting address of array
  - Register %eax contains array index
  - Desired digit at 4 * %eax + %edx
  - Use memory reference: (%edx,%eax,4)

Memory Reference Code

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

```asm
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```
Array Accessing Example (2)

- Code does not do any bounds checking!

  - 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

- Out of range behavior implementation-dependent
- No guaranteed relative allocation of different arrays
Array Loop Example (1)

- Original source

- Transformed version
  - As generated by GCC
  - Eliminate loop variable i
  - Convert array code to pointer code
  - Express in do-while form
    - No need to test at entrance

int zd2int(zip_dig z){
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++)
        zi = 10 * zi + z[i];
    return zi;
}

int zd2int(zip_dig z){
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
Array Loop Example (2)

- Registers
  - `%ecx` : `z`
  - `%eax` : `zi`
  - `%ebx` : `zend`

```c
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

```
# %ecx = z
xorl %eax,%eax
leal 16(%ecx),%ebx
.L59:
    leal (%eax,%eax,4),%edx  # 5*zi
    movl (%ecx),%eax         # *z
    addl $4,%ecx             # z++
    leal (%eax,%edx,2),%eax  # zi = *z + 2*(5*zi)
    cmpl %ebx,%ecx           # z : zend
    jle .L59                 # if <= goto loop
```

- Register increments by 4
- `10 * zi + *z = *z + 2*(zi+4*zi)`
Nested Array (1)

- 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

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

$$4 \times R \times C \text{ Bytes}$$
Nested Array (2)

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

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

- **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 |
```

Row-major ordering:
- Starting from the top left corner:
  - 1
  - 5
  - 2
  - 0
  - 6
  - 1
  - 5
  - 2
  - 1
  - 3
  - 1
  - 5
  - 2
  - 2
  - 1
- Allocated contiguously.
### Nested Array Access (1)

- **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];
```

![Diagram of array access](image.png)
Nested Array Access (2)

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

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

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

```assembly
# %eax = index
leal (%eax,%eax,4),%eax # 5 * index
leal pgh(,%eax,4),%eax # pgh + (20 * index)
```
Nested Array Access (3)

- **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];
```

```
\[
\begin{array}{ccc}
A[0] & \cdots & A[i] \\
[0] & \cdots & [i][j] \\
[0] & \cdots & [R-1][0] \\
[A] & \cdots & [A(R-1)][C-1] \\
\end{array}
\]
```

- $A + i \times C \times 4$
- $A + (i \times C + j) \times 4$
- $A + (R-1) \times C \times 4$
**Nested Array Access (4)**

- **Array Elements**
  - `pgh[index][dig]` is int
  - Address: 
    \[ pgh + 20 \times index + 4 \times dig \]

- **Code**
  - Computes address `pgh + 4*dig + 4*(index+4*index)`
  - `movl` performs memory reference

```c
int get_pgh_digit
   (int index, int dig)
{
    return pgh[index][dig];
}
```

```assembly
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx  # 4*dig
leal (%eax,%eax,4),%eax  # 5*index
movl pgh(%edx,%eax,4),%eax  # *(pgh + 4*dig + 20*index)
```
**Nested Array Access (5)**

- **Strange referencing examples**
  - Code does not do any bounds checking
  - Ordering of elements within array guaranteed

```c
int pgh[4][5];
```

<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>
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
# Structures

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

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

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

**Memory Layout**

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>a</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>16</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
```

**Assembly**

```
# %eax = val
# %edx = r
movl %eax, (%edx)  # Mem[r] = val
```
Structure Referencing (1)

- Generating pointer to structure member
  - Offset of each member determined at compile time

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

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

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```
Structure Referencing (2)

- Generating pointer to member (cont’d)

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

void set_p (struct rec *r) {
    r->p = &r->a[r->i];
}
```

```
# %edx = r
movl (%edx),%ecx     # r->i
leal 0(%ecx),%eax    # 4*(r->i)
leal 4(%eax,%edx),%eax # r+4+4*(r->i)
movl %eax,16(%edx)   # update r->p
```
Alignment (1)

- **Aligned data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
  - Required on some machines; advised on IA-32
    - treated differently by Linux and Windows

- **Motivation for aligning data**
  - Memory accessed by (aligned) double or quad-words
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory very tricky when datum spans 2 pages

- **Compiler**
  - Inserts gaps (or “pads”) in structure to ensure correct alignment of fields
Alignment (2)

- **Size of primitive data type:**
  - 1 byte (e.g., char): No restrictions on address
  - 2 bytes (e.g., short)
    - lowest 1 bit of address must be $0_2$
  - 4 bytes (e.g., int, float, char *, etc)
    - lowest 2 bits of address must be $00_2$
  - 8 bytes (e.g., double)
    - Windows (and most other OS’s & instruction sets): lowest 3 bits of address must be $000_2$
    - Linux: lowest 2 bits of address must be $00_2$ (i.e., treated the same as a 4-byte primitive data type)
  - 12 bytes (long double)
    - Windows, Linux: lowest 2 bits of address must be $00_2$ (i.e., treated the same as a 4-byte primitive data type)
Alignment (3)

- Offsets within structure
  - Must satisfy element’s alignment requirement

- Overall structure placement
  - Each structure has alignment requirement $K$
    - Largest alignment of any element
  - Initial address & structure length must be multiples of $K$

- Example (under Windows):
  - $K = 8$, due to double element

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
### Alignment (4)

#### Linux vs. Windows

- **Windows (including Cygwin):** $K = 8$

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

- **Linux:** $K = 4$

  ```
  struct S1 {
    char c;
    int i[2];
    double v;
  } *p;
  ```
Alignment (5)

- Overall alignment requirement

```c
struct S2 {
    double x;
    int i[2];
    char c;
} *p;
```

- `p` must be multiple of:
  - 8 for Windows
  - 4 for Linux

```c
struct S3 {
    float x[2];
    int i[2];
    char c;
} *p;
```

- `p` must be multiple of 4 (all cases)
Alignment (6)

- Ordering elements within structure

```c
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```

10 bytes wasted space in Windows

```c
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```

2 bytes wasted space
Union Allocation

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

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

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

(Windows alignment)
Summary

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

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