Alignment

Unaligned Data

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>p+1</td>
<td>p+5</td>
<td>p+9</td>
</tr>
</tbody>
</table>

Aligned Data

- Primitive data type requires K bytes
- Address must be a multiple of K

Aligned Data

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

Alignment Principles

**Aligned Data**
- Primitive data type requires K bytes
- Address must be a multiple of K

**Motivation for Aligning Data**
- Memory accessed by (aligned) chunks of 4, 8 or more bytes (system dependent)
- It’s inefficient to load or store datum that spans quad word boundaries
- Virtual memory is 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, where K is the largest alignment of any element
   - Initial address and structure length must be multiples of K

Example: K = 8, due to double element

Meeting Overall Alignment Requirement

For largest alignment requirement K
   - Overall structure must be multiple of K

Arrays of Structures

Overall structure length multiple of K
   - Satisfy alignment requirement for every element
**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];
```

Put large data types first!

**Saving Space**

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

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

Effect (K = 4)

- `c` 3 bytes
- `i` 4 bytes
- `d` 3 bytes

```
c  3 bytes
i   4 bytes
d  3 bytes
```

**Union Allocation**

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

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

- `c` 1 byte
- `i[0]` 4 bytes
- `i[1]` 4 bytes
- `v` 8 bytes

```
c
i[0] 4 bytes
i[1] 4 bytes
v  8 bytes
```

```
# %rdi holds idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(%rax,4), %eax
```
**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;

- Get direct representation to bit representation of float.
- `bit2float` generates float with given bit pattern.
- Note: this is not the same as `(float) u`.
- `float2bit` generates bit pattern from float.
- Note: this is not the same as `(unsigned) f`.

**Byte Order Revisited**

- Idea
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes.
  - Which is the most (least) significant?
  - Can cause problems when exchanging binary data between machines.

**Big Endian**
- Most significant byte has lowest address.
- PowerPC, Sparc

**Little Endian**
- Least significant byte has lowest address.
- Intel x86, Alpha

**Byte Ordering Example**

```c
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l;
} dw;
```

- `int j;
  for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;
  printf("Chars 0-7 == [0x%02x,0x%02x,0x%02x,0x%02x,0x%02x,0x%02x,0x%02x,0x%02x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);
  printf("Shorts 0-3 == [0x%04x,0x%04x,0x%04x,0x%04x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);
  printf("Ints 0-1 == [0x%08x,0x%08x]\n",
    dw.i[0], dw.i[1]);
  printf("Long == [0x%16x, dw.l];
```

**Output on Pentium:**

- **Chars 0-7** == `[0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]`
- **Shorts 0-3** == `[0xf1f0,0xf3f2,0xf5f4,0xf7f6]`
- **Ints 0-1** == `[0xf3f2f1f0,0xf7f6f5f4]`
- **Long** == `[0x7f6f5f4f3f2f1f0]`
**Silver Endian**

<table>
<thead>
<tr>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
</tr>
</thead>
<tbody>
<tr>
<td>c0</td>
<td>c1</td>
<td>c2</td>
<td>c3</td>
<td>c4</td>
<td>c5</td>
<td>c6</td>
<td>c7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Output on Sun:**

- **Chars 0-7** == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- **Shorts 0-3** == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
- **Ints 0-1** == [0xf0f1f2f3, 0xf4f5f6f7]
- **Long 0** == [0xf0f1f2f3f4f5f6f7]

### Arrays in C
- Contiguous allocation of memory, row order.
- 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
- Allocate bytes in order declared.
- Pad in middle and at end to satisfy alignment.

### Unions
- Overlay declarations.
- Way to circumvent type system.