

Lecture 7

<2016-04-18 Mon>

Contents

1	Structures	2
1.1	Structure Representation	2
1.2	Structures and Alignment	3
1.3	Satisfying Alignment with Structures	3
1.3.1	example	4
1.4	Arrays of Structures	5
1.4.1	Access Array Elements	5
1.4.2	Save Space	6
2	Union	7
2.1	Union Allocation	7
3	Byte Ordering Revisited	8
3.1	Big Endian	8
3.2	Little Endian	8
3.3	Bi Endian	8
3.4	example	9
3.4.1	32 bit, Little Endian	9
3.4.2	32 bit, Big Endian	9
4	Summary of Compound Types in C	9
4.1	Arrays	9
4.2	Structure	10
4.3	Unions	10

1 Structures

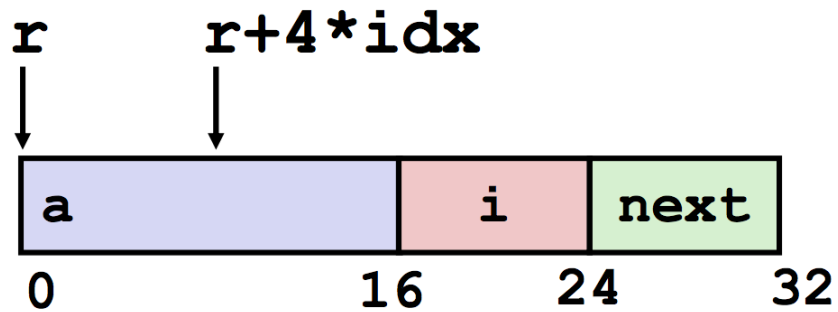
1.1 Structure Representation

- structure represented as block of memory
- fields ordered according to declaration
- compiler determines overall size + position of fields

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

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

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



register	variable
<code>%rdi</code>	first argument
<code>%rsi</code>	second argument

```

get_ap:
    leaq    (%rdi,%rsi,4), %rax
    ret

set_val:
.L11:
    movsq   16(%rdi), %rax
    movl    %esi, (%rdi,%rax,4)
    movq    24(%rdi), %rdi
    testq   %rdi, %rdi
    jne     .L11
    ret

```

1.2 Structures and Alignment

- Aligned Data
 - primitive data type requires K bytes
 - address must be multiple of K
- Motivation
 - memory accessed by (aligned) chunks of 4 or 8 bytes
 - * inefficient to load or store datum that spans quad word boundaries
 - * virtual memory trickier when datum spans 2 pages

bytes	example	requirement
1 byte	char	no restriction
2 bytes	short	lowest 1 bit of address must be 0
4 bytes	int, float	lowest 2 bit of address must be 00
8 bytes	double, long, char*	lowest 3 bit of address must be 000
16 bytes	long double	lowest 4 bit of address must be 0000

1.3 Satisfying Alignment with Structures

- within structure
 - satisfy each element's alignment requirement
- overall structure placement

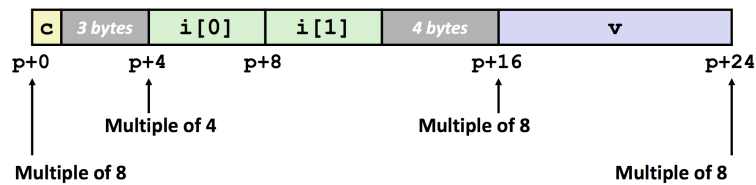
- each structure has alignment requirement K
 - * K = largest alignment of any element in struct
- initial address & structure must be multiples of K

1.3.1 example

1. struct

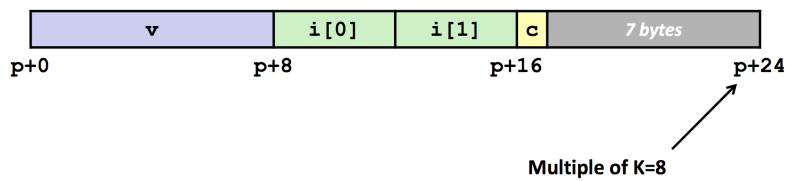
- $K = 8$, due to `double` element

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



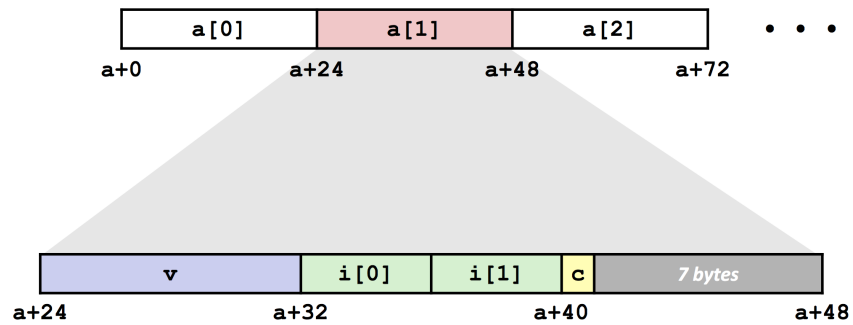
2. reordered within struct

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



1.4 Arrays of Structures

```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```



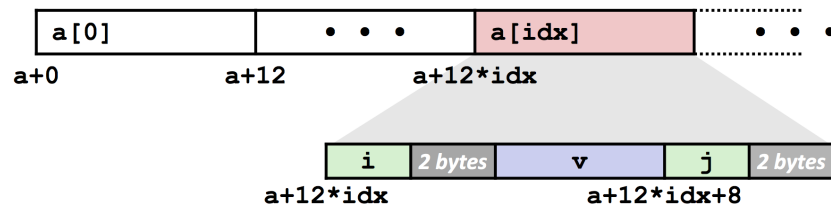
- overall structure length of multiple of K
- satisfy alignment requirement for every element

1.4.1 Access Array Elements

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

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

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



- compute array offset $12 * \text{idx}$
 - `sizeof(S3)`, including alignment spacers
- element j is at offset 8 within structure
- assembler gives offset $a+8$ (resolved during linking)

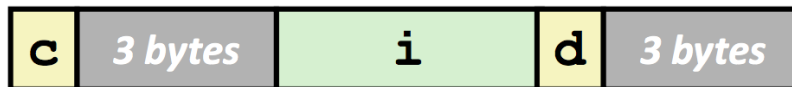
1.4.2 Save Space

- put large data types first

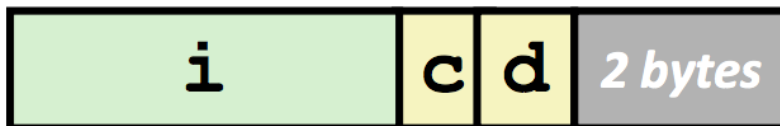
```
struct s4 {
    char c;
    int i;
    char d;
};
```

```
struct s5 {
    int i;
    char c;
    char d;
};
```

- s4 : 12 bytes



- s5 : 8 bytes



2 Union

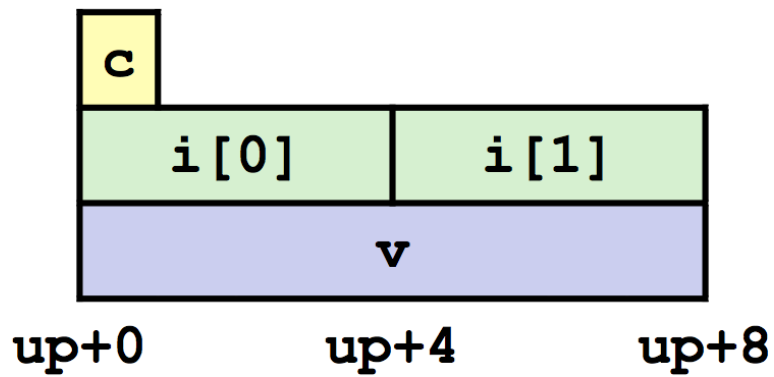
2.1 Union Allocation

- allocate according to largest element
- can only use 1 field at a time

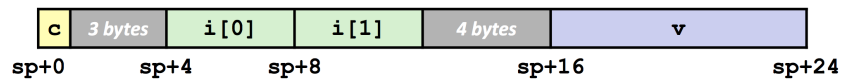
```
union u1 {  
    char c;  
    int i[2];  
    double v;  
} *up;
```

```
struct s1 {  
    char c;  
    int i[2];  
    double v;  
} *sp;
```

- union



- struct



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

bit_float_t arg;
arg.u;          /* interpret arg as unsigned */
arg.f;          /* interpret arg as float */
```

- interpret bytes pattern as **unsigned** / float
- not same as casting

3 Byte Ordering Revisited

- **short** / **long** / **quad** stored in memory as 2/4/8 consecutive bytes
- which byte is most significant
- can cause problem across machine

3.1 Big Endian

- most significant byte has lowest address
- e.g. Sparc

3.2 Little Endian

- least significant byte has lowest address
- e.g. Intel x86, ARM Android, iOS

3.3 Bi Endian

- can be configured either way
- e.g. ARM

3.4 example

```
union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;
```

3.4.1 32 bit, Little Endian

f0	f1	f2	f3	f4	f5	f6	f7
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

LSB
←
MSB
Print
LSB
MSB

3.4.2 32 bit, Big Endian

f0	f1	f2	f3	f4	f5	f6	f7
c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]
s[0]		s[1]		s[2]		s[3]	
i[0]				i[1]			
l[0]							

MSB → LSB MSB → LSB

Print

4 Summary of Compound Types in C

4.1 Arrays

- contiguous allocation of memory

- aligned to satisfy every element's alignment requirement
- pointer to first element
- no bounds checking

4.2 Structure

- allocate bytes in order declared
- pad in middle and at end to satisfy alignment

4.3 Unions

- overlay declarations
- way to circumvent type system