Recitation 6: Structures

15–213/18–243: Introduction to Computer Systems
Recitation, 14 February 2010

Instructors:
15–213/18–243 TAs
Today

- Structures & Alignment
- Unions
Structures

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

- Accessing Structure Member

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

IA32 Assembly

```assembly
# %eax = val
# %edx = r
movl %eax, (%edx)   # Mem[r] = val
```
**Pointer To A Structure Member**

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

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

- Offset of each structure member determined at compile time

---

```asm
# %ecx = idx
# %edx = r
lea 0(%ecx,4),%eax  # 4*idx
lea 4(%eax,%edx),%eax  # r+4*idx+4
```
Structs and alignment

- Struct elements can be referred to by `struct->item` but should not be referenced by `(*struct)+1`

```c
struct rec {
  char c;
  int x;
} *p;
```

- This struct uses 8 bytes, not 5!
Alignment

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
  - Required on some machines; advised on IA32
    - treated differently by IA32 Linux, x86-64 Linux, and Windows!

- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory very tricky when datum spans 2 pages

- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields
Alignment with Structures

- **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 multiples of $K$

- **Example (under Windows or x86-64):**
  - $K = 8$, due to `double` element

```
struct rec {
  char c;
  int i[2];
  double v;
} *p;
```
Different Alignment Conventions

- **x86-64 or IA32 Windows:**
  - K = 8, due to `double` element
  ```c
  struct rec {
    char c;
    int i[2];
    double v;
  } *p;
  ```

- **IA32 Linux**
  - K = 4; `double` treated like a 4-byte data type
  ```c
  struct rec {
    char c;
    int i[2];
    double v;
  } *p;
  ```

Saturday, February 12, 2011
Saving Space

- Put large data types first

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

- Effect (example x86-64, both have K=8)

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

- Initial address should be a multiple of K
- Overall structure length = multiple of K
- Satisfy alignment requirement for every element

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

- Show the memory layout of the following struct on a 64-bit (x86-64) machine
- Reorder for optimal packing

```c
struct foo {
  char a[9];
  short b[3];
  float c;
  char d;
  int e;
  char *f;
  short g;
} a[10];
```

### Before Reorder

```
AAAAAAAAAAxB1B2B3
CCCCDxxxxEEEEExxxx
FFFFFFFFFFGGxxxxx
```

### After Reorder

```
FFFFFFFFCCCCEEEE
B1B2B3GGAAAAAA
ADxxxxxx
```
Today

- Structures & Alignment
- Unions
Union Allocation

- 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;
```
Using Union to Access Bit Patterns

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;
}

Same as (float) u?  
Same as (unsigned) f?
Summary

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

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