The Hardware/Software Interface
CSE351 Winter 2013

Data Structures II: Structs and Unions
Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structs**
  - Alignment

- **Unions**
Structures

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

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

Characteristics

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

Accessing Structure Member

- Given an instance of the struct, we can use the . operator, just like Java:
  - struct rec r1;  r1.i = val;
- What if we have a pointer to a struct: struct rec *r = &r1;
  - Using * and . operators:     (*r).i = val;
  - Or, use -> operator for short: r->i  = val;
- Pointer indicates first byte of structure; access members with offsets

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

IA32 Assembly

```
# %eax = val
# %edx = r
movl %eax,(%edx)  # Mem[r] = val
```
Generating Pointer to Structure Member

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

### Generating Pointer to Array Element
- Offset of each structure member determined at compile time

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

```c
# %ecx = idx
# %edx = r
leal 0(%ecx,4),%eax  # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```
Structures & Alignment

- **Unaligned Data**

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

- **Aligned Data**
  - Primitive data type requires \( K \) bytes
  - Address must be multiple of \( K \)

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

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

- **Aligned data is required on some machines; it is advised on IA32**
  - Treated differently by IA32 Linux, x86-64 Linux, and Windows!

- **What is the motivation for alignment?**
Alignment Principles

■ Aligned Data
  ▪ Primitive data type requires K bytes
  ▪ Address must be multiple of K

■ Aligned data is required on some machines; it is advised on IA32
  ▪ Treated differently by IA32 Linux, x86-64 Linux, and Windows!

■ Motivation for Aligning Data
  ▪ Physical memory is accessed by aligned chunks of 4 or 8 bytes (system-dependent)
    ▪ Inefficient to load or store datum that spans quad word boundaries
  ▪ Also, virtual memory is very tricky when datum spans two pages (later...)

■ Compiler
  ▪ Inserts gaps in structure to ensure correct alignment of fields
  ▪ \texttt{sizeof()} should be used to get true size of structs
Specific Cases of Alignment (IA32)

- **1 byte: char, ...**
  - no restrictions on address

- **2 bytes: short, ...**
  - lowest 1 bit of address must be 0₂

- **4 bytes: int, float, char *, ...**
  - lowest 2 bits of address must be 00₂

- **8 bytes: double, ...**
  - Windows (and most other OSs & instruction sets): lowest 3 bits 000₂
  - Linux: lowest 2 bits of address must be 00₂
    - 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 00₂
Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy 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): $K = ?$**
  - $K = 8$, due to `double` member

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p1;
```
Different Alignment Conventions

- **IA32 Windows or x86-64:**
  - $K = 8$, due to `double` member

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p1;
```

![Diagram of memory alignment for IA32 Windows or x86-64]

- **IA32 Linux:** $K = ?$
  - $K = 4$; `double` aligned like a 4-byte data type

![Diagram of memory alignment for IA32 Linux]
Saving Space

- Put large data types first:

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p1;
```

```c
struct S2 {
    double v;
    int i[2];
    char c;
} *p2;
```

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

Unfortunately, doesn’t satisfy requirement that struct’s *total size* is a multiple of K
Arrays of Structures

- Satisfy alignment requirement for every element

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

- Put large data types first:

  ```c
  struct S3 {
    char c;
    int i;
    char d;
  } *p3;
  ```

  ```c
  struct S4 {
    int i;
    char c;
    char d;
  } *p4;
  ```

- Effect (K=4)

- This strategy *can* save some space for certain structs.
Unions

- Allocated according to largest element
- Can only use one member at a time

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

struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```
What Are Unions Good For?

- Unions allow the same region of memory to be referenced as different types
  - Different “views” of the same memory location
  - Can be used to circumvent C’s type system (bad idea)

- Better idea: use a struct inside a union to access some memory location either as a whole or by its parts
typedef union
{
    unsigned char byte;
    struct {
        unsigned char b0:1;
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char reserved:4;
    } bits;
} hw_register;

hw_register reg;
reg.byte = 0x3F;       // 00111111
reg.bits.b2 = 0;       // 00111001
reg.bits.b3 = 0;       // 00110001
unsigned short a = reg.byte;
printf("0x%X\n", a);    // output: 0x33

(Note: the placement of these fields and other parts of this example are implementation-dependent)
Summary

- **Arrays in C**
  - Contiguous allocations of memory
  - No bounds checking
  - Can usually be treated like a pointer to first element

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

- **Unions**
  - Provide different views of the same memory location