Structures

- Reminder... the C struct declaration creates a data type that groups objects of possibly different types into a single object
- Implementation similar to arrays
  - All components are stored in a contiguous region of memory
  - A pointer to a structure is the address of its first byte
- The compiler maintains information about each structure type indicating the byte offset of each field
  - Generates references to structure elements using these offsets as displacements in memory referencing instructions
Structure allocation

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

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

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

**Memory Layout**
```
0 4 16 20
```

**Assembly**
```
# %eax = val
# %edx = r
movl %eax,(%edx)  # Mem[r] = val
```
Structure Access

Generating Ptr to Structure Member

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

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

find_a:
```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
```

```
pushl %ebp
movl %esp,%ebp
movl 12(%ebp),%eax  // idx (2nd arg)
sall $2,%eax  // mult by 4
addl 8(%ebp),%eax  // ptr to struct (1st arg)
addl $4,%eax
popl %ebp
ret
```
“i” represents the element of “a” that I want “p” to point to.
Data 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 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 in structure to ensure correct alignment of fields
Specific cases of alignment

**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 0₂

- **4 bytes** (e.g., int, float, char *, etc.)
  - lowest 2 bits of address must be 00₂

- **8 bytes** (e.g., double)
  - Windows (and most other OS's & instruction sets):
    - lowest 3 bits of address must be 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)
  - Linux:
    - lowest 2 bits of address must be 00₂
    - i.e., treated the same as a 4-byte primitive data type

**IA32/LINUX address**

- 2 bytes hex: ends in even hex digit (0, 2, 4, 8, A, C, E)

- 4 bytes hex: ends in divisible by 4 hex digit (0,4,8,C)

- 8 bytes hex: ends in divisible by 8 hex digit (0,8)
Satisfying alignment in structures

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 $K$

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

Linux:
- $K = 4$; `double` treated like a 4-byte data type

Long long treated like 8-byte data type
Saving space

- **Put large data types first**

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

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

- **Effect (K=4)**

```
c  3 bytes  
```

```
ici  3 bytes  
```

```
ici  2 bytes  
```
Another Example

```c
struct a_struct {
    char    a;
    struct a_struct *b;
};

struct b_struct {
    char    c;
    int     i;
    double * d;
    short   e[3];
    struct a_struct m;
};
```