Computer Systems I

Class 9

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

Specific Cases of Alignment (IA32)

- **1 byte: `char`, …**
  - no restrictions on address
- **2 bytes: `short`, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, `char *`, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, …**
  - Windows (and most other OS's & instruction sets):
    - lowest 3 bits of address must be 000_2
  - Linux:
    - lowest 2 bits of address must be 00_2
    - i.e., treated the same as a 4-byte primitive data type

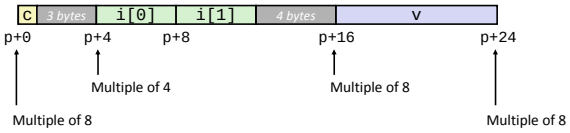
Specific Cases of Alignment (x86-64)

- **1 byte: `char`, …**
  - no restrictions on address
- **2 bytes: `short`, …**
  - lowest 1 bit of address must be 0_2
- **4 bytes: `int`, `float`, …**
  - lowest 2 bits of address must be 00_2
- **8 bytes: `double`, `char *`, …**
  - Windows & Linux:
    - lowest 3 bits of address must be 000_2

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement K
    - K = 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 S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Multiple of 4

Multiple of 8

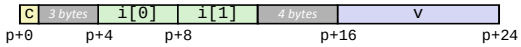
Multiple of 8

Multiple of 8

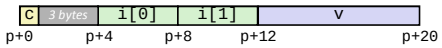
Different Alignment Conventions

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

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



- **IA32 Linux**
  - K = 4; **`double`** treated like a 4-byte data type



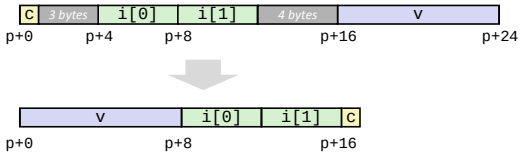
Saving Space

- Put large data types first

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

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

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



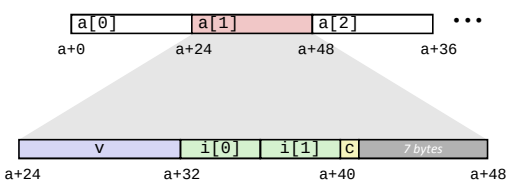
Arrays of Structures

- Padding has been added inside a struct so that fields are aligned.
- This only works if the entire struct is aligned (starts at a correct address.
- What effect does this have on arrays of structs?

Arrays of Structures

- Satisfy alignment requirement for every element

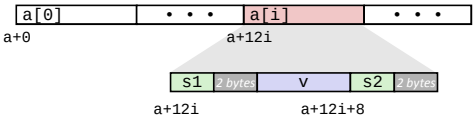
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

- **Compute array offset 12i**
- **Compute offset 8 with structure**
- **Assembler gives offset a+8**
  - Resolved during linking

```
struct S3 {
  short s1;
  float v;
  short s2;
} a[10];
```




```
short get_s2(int idx)
{
  return a[idx].s2;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

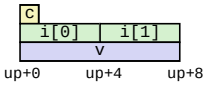

Unions

- **What is a union?**
- **How does it differ from a struct?**

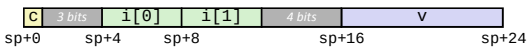
Union Allocation

- **Allocate according to largest element**
- **Can only use ones field at a time**

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




```
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;
```

Don't forget byte ordering!

```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

Same as `(float) u` ?

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

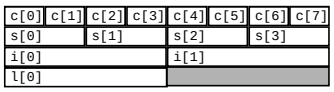
Same as `(unsigned) f` ?

Byte Ordering Revisited

- **Idea**
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which is 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

Byte Ordering Example

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

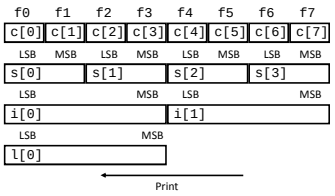


Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\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%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```


Byte Ordering on IA32

Little Endian

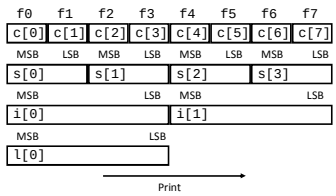


Output on IA32:

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


Byte Ordering on Sun

Big Endian

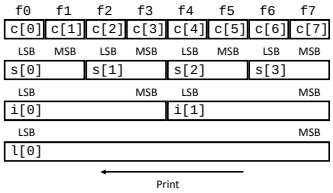


Output on Sun:

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

Byte Ordering on x86-64

Little Endian



Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```


Summary

- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
  - Pointer to first element
  - No bounds checking
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Overlay declarations
  - Way to circumvent type system

Struct review

```
typedef struct {
  double one;
  char two[5];
  int three;
  int *four;
  char five;
} struct_one;

struct_one array[10];

void main() {
  printf("address of array = %d\n", array);
  printf("address of array[3] = %d\n", &array[3]);
  printf("address of array[5].two[2] = %d\n", &array[5].two[2]);
}
```

address of array = 10000

Union review

```
void main() {
  union_one U;
  U.number[0] = 0x01234567;
  U.number[1] = 0x89abcdef;
  printf("number[0] = %08x\n", U.number[0]);
  printf("number[1] = %08x\n", U.number[1]);
  printf("character[0-3] = %02x%02x%02x%02x\n",
         U.character[0], U.character[1], U.character[2], U.character[3]);
  printf("character[4-7] = %02x%02x%02x%02x\n",
         U.character[4], U.character[5], U.character[6], U.character[7]);
}
```

```
number[0] = ???
number[1] = ???
character[0-3] = ???
character[4-7]= ???
```


Summary for last week

- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
  - Pointer to first element
  - No bounds checking
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Overlay declarations
  - Way to circumvent type system