Structured Data II
Heterogenous Data
Sept. 26, 2000

Topics
- Structure Allocation
- Alignment
- Unions
- Byte Ordering
- Byte Operations
- IA32/Linux Memory Organization
- Understanding C declarations

Basic Data Types

Integral
- Stored & operated on in general registers
- Signed vs. unsigned depends on instructions used

<table>
<thead>
<tr>
<th>Type</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>b</td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>l</td>
<td>4</td>
<td>[unsigned] int, char *</td>
</tr>
<tr>
<td>quad word</td>
<td></td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Floating Point
- Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Type</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>s</td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12</td>
<td>long double</td>
</tr>
</tbody>
</table>

Generating Pointer to Structure Member

Accessing Structure Member

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

Memory Layout

```
  i  a  p
0 4 16 20
```

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

Generating Pointer to Array Element

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

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

```assembly
# %eax = val
# %edx = r
leal 0(%ecx,4),%eax  # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```
**Structure Referencing (Cont.)**

**C Code**

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

```c
void set_p(struct rec *r) {
    r->p = &r->a[r->i];
}
```

**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 0
- **4 bytes** (e.g., `int`, `float`, `char *`, etc.)
  - lowest 2 bits of address must be 00
- **8 bytes** (e.g., `double`)
  - Windows (and most other OS’s & instruction sets):
    - lowest 3 bits of address must be 000
  - Linux:
    - lowest 2 bits of address must be 00
    - i.e. treated the same as a 4 byte primitive data type
- **12 bytes** (`long double`)
  - Linux:
    - lowest 2 bits of address must be 00
    - i.e. treated the same as a 4 byte primitive data type

**Satisfying Alignment with 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 $K$

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

<table>
<thead>
<tr>
<th>Element</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>p+0</td>
</tr>
<tr>
<td>i[0]</td>
<td>p+4</td>
</tr>
<tr>
<td>i[1]</td>
<td>p+8</td>
</tr>
<tr>
<td>v</td>
<td>p+16</td>
</tr>
<tr>
<td></td>
<td>p+24</td>
</tr>
</tbody>
</table>

- Multiple of 4
- Multiple of 8
Linux vs. Windows

Windows (including Cygwin):
- K = 8, due to double element

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

```
c i[0] i[1] v
p+0 p+4 p+8 p+16 p+24
```

Multiple of 4
Multiple of 8
Multiple of 8

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

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

```
c i[0] i[1] v
p+0 p+4 p+8 p+12 p+20
```

Multiple of 4
Multiple of 4
Multiple of 4

Effect of Overall Alignment Requirement

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

```
x i[0] i[1] c
p+0 p+8 p+12 p+16 Windows: p+24
```

```
x i[0] i[1] c
p+0 p+4 p+8 p+12 p+16 p+20 Linux: p+20
```

```
x[0] x[1] i[0] i[1] c
```

```
x[0] x[1] i[0] i[1] c
```

```
p must be multiple of 4 (in either OS)
```

```
p must be multiple of:
8 for Windows
4 for Linux
```

```
struct S3 {
    float x[2];
    int i[2];
    char c;
} *p;
```

```
x[0] x[1] i[0] i[1] c
```

```
x[0] x[1] i[0] i[1] c
```

```
p must be multiple of 4
```

Ordering Elements Within Structure

```
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```

```
c1 v c2 i
p+0 p+8 p+16 p+20 p+24
```

10 bytes wasted space in Windows

```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```

```
v c1 c2 i
p+0 p+8 p+12 p+16
```

2 bytes wasted space

Arrays of Structures

Principle
- Allocated by repeating allocation for array type
- In general, may nest arrays & structures to arbitrary depth

```
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```

```
a[1].i a[1].v a[1].j
```

```
a[0] a[1] a[2] ...
```

```
a[0] a[1] a[2] ...
```

```
a[1].i a[1].v a[1].j
```
## Accessing Element within Array

- Compute offset to start of structure
  - Compute $12^\text{th}$ as $4*(i+2)$
- Access element according to its offset within structure
  - Offset by 8
  - Assembler gives displacement as $a + 8$
    - Linker must set actual value

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

## Satisfying Alignment within Structure

### Achieving Alignment

- Starting address of structure array must be multiple of worst-case alignment for any element
  - $a$ must be multiple of 4
- Offset of element within structure must be multiple of element’s alignment requirement
  - $v$’s offset of 4 is a multiple of 4
- Overall size of structure must be multiple of worst-case alignment for any element
  - Structure padded with unused space to be 12 bytes

```c
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```

## Union Allocation

### Principles

- Overlay union elements
- Allocate according to largest element
- Can only use one field at a time

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

## Implementing “Tagged” Union

- Structure can hold 3 kinds of data
- Only one form at any given time
- Identify particular kind with flag `type`

```c
typedef enum { CHAR, INT, DBL } utype;
typedef struct {
    utype type;
    union {
        char c;
        int i[2];
        double v;
    } e;
} store_el, *store_ptr;
store_el k;
```

(Windows alignment)
Using “Tagged” Union

```c
store_ele k1;
  k1.type = CHAR;
  k1.e.c = 'a';

store_ele k2;
  k2.type = INT;
  k2.e.i[0] = 17;
  k2.e.i[1] = 47;

store_ele k3;
  k3.type = DBL;
  k3.e.v = 3.14159265358979323846;
```

Using Union to Access Bit Patterns

```c
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;
}
```

Byte Ordering

**Idea**
- Long/quad words stored in memory as 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
- IBM 360/370, Motorola 68K, Sparc

**Little Endian**
- Least significant byte has lowest address
- Intel x86, Digital VAX

Byte Ordering Example

```c
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%hx,0x%hx,0x%hx,0x%hx,0x%hx,0x%hx,0x%hx,0x%hx]\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%hx,0x%hx,0x%hx,0x%hx]\n",
   dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

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

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

Byte Ordering on x86

Little Endian

```
f0 f1 f2 f3 f4 f5 f6 f7
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i[0]</th>
<th>i[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>MS</td>
</tr>
</tbody>
</table>

Output on Pentium:

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 == [f3f2f1f0]

Byte Ordering on Sun

Big Endian

```
f0 f1 f2 f3 f4 f5 f6 f7
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i[0]</th>
<th>i[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>MS</td>
</tr>
</tbody>
</table>

Output on Sun:

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 == [f3f2f1f0]

Byte Ordering on Alpha

Little Endian

```
f0 f1 f2 f3 f4 f5 f6 f7
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
<td>MSB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i[0]</th>
<th>i[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS</td>
<td>MS</td>
</tr>
</tbody>
</table>

Output on Alpha:

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 == [f3f2f1f0]
**Byte-Level Operations**

**IA32 Support**
- Arithmetic and data movement operations have byte-level version
  - movb, addb, testb, etc.
- Some registers partially byte-addressable
- Can perform single byte memory references

**Compiler**
- Parameters and return values of type char passed as int's
- Use movsbl to sign-extend byte to int

---

**Byte-Level Operation Example**

```c
char string_xor(char *s)
{
    char result = 0;
    char c;
    do {
        c = *s++;
        result ^= c;
    } while (c);
    return result;
}
```

---

**Linux Memory Layout**

- **Stack**
  - Runtime stack (8MB limit)
- **Heap**
  - Dynamically allocated storage
  - When call malloc, calloc, new
- **DLLs**
  - Dynamically Linked Libraries
  - Library routines (e.g., printf, malloc)
  - Linked into object code when first executed
- **Data**
  - Statically allocated data
  - E.g., arrays & strings declared in code
- **Text**
  - Executable machine instructions
  - Read-only

---

**Linux Memory Allocation**

- **Initial Stack**
- **Linked Stack**
- **Some Heap**
- **More Heap**
Memory Allocation Example

```c
char big_array[1<<24]; /* 16 MB */
char huge_array[1<<28]; /* 256 MB */
int beyond;
char *p1, *p2, *p3, *p4;
int useless() { return 0; }

int main()
{
    p1 = malloc(1 << 28); /* 256 MB */
p2 = malloc(1 << 8);  /* 256 B */
p3 = malloc(1 << 28); /* 256 MB */
p4 = malloc(1 << 8);  /* 256 B */
    /* Some print statements ... */
}
```

Dynamic Linking Example

```c
#include <stdio.h>

int useless() { return 0; }

int main()
{
    char *p1, *p2, *p3, *p4;
    int useless() { return 0; }

    p1 = malloc(1 << 28); /* 256 MB */
p2 = malloc(1 << 8);  /* 256 B */
p3 = malloc(1 << 28); /* 256 MB */
p4 = malloc(1 << 8);  /* 256 B */
    /* Some print statements ... */
}
```

```bash
(gdb) print malloc
$1 = {<text variable, no debug info>}
    0x8048454 < malloc>
(gdb) run
Program exited normally.
(gdb) print malloc
$2 = {void *(unsigned int )}
    0x40006240 < malloc>
```

Initially
- Code in text segment that invokes dynamic linker
- Address 0x8048454 should be read 0x08048454

Final
- Code in DLL region

Example Addresses

- $esp 0xbfffff78
- p3 0x500b5008
- p1 0x400b4008
- Final malloc 0x40006240
- p4 0x1904a640
- p2 0x1904a538
- beyond 0x1904a524
- big_array 0x1804a520
- huge_array 0x0804a510
- main() 0x0804856f
- useless() 0x08048560
- Initial malloc 0x08048454

Breakpointing Example

```bash
(gdb) break main
(gdb) run
    Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
$3 = (void *) 0xbfffff78
```

Main
- Address 0x804856f should be read 0x0804856f

Stack
- Address 0xbfffff78
C operators

Operators

()  [ ]  \rightarrow  .  
!  ~  ++  --  +  -  *  &  (type)  sizeof

*  /  %  left to right
+  -  left to right
<<  >>  left to right
<=  >=  ==  !=  left to right
&  |  ^  left to right
&&  ||  left to right
?:  left to right
=  +=  -=  /=  %=  &=  ^=  !=  <<=  >>=  right to left
,

Note: Unary +, -, and * have higher precedence than binary forms

C pointer declarations

int *p
p is a pointer to int

int *p[13]
p is an array[13] of pointer to int

int *(p[13])
p is an array[13] of pointer to int

int **p
p is a pointer to a pointer to an int

int (*p)[13]
p is a pointer to an array[13] of int

int *f()
f is a function returning a pointer to int

int (*f)()
f is a pointer to a function returning int

int (*(*f())[13])()
f is a function returning ptr to an array[13] of pointers to functions returning int

int (*(*x[3])())[5]
x is an array[3] of pointers to functions returning pointers to array[5] of ints