15-213
“The course that gives CMU its Zip!”

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>Intel</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>b</td>
<td></td>
<td>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>Word</td>
<td>w</td>
<td></td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>Double word</td>
<td>l</td>
<td></td>
<td>4</td>
<td>[unsigned] int, char *</td>
</tr>
<tr>
<td>Quad word</td>
<td></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>Intel</th>
<th>GAS</th>
<th>Bytes</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>s</td>
<td></td>
<td>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td></td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td></td>
<td>10/12</td>
<td>long double</td>
</tr>
</tbody>
</table>
Structures

Concept

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

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

Accessing Structure Member

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

Assembly

```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
```
**Structure Referencing (Cont.)**

**C Code**

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

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

```
# %edx = r
movl (%edx),%ecx    # r->i
leal 0(%ecx,4),%eax   # 4*(r->i)
leal 4(%edx,%eax),%eax # r+4+4*(r->i)
movl %eax,16(%edx) # Update r->p
```

**Diagram**

- Element `i` in the `rec` structure
- Pointer `p` pointing to `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 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
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

```c
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

Multiple of 8
Linux vs. Windows

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

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

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

```
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  Multiple of 4
```

```
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  Multiple of 8
```
Effect of Overall Alignment Requirement

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

\[ p \text{ must be multiple of:} \]
8 for Windows
4 for Linux

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

\[ p \text{ must be multiple of 4 (in either OS)} \]
Ordering Elements Within Structure

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

10 bytes wasted space in Windows

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

2 bytes wasted space
Arrays of Structures

Principle

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

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

- Compute offset to start of structure
  - Compute $12 \times i$ as $4 \times (i+2) i$
- 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;
}
```

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

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

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

(Windows alignment)
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_ele, *store_ptr;

store_ele k;
```

```plaintext
k.e.e

k.e.e

k.e.e[0]  k.e.e[1]

k.type  k.e

k.e.e
```
Using “Tagged” Union

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

\[0\] \[a\]

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

\[1\] \[17\] \[47\]

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

\[2\] \[3.1415926535\...\]
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;
}

- Get direct access to bit representation of float
- bit2float generates float with given bit pattern
  - NOT the same as (float) u
- float2bit generates bit pattern from float
  - NOT the same as (unsigned) f
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;
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Byte Ordering Example (Cont).

```c
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 x86

Little Endian

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

```
\begin{array}{cccccccc}
  f_7 & f_6 & f_5 & f_4 & f_3 & f_2 & f_1 & f_0 \\
\end{array}
```

```
\begin{array}{cccc}
  s_3 & s_2 & s_1 & s_0 \\
  \text{MSB} & \text{LSB} & \text{MSB} & \text{LSB} \\
\end{array}
```

```
\begin{array}{cccc}
  i_1 & i_0 \\
  \text{MSB} & \text{LSB} & \text{MSB} & \text{LSB} \\
\end{array}
```

```
\begin{array}{c}
  l_0 \\
  \text{MSB} & \text{LSB} \\
\end{array}
```

### Output on Sun:

- **Characters 0–7**
  
  \[ \{0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7\} \]

- **Shorts 0–3**
  
  \[ \{0xff01, 0xff2f3, 0xff4f5, 0xff6f7\} \]

- **Ints 0–1**
  
  \[ \{0xff01f2f3, 0xff4f5f6f7\} \]

- **Long 0**
  
  \[ \{0xff01f2f3\} \]
Byte Ordering on Alpha

Little Endian

Output on Alpha:

Characters 0–7 == \([\text{x}f0, \text{x}f1, \text{x}f2, \text{x}f3, \text{x}f4, \text{x}f5, \text{x}f6, \text{x}f7]\)

Shorts 0–3 == \([\text{x}f1f0, \text{x}f3f2, \text{x}f5f4, \text{x}f7f6]\)

Ints 0–1 == \([\text{x}f3f2f1f0, \text{x}f7f6f5f4]\)

Long 0 == \([\text{x}f7f6f5f4f3f2f1f0]\)
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

<table>
<thead>
<tr>
<th>%eax</th>
<th>%ah</th>
<th>%al</th>
</tr>
</thead>
<tbody>
<tr>
<td>%edx</td>
<td>%dh</td>
<td>%dl</td>
</tr>
<tr>
<td>%ecx</td>
<td>%ch</td>
<td>%cl</td>
</tr>
<tr>
<td>%ebx</td>
<td>%bh</td>
<td>%bl</td>
</tr>
</tbody>
</table>
Byte-Level Operation Example

- Compute Xor of characters in string

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

```assembly
# %edx = s, %cl = result
movb $0,%cl       # result = 0
L2:               # loop:
    movb (%edx),%al  # *s
    incl %edx        # s++
    xorb %al,%cl     # result ^= c
    testb %al,%al    # al
    jne L2           # If != 0, goto loop
    movsbl %cl,%eax  # Sign extend to int
```
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

Initially

Stack

BF

7F

3F

80

40

00

Text

Data

Linked

Stack

BF

7F

3F

80

40

00

DLLs

Text

Data

Some

Heap

Stack

BF

7F

3F

80

40

00

Heap

DLLs

Text

Data

More

Heap

Stack

BF

7F

3F

80

40

00

Heap

DLLs

Heap

Text

Data

class09.ppt
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

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

Final
- Code in DLL region
Breakpointing Example

(gdb) break main
(gdb) run
   Breakpoint 1, 0x804856f in main ()
(gdb) print $esp
   $3 = (void *) 0xbfffffc78

Main
   • Address 0x804856f should be read 0x0804856f

Stack
   • Address 0xbfffffc78
Example Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$esp</td>
<td>0xbfffffc78</td>
</tr>
<tr>
<td>p3</td>
<td>0x500b5008</td>
</tr>
<tr>
<td>p1</td>
<td>0x400b4008</td>
</tr>
<tr>
<td>Final malloc</td>
<td>0x40006240</td>
</tr>
<tr>
<td>p4</td>
<td>0x1904a640</td>
</tr>
<tr>
<td>p2</td>
<td>0x1904a538</td>
</tr>
<tr>
<td>beyond</td>
<td>0x1904a524</td>
</tr>
<tr>
<td>big_array</td>
<td>0x1804a520</td>
</tr>
<tr>
<td>huge_array</td>
<td>0x0804a510</td>
</tr>
<tr>
<td>main()</td>
<td>0x0804856f</td>
</tr>
<tr>
<td>useless()</td>
<td>0x08048560</td>
</tr>
<tr>
<td>Initial malloc</td>
<td>0x08048454</td>
</tr>
</tbody>
</table>
# C operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()  []  -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>!  ~  ++  --  +  -  *  &amp;  (type)  sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>*  /  %</td>
<td>left to right</td>
</tr>
<tr>
<td>+  -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt;  &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;  &lt;=  &gt;  &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>==  !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
</tr>
<tr>
<td>=  +=  -=  *=  /=  %=  &amp;=  ^=  !=  &lt;&lt;=  &gt;&gt;=</td>
<td>right to left</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>

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