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>1</td>
<td></td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td>2</td>
<td></td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>1</td>
<td>4</td>
<td></td>
<td>[unsigned] int</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>4</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td>8</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12</td>
<td>long double</td>
<td></td>
</tr>
</tbody>
</table>

Array Allocation

Basic Principle

$T \ A[L]$;
- Array of data type $T$ and length $L$
- Contiguously allocated region of $L \cdot $sizeof($T$) bytes

```c
char string[12];
int val[5];
double a[4];
char *p[3];
```

Array Access

Basic Principle

$T \ A[0];$
- Array of data type $T$ and length $L$
- Identifier $A$ can be used as a pointer to array element 0

```c
int val[5];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>X</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>X+4</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>X+8</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>X+4+i</td>
</tr>
</tbody>
</table>

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

Machine-Level Programming IV: Structured Data
Feb 5, 2004

- Topics
  - Arrays
  - Structs
  - Unions
Array Example

typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

Notes
- Declaration "zip_dig cmu" equivalent to "int cmu[5]"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Referencing Examples

zip_dig cmu;

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>60</td>
<td>64</td>
<td>68</td>
<td>72</td>
<td>76</td>
</tr>
</tbody>
</table>

zip_dig mit;

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td>52</td>
<td>56</td>
</tr>
</tbody>
</table>

zip_dig ucb;

<table>
<thead>
<tr>
<th></th>
<th>9</th>
<th>4</th>
<th>7</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
<td>60</td>
<td>64</td>
<td>68</td>
<td>72</td>
<td>76</td>
</tr>
</tbody>
</table>

Code Does Not Do Any Bounds Checking!

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit[3]</td>
<td>36 + 4*3 = 48</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit[5]</td>
<td>36 + 4*5 = 56</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>mit[-1]</td>
<td>36 + 4*-1 = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15 = 76</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- Out of range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

Array Accessing Example

int get_digit(zip_dig z, int dig) {
    return z[dig];
}

Computation
- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at 4*%eax + %edx
- Use memory reference (%edx, %eax, 4)

Memory Reference Code

```
# %edx = z
# %eax = dig
movl (%edx, %eax, 4), %eax  # z[dig]
```

Array Loop Example

int zd2int(zip_dig z) {
    int i; int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}

int zd2int(zip_dig z) {
    int i; int zi = 0;
    for (i = 0; i < 5; i++) {
        do {
            zi = 10 * zi + z[i];
            i++;
        } while (i < 5);
    }
    return zi;
}

Original Source
- How do we implement this?
- Can we improve it?

First step, convert to do-while

Next?
Array Loop Example - convert to ptr

```c
int zd2int(zip_dig z) {
    int i; int zi = 0; i = 0; if (i < 5) {
        do {
            zi = 10 * zi + *(z+i); i++;
        } while (i < 5);
    return zi;
}
```

Can we further improve this?
(hint: what does i do?)

<table>
<thead>
<tr>
<th>z[i]</th>
<th>*(z+i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>z+1</td>
</tr>
<tr>
<td>z+2</td>
<td>z+3</td>
</tr>
<tr>
<td>z+4</td>
<td>z+5</td>
</tr>
</tbody>
</table>

Do we need z+i?

```
 Array Loop Example - optimize

```c
int zd2int(zip_dig z) {
    int* zend; int zi = 0; zend = z + 4; do {
        zi = 10 * zi + *z; z++;
    } while (z <= zend); return zi;
}
```

Can I do anything else?

```
 Array Loop Example - optimize

```c
int zd2int(zip_dig z) {
    int i; int zi = 0; if (i < 5) {
        do {
            zi = 10 * zi + *(z++); i++;
        } while (i < 5);
    return zi;
}
```

Array Loop Example

Original Source

- As generated by GCC
- Express in do-while form
- No need to test at entrance
- Convert array code to pointer code
- Eliminate loop variable i

Transformed Version
Array Loop Implementation

Registers
- %ecx z
- %eax zi
- %ebx zend

Computations
- \(10 \times zi + *z\) implemented as \(*z + 2 \times (zi + 4 \times zi)\)
- zi increments by 4

int zd2int(zip_dig z) {
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}

Nested Array Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] = {
  {1, 5, 2, 0, 6},
  {1, 5, 2, 1, 3},
  {1, 5, 2, 1, 7},
  {1, 5, 2, 2, 1}     
};
```

- Declaration "zip_dig pgh[4]" ⇔ "int pgh[4][5]"
  - Variable pgh denotes array of 4 elements
  - Each element is an array of 5 int's
    - Allocated contiguously
- "Row-Major" ordering of all elements guaranteed

Nested Array Allocation

Declaration

\( T A[R][C]; \)
- Array of data type \( T \)
- \( R \) rows, \( C \) columns
- Type \( T \) element needs \( K \) bytes

Array Size
- \( R \times C \times K \) bytes

Arrangement
- Row-Major Ordering

```
int A[R][C];
```

```
A[0][0] .... A[0][C-1]
A[1][0] .... A[1][C-1]
...                ...
A[R-1][0] .... A[R-1][C-1]
```

```
A[i] is array of \( C \) elements
Each element of type \( T \)
Starting address \( A + i \times C \times K \)
```

```
int A[R][C];
```

```
```

```
A + i \times C \times K
```

```
A + (R-1) \times C \times K
```

Nested Array Row Access

Row Vectors
- \( A[i] \) is array of \( C \) elements
- Each element of type \( T \)
- Starting address \( A + i \times C \times K \)

```
int A[R][C];
```

```
```

```
A + i \times C \times K
```

```
A + (R-1) \times C \times K
```
Nested Array Row Access Code

```c
int *get_pgh_zip(int index) {
    return pgh[index];
}
```

**Row Vector**
- `pgh[index]` is array of 5 int's
- Starting address `pgh+20*index`

**Code**
- Computes and returns address
- Compute as `pgh + 4*(index+4*index)`

```c
# %eax = index
leal (%eax,%eax,4),%eax # 5 * index
leal pgh(%eax,%eax,4),%eax # pgh + (20 * index)
```

---

Nested Array Element Access Code

**Array Elements**
- `pgh[index][dig]` is `int`
- Address: `pgh + 20*index + 4*dig`

**Code**
- Computes address
  - `pgh + 4*dig + 4*(index+4*index)`
- `movl` performs memory reference

```c
# %ecx = dig
# %eax = index
leal (%eax,%eax,4),%edx # 4*dig
leal (%eax,%eax,4),%eax # 5*index
movl pgh(%edx,%eax,4),%eax # *(pgh + 4*dig + 20*index)
```

---

Nested Array Element Access

**Array Elements**
- `A[i][j]` is element of type `T`
- Address `A + (i*C+j)*K`

```c
int A[R][C];
```

**References**

```c
zip_digit pgh[4];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pgh[3][3]</code></td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][5]</code></td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][-1]</code></td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[4][-1]</code></td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][19]</code></td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][-1]</code></td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>?</td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array is guaranteed
**Multi-Level Array Example**

- Variable `univ` denotes array of 3 elements
- Each elem is a pointer
  - 4 bytes
- Each pointer points to an array of int's

```c
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

**Element Access in Multi-Level Array**

```c
int get_univ_digit(int index, int dig) {
    return univ[index][dig];
}
```

**Computation**

- Element access
  - `Mem[Mem[univ+4*index]+4*dig]`
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array

```c
leal 0(%ecx,4),%edx  # 4*index  
movl univ(%edx),%eax  # Mem[univ+4*index]  
movl (%eax,%eax,4),%eax  # Mem[...+4*dig]
```

**Array Element Accesses**

Syntax is the same, computation is different!

**Nested Array**

```c
int get_pgh_digit(int index, int dig) {
    return pgh[index][dig];
}
```

**Multi-Level Array**

```c
int get_univ_digit(int index, int dig) {
    return univ[index][dig];
}
```

**Strange Referencing Examples**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>univ[2][3]</code></td>
<td>56+4*3 = 68</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td><code>univ[1][5]</code></td>
<td>16+4*5 = 36</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[2][-1]</code></td>
<td>56+4*-1 = 52</td>
<td>-1</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[3][-1]</code></td>
<td>??</td>
<td>??</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[1][12]</code></td>
<td>16+4*12 = 64</td>
<td>7</td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed
Using Nested Arrays

**Strengths**
- C compiler handles doubly subscripted arrays
- Generates very efficient code
- Avoids multiply in inner computation

**Limitation**
- Only works if have fixed array size

```
#define N 16
typedef int fix_matrix[N][N];

/* Compute element i,k of
fixed matrix product */
int fix_prod_ele (fix_matrix a, fix_matrix b, int i, int k) {
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```

Dynamic Nested Arrays

**Strength**
- Can create matrix of arbitrary size

**Programming**
- Must do index computation explicitly

**Performance**
- Accessing single element costly
- Must do multiplication

```
int * new_var_matrix(int n) {
    return (int *) malloc(sizeof(int), n*n);
}

int var_ele (int *a, int i, int j, int n) {
    return a[i*n+j];
}
```

Dynamic Array Multiplication

**Without Optimizations**
- Multiplies
  - 2 for subscripts
  - 1 for data
- Adds
  - 4 for array indexing
  - 1 for loop index
  - 1 for data

```
/* Compute element i,k of
variable matrix product */
int var_prod_ele (int *a, int *b, int i, int k, int n) {
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result += a[i*n+j] * b[j*n+k];
    return result;
}
```

Can we optimize this?

Optimizing Dynamic Array Multi

```
/* Compute element i,k of
variable matrix product */
int var_prod_ele(int *a, int *b, int i, int k, int n) {
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result += a[i*n+j] * b[j*n+k];
    return result;
}
```

<table>
<thead>
<tr>
<th>iter</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>i*n</td>
<td>i*n+4</td>
<td>i*n+8</td>
<td>i*n+12</td>
</tr>
<tr>
<td>b</td>
<td>k</td>
<td>n+k</td>
<td>2*n+k</td>
<td>3*n+k</td>
</tr>
</tbody>
</table>
/* Compute element i,k of variable matrix product */
int var_prod_ele(int *a, int *b, int i, int k, int n) {
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result += a[i*n+j] * b[j*n+k];
    return result;
}

iter 0 1 2 3
a index i*n i*n+4 i*n+8 i*n+12
b index k n+k 2*n+k 3*n+k

/* Compute element i,k of variable matrix product */
int var_prod_ele(int *a, int *b, int i, int k, int n) {
    int j;
    int iTn = i * n;
    int result = 0;
    for (j = 0; j < n; j++)
        result += a[iTn+j] * b[j*n+k];
    return result;
}

iter 0 1 2 3
a index i*n i*n+4 i*n+8 i*n+12
b index k n+k 2*n+k 3*n+k
### Optimizing Dynamic Array Mult.

**Optimizations**
- Performed when set optimization level to -O2

**Code Motion**
- Expression i*n can be computed outside loop

**Strength Reduction**
- Incrementing j has effect of incrementing j*n+k by n

**Performance**
- Compiler can optimize regular access patterns

```c
int result = 0;
for (j = 0; j < n; j++)
    result += a[i*n+j] * b[j*n+k];
return result;
```

### Structures

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

**Memory Layout**

<table>
<thead>
<tr>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

**Accessing Structure Member**

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

**Assembly**

```
$ %eax = val
$ %edx = r
movl %eax, (%edx) # Mem[r] = val
```

### Generating Ptr to Structure Member

**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];
}
```

### Structure Referencing (Cont.)

**C Code**

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

**Assembly**

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

---

**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

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

---

**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,
  - 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
- **4 bytes (e.g., int, float, char *, etc.):**
  - Lowest 2 bits of address must be 00,
  - I.e., treated the same as a 4-byte primitive data type
- **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

---

**Linux vs. Windows**

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

**Linux:**
- K = 4; double treated like a 4-byte data type
Overall Alignment Requirement

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

- `p` must be multiple of 8 for Windows
- `p` must be multiple of 4 (in either OS)

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

Windows: `p+24`
Linux: `p+20`

Arrays of Structures

- 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];
```

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

Accessing Element within Array

- Compute offset to start of structure
  - Compute `$12*\text{sizeof}(\text{S6})$`
  - Access element according to its offset within structure
  - Offset by 8
  - Assembler gives displacement as `a + 8`
    - Linker must set actual value

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

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

```
skip 8

; ;

```

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


```c

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

Windows: `p+24`
Linux: `p+20`

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

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

```
skip 8

; ;

```

10 bytes wasted space in Windows

2 bytes wasted space

11
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

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

```
a[0]   …   a[1]   …   
        a+121       

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

```
| a+0    | a+121  |
```

Multiple of 4

```
```

Union Allocation

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

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

```
sp+0  sp+4  sp+8  sp+16  sp+24

(Windows alignment)
```

```
sp+0  sp+4  sp+8  sp+16  sp+24

(Windows alignment)
```

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;
}
```

```
float u  f
```

```
0  4

- 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 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, Alpha
Byte Ordering Example

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

c[3] s[1] i[0]
c[2] c[1] s[0]
c[0]

c[4]  

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 Example (Cont.

Byte Ordering on x86

Little Endian

Big Endian

Output on Pentium:
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf0f0,0xf2f2,0xf4f4,0xf6f6]
Ints 0-1 == [0xf0f0f0f0,0xf0f0f0f0f0f0f0f0]
Long 0 == [f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f09]

Output on Sun:
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf0f0,0xf2f2,0xf4f4,0xf6f6]
Ints 0-1 == [0xf0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f09]
Long 0 == [f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f09]
Byte Ordering on Alpha

Little Endian


<table>
<thead>
<tr>
<th>LSB</th>
<th>MSB</th>
<th>LSB</th>
<th>MSB</th>
<th>LSB</th>
<th>MSB</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>MSB</th>
<th>LSB</th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l[0]$</td>
<td>$l[1]$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output on Alpha:

- Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
- Shorts 0-3 == [0xf0f0, 0xf3f2, 0xf5f4, 0xf7f6f5f4]
- Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4f3f2f1f0]
- Long 0 == [0xf7f6f5f4f3f2f1f0]

Summary

Arrays in C
- Contiguous allocation of memory
- Pointer to first element
- No bounds checking

Compiler Optimizations
- Compiler often turns array code into pointer code (zd2int)
- Uses addressing modes to scale array indices
- Lots of tricks to improve array indexing in loops

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

Unions
- Overlay declarations
- Way to circumvent type system