Machine-Level Programming IV: Structured Data
Feb 4, 2003

• Topics
  • Arrays
  • Structs
  • Unions

Basic Data Types

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

<table>
<thead>
<tr>
<th></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>1</td>
<td></td>
<td>4</td>
<td>[unsigned] int</td>
</tr>
</tbody>
</table>

Floating Point
- Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th></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>

Array Allocation
Basic Principle

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

char string[12];

int val[5];

double a[4];

char *p[3];

Array Access
Basic Principle

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

int val[5];

Reference Type Value

val[4] int 3
val int * x
val+1 int * x + 4
&val[2] int * x + 8
val[5] int ??
*(val+1) int 5
val + i int * x + 4 i
### Array Example

```c
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

### Array Accessing Example

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

### Referencing Examples

<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 Loop Example

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

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

**Code Does Not Do Any Bounds Checking!**

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

**First step, convert to do-while**

Next?
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?)

i 0 1 2 3 4 5
(z+i) z z+1 z+2 z+3 z+4 z+5
Do we need z+i?

Can I do anything else?

i 0 1 2 3 4 5
(z+i) z z+1 z+2 z+3 z+4 z+5
Do we need i?

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

Transformed Version

- 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
### Nested Array Example

```c
#define PCOUNT 4

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

  - Variable `pgh` denotes array of 4 elements
    - Allocated contiguously
  - 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 * C * K` bytes

**Arrangement**

- Row-Major Ordering

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

```c
A
[0]
[0]  \ldots  A[0][C-1]
  \vdots  \vdots
A[R-1][0]  \ldots  A[R-1][C-1]
```

- `4*R*C` bytes

### Nested Array Row Access

**Row Vectors**

- `A[i]` is array of `C` elements
- Each element of type `T`
- Starting address `A + i*C*K`

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

```c
A[0]  \ldots  A[0][C-1]  \ldots  A[i]  \ldots  A[i][C-1]  \ldots  A[A[R-1]]  \ldots  A[A[R-1][C-1]]
A[0]  \ldots  A[i]  \ldots  A[A[R-1] [0]  \ldots  A[A[R-1] [C-1]]
```

- `A + i*C*4`
- `A + (R-1)*C*4`
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,4),%eax       # pgh + (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];
```

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

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

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

```c
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx          # 4*dig
leal (,%eax,%eax,4),%eax      # 5*index
movl pgh(,%edx,%eax,4),%eax   # *(pgh + 4*dig + 20*index)
```

Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pgh[3][3]</code></td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][5]</code></td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][-1]</code></td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[4][-1]</code></td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][19]</code></td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][-1]</code></td>
<td>76+20<em>0+4</em>-1 = 72</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 an array of 3 elements.
- Each element 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];
}
```

Element Access in Multi-Level Array

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

Array Element Accesses

Syntax is the same, computation is different!

<table>
<thead>
<tr>
<th>Nested Array</th>
<th>Multi-Level Array</th>
</tr>
</thead>
</table>
| int get_pgh_digit(int index, int dig) {
  return pgh[index][dig];
} | int get_univ_digit(int index, int dig) {
  return univ[index][dig];
} |

- Element at `mem[pgh+20*index+4*dig]`
- Element at `mem[mem[univ+4*index]+4*dig]`

Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address Value Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>univ[2][3]</code></td>
<td><code>56+4*3 = 68</code></td>
</tr>
<tr>
<td><code>univ[1][5]</code></td>
<td><code>16+4*5 = 36</code></td>
</tr>
<tr>
<td><code>univ[2][-1]</code></td>
<td><code>56+4*-1 = 52</code></td>
</tr>
<tr>
<td><code>univ[3][-1]</code></td>
<td>??</td>
</tr>
<tr>
<td><code>univ[1][12]</code></td>
<td><code>16+4*12 = 64</code></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 index computation

Limitation
- Only works if have fixed array size

```c
#define N 16
typdef 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

```c
int * new_var_matrix(int n) {
    return (int *) calloc(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

```c
/* Compute element i,k of variable matrix product */
int var_prod_ele(int *a, 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;
}
```

Optimizing Dynamic Array Mul

```c
/* 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>i*n</th>
<th>i*n+4</th>
<th>i*n+8</th>
<th>i*n+12</th>
</tr>
</thead>
<tbody>
<tr>
<td>a index</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b index</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Optimizing Dynamic Array Mult

```c
/* 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;
}
```

```
\begin{tabular}{|c|c|c|c|c|}
\hline
iter & 0 & 1 & 2 & 3 \\
\hline
a index & i*n & i*n+4 & i*n+8 & i*n+12 \\
\hline
b index & k & n+k & 2*n+k & 3*n+k \\
\hline
\end{tabular}
```

Invariant Code Motion

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

```
\begin{tabular}{|c|c|c|c|c|}
\hline
iter & 0 & 1 & 2 & 3 \\
\hline
a index & i*n & i*n+4 & i*n+8 & i*n+12 \\
\hline
b index & k & n+k & 2*n+k & 3*n+k \\
\hline
\end{tabular}
```

Invariant Code Motion

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

```
\begin{tabular}{|c|c|c|c|c|}
\hline
iter & 0 & 1 & 2 & 3 \\
\hline
a index & i*n & i*n+4 & i*n+8 & i*n+12 \\
\hline
b index & k & n+k & 2*n+k & 3*n+k \\
\hline
\end{tabular}
```

Invariant Code Motion

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

```
\begin{tabular}{|c|c|c|c|c|}
\hline
iter & 0 & 1 & 2 & 3 \\
\hline
a index & i*n & i*n+4 & i*n+8 & i*n+12 \\
\hline
b index & k & n+k & 2*n+k & 3*n+k \\
\hline
\end{tabular}
```

Invariant Code Motion

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

```
\begin{tabular}{|c|c|c|c|c|}
\hline
iter & 0 & 1 & 2 & 3 \\
\hline
a index & i*n & i*n+4 & i*n+8 & i*n+12 \\
\hline
b index & k & n+k & 2*n+k & 3*n+k \\
\hline
\end{tabular}
```

Invariant Code Motion

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

```
\begin{tabular}{|c|c|c|c|c|}
\hline
iter & 0 & 1 & 2 & 3 \\
\hline
a index & i*n & i*n+4 & i*n+8 & i*n+12 \\
\hline
b index & k & n+k & 2*n+k & 3*n+k \\
\hline
\end{tabular}
```
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

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

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

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;
};
```

Memory Layout

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

Accessing Structure Member

```
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

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

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

Generating Pointer to Array Element

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

```
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
```

Structure Referencing (Cont.)

```
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
```
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)
  - 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

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

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

```c
struct S2 {
    double x;
    int i[2];
    char c;
} *p;
```
p must be multiple of:

- 8 for Windows
- 4 for Linux

```
struct S3 {
    float x[2];
    int i[2];
    char c;
} *p;
```
p must be multiple of 4 (in either OS)

Windows:
p+24

Linux:
p+20

```
v  c1  c2  i
p+0  p+8  p+12  p+16
```
10 bytes wasted space in Windows

```
v  c1  c2  i
p+0  p+8  p+12  p+16
```
2 bytes wasted space

Ordering Elements Within Structure

```
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```
```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```

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];
```

Accessing Element within Array

- Compute offset to start of structure
  - Compute 12*i 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

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

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

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

```
# %eax = idx
lea (%eax,%eax,2),%eax  # 3*idx
movswl a+8(,%eax,4),%eax
```
Satisfying Alignment within Structure

Achieving Alignment
- Starting address of structure array must be multiple of worst-case alignment for any element
  - for short must be multiple of 4
- Offset of element within structure must be multiple of element’s alignment requirement
  - for float v, its 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];

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;

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;

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 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

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

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,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**

```
f0 f1 f2 f3 f4 f5 f6 f7
LSB MSB LSB MSB LSB MSB LSB MSB
LSB MSB LSB MSB
i[0] i[1]
LSB MSB
l[0]
```

Output on Pentium:
- Characters 0-7 == \[0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]\n- Shorts 0-3 == \[0xf1f0,0xf3f2,0xf5f4,0xf7f6]\n- Ints 0-1 == \[0xf3f2f1f0,0xf7f6f5f4]\n- Long 0 == \[f3f2f1f0]\n
---

Byte Ordering on Sun

**Big Endian**

```
f0 f1 f2 f3 f4 f5 f6 f7
MSB LSB MSB LSB MSB LSB MSB LSB
MSB LSB MSB LSB
i[0] i[1]
MSB LSB
l[0]
```

Output on Sun:
- Characters 0-7 == \[0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]\n- Shorts 0-3 == \[0xf0f1,0xf2f3,0xf4f5,0xf6f7]\n- Ints 0-1 == \[0xf0f1f2f3,0xf4f5f6f7]\n- Long 0 == \[0xf0f1f2f3\]
Byte Ordering on Alpha

Little Endian

<table>
<thead>
<tr>
<th>f0</th>
<th>f1</th>
<th>f2</th>
<th>f3</th>
<th>f4</th>
<th>f5</th>
<th>f6</th>
<th>f7</th>
</tr>
</thead>
</table>

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

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