Machine Representation of Programs: Arrays, Structs and Unions

Instructor: Sanjeev Setia

This lecture
- Arrays
- Structures
- Alignment
- Unions
This lecture

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
- Alignment
- Unions

Basic Data Types

- Integral
  - Stored & operated on in general (integer) registers
  - Signed vs. unsigned depends on instructions used
    
    | Type       | Intel | GAS | Bytes | C               |
    |------------|-------|-----|-------|-----------------|
    | byte       | b     | 1   |       | [unsigned] char |
    | word       | w     | 2   |       | [unsigned] short|
    | double word| l     | 4   |       | [unsigned] int  |
    | quad word  | q     | 8   |       | [unsigned] long int (x86-64) |

- Floating Point
  - Stored & operated on in floating point registers
    
    | Type | Intel | GAS | Bytes | C       |
    |------|-------|-----|-------|---------|
    | Single | s | 4 | float |
    | Double | l | 8 | double |
    | Extended | t | 10/12/16 | long double |
Array Allocation

- **Basic Principle**
  
  \( T \ A[L] \);
  
  - Array of data type \( T \) and length \( L \)
  - Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes

```plaintext
char string[12];

int val[5];

double a[3];

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: Type \( T^* \)

```plaintext
int val[5];
```

- **Reference**

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td></td>
</tr>
<tr>
<td>val</td>
<td></td>
</tr>
<tr>
<td>val+1</td>
<td></td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td></td>
</tr>
<tr>
<td>val[5]</td>
<td></td>
</tr>
<tr>
<td>*(val+1)</td>
<td></td>
</tr>
<tr>
<td>val + i</td>
<td></td>
</tr>
</tbody>
</table>
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: Type \( T^* \)

<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>

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 };
```

- 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];
}
```

### IA32

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

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at $4 \times %eax + %edx$
- Use memory reference $(%edx, %eax, 4)$

Referencing Examples

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>9</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>7</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

- Reference `mit[3]`
- Reference `mit[5]`
- Reference `mit[-1]`
- Reference `cmu[15]`
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>

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

Array Loop Example

- Original

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

- Transformed

```c
int zd2int(zip_dig z) {
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z <= zend);
    return zi;
}
```
Array Loop Implementation (IA32)

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

# %ecx = z
xorl %eax,%eax
leal 16(%ecx),%ebx
.L59:  
    leal (%eax,%eax,4),%edx
    movl (%ecx),%eax
    addl $4,%ecx
    leal (%eax,%edx,2),%eax
    cmpl %ebx,%ecx
    jle .L59
```

Array Loop Implementation (IA32)

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

- Computations
  - \(10 \times zi + *z\) implemented as \(*z + 2 \times (zi+4)\)
  - \(z++\) 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;
}

# %ecx = z
xorl %eax,%eax
leal 16(%ecx),%ebx
.L59:
    leal (%eax,%eax,4),%edx
    movl (%ecx),%eax
    addl $4,%ecx
    leal (%eax,%edx,2),%eax
    cmpl %ebx,%ecx
    jle .L59
```
Nested Array Example

```c
#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)];
```

- “zip_dig pgh[4]” equivalent to “int pgh[4][5]”
  - Variable pgh: array of 4 elements, allocated contiguously
  - Each element is an array of 5 int's, allocated contiguously
- “Row-Major” ordering of all elements guaranteed

Multidimensional (Nested) Arrays

- **Declaration**
  - \( T \ A[R][C]; \)
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Type \( T \) element requires \( K \) bytes
- **Array Size**
  - \( R \times C \times K \) bytes
- **Arrangement**
  - Row-Major Ordering

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

- \( A[0][0] \ldots A[0][C-1] \)
  - \( \ldots \)
  - \( \ldots \)
  - \( \ldots \)
  - \( \ldots \)
  - \( \ldots \)
  - \( \ldots \)
  - \( \ldots \)
  - \( A[R-1][0] \ldots A[R-1][C-1] \)
- \( 4 \times R \times C \) Bytes
Nested Array Row Access

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

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

---

### Nested Array Row Access Code

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

```
#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 } };
```

- What data type is $pgh[index]$?
- What is its starting address?

```
# %eax = index
leal (%eax,%eax,4),%eax
leal pgh(,%eax,4),%eax
```
Nested Array Row Access Code

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

```c
#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, 1, 7 };
```

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

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

- **IA32 Code**
  - Computes and returns address
  - Compute as `pgh + 4*(index+4*index)`

Nested Array Row Access

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

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

```

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

A ... [i] ...  ...  A

A+i*C*4  ...  A+R-1)C*4

A+i*C*4+j*4
```
Nested Array Element Access Code

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

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

- **IA32 Code**
  - Computes address `pgh + 4*dig + 4*(index+4*index)`
  - `movl` performs memory reference

Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgh[3][3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pgh[0][-1]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgh[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][-1]</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 guaranteed

Mul/-Level Array Example

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

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

- Variable univ denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of int's
Element Access in Multi-Level Array

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

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

**Computation (IA32)**
- 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
Array Element Accesses

**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];
}
```

Access looks similar, but element:

Mem[pgh+20*index+4*dig]  
Mem[Mem[univ+4*index]+4*dig]

---

Strange Referencing Examples

Reference | Address | Value | Guaranteed?
---|---|---|---
univ[2][3] | 36 | | 
univ[1][5] | | | 
univ[2][-1] | 4*-1 = 52 | 9 | 
univ[3][-1] | | ?? | 
univ[1][12] | 4*12 = 64 | 7 |
Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>univ[2][3]</td>
<td>56+4*3 = 68</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>univ[1][5]</td>
<td>16+4*5 = 36</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>univ[2][-1]</td>
<td>56+4*-1 = 52</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>univ[3][-1]</td>
<td>??</td>
<td>??</td>
<td>No</td>
</tr>
<tr>
<td>univ[1][12]</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 index computation

- **Limitation**
  - Only works for 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 any 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);
}
```

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

```asm
movl 12(%ebp),%eax   # I
movl 8(%ebp),%edx    # a
imull 20(%ebp),%eax  # n*I
addl 16(%ebp),%eax   # n*i+j
movl (%edx,%eax,4),%eax # Mem[a+4*(i*n+j)]
```

---

Dynamic Array Multiplication

- **Without Optimizations**
  - Multiplies: 3
    - 2 for subscripts
    - 1 for data
  - Adds: 4
    - 2 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 *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;
}
```
Optimizing Dynamic Array Multiplication

- **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
- **Operations count**
  - 4 adds, 1 mult
- **Compiler can optimize regular access patterns**

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

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

This lecture

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
- **Alignment**
- **Unions**
Structures

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

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

**Accessing Structure Member**

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

**Memory Layout**

```
0 4 16 20
```

**IA32 Assembly**

```
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
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
```

This lecture

- Arrays
- Structures
- Alignment
- Unions
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

- **4 bytes:** int, float, char *, ...
  - lowest 2 bits of address must be 00

- **8 bytes:** 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
  - Windows, 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

- **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

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>i[0]</th>
<th>i[1]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>3 bytes</td>
<td></td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
<td>p+24</td>
</tr>
</tbody>
</table>

Multiple of 4

Multiple of 8

Multiple of 8

---

Different Alignment Conventions

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

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

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

Multiple of 4

Multiple of 8

Multiple of 8

---

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

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>i[0]</th>
<th>i[1]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>1 byte</td>
<td></td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+12</td>
<td>p+20</td>
</tr>
</tbody>
</table>

Multiple of 4

Multiple of 8

Multiple of 8
Saving Space

- Put large data types first

```c
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

- Satisfy alignment requirement for every element

```c
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

<table>
<thead>
<tr>
<th>a[0]</th>
<th>• • • a[i]</th>
<th>• • •</th>
</tr>
</thead>
</table>
a+0   a+12i

\[ i \text{ bytes} \quad v \quad j \text{ bytes} \]
a+12i   a+12i+8

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

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

This lecture

- Arrays
- Structures
- Alignment
- Unions
**Union Allocation**

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

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

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

**Using Union to Access Bit Patterns**

```c
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```

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

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

Same as (float) 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**

```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]\n",
    dw.i[0], dw.i[1]);

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

Byte Ordering on IA32

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>
<tbody>
<tr>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
</tr>
</tbody>
</table>

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

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

Long 0 == [0x%lx]

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

BigEndian

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

Output on Sun:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [0xf7f6f5f4f3f2f1f0]

Byte Ordering on x86-64

LittleEndian

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

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