Arrays
Basic Data Types

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

<table>
<thead>
<tr>
<th>Intel</th>
<th>ASM</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</td>
</tr>
<tr>
<td>quad word</td>
<td>q</td>
<td>8</td>
<td>[unsigned] long int (x86-64)</td>
</tr>
</tbody>
</table>

- Floating Point
  - Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Intel</th>
<th>ASM</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>10/12/16</td>
<td>long double</td>
</tr>
</tbody>
</table>

Array Allocation

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

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

Basic Principle

\( T \times \{ L \}; \)

- Array of data type \( T \) and length \( L \)
- Identifier \( A \) can be used as a pointer to array element 0

\[
\begin{array}{c|cccccccc}
1 & 5 & 2 & 1 & 3 \\
x & x+4 & x+8 & x+12 & x+16 & x+20 \\
\end{array}
\]

<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

typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig umn = { 5, 5, 4, 5, 5 };
Array Accessing Example

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)

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

```c
zip_dig cmu;
```

![Array Accessing Example](image)

Memory Reference Code

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

Array Loop Example (IA32)

```c
void zincr(zip_dig z) {
    int i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

```c
# edx = z
movl $0, %eax            # %eax = i
.L4:                      # loop:
    addl $1, (%edx,%eax,4) # z[i]++
    addl $1, %eax          # i++
    cmpl $5, %eax          # i:5
    jne .L4                # if !=, goto loop
```
Pointer Loop Example (IA32)

```c
void zincr_p(zip_dig z) {
    int *zend = z+ZLEN;
    do {
        (*z)++;
        z++;
    } while (z != zend);
}

void zincr_v(zip_dig z) {
    void *vz = z;
    int i = 0;
    do {
        (*((int *) (vz+i)))++;
        i += ISIZE;
    } while (i != ISIZE*ZLEN);
}
```

```assembly
# edx = z = vz
movl $0, %eax
.L8:  # loop:
addl $1, (%edx,%eax)  # Increment vz+i
addl $4, %eax  # i += 4
cmpl $20, %eax  # Compare i:20
jne .L8  # if !=, goto loop
```

Referencing Examples

<table>
<thead>
<tr>
<th>zip_dig cmu;</th>
<th>1</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
</tr>
<tr>
<td>zip_dig mit;</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td>52</td>
</tr>
<tr>
<td>zip_dig umn;</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>56</td>
<td>60</td>
<td>64</td>
<td>68</td>
<td>72</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>5</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

2/12/15  CSCI 2021
Nested Array Example

```c
#define PCOUNT 4
zip_dig mpls[PCOUNT] = 
    {{5, 5, 4, 5, 5},
     {5, 5, 4, 1, 3},
     {5, 5, 4, 1, 4},
     {5, 5, 4, 4, 5}};
```

Declaration "zip_dig mpls[4]" equivalent to "int mpls[4][5]"

- Variable `mpls` denotes an array of 4 elements
- Allocated contiguously
- Each element is an array of 5 int's
- Allocated contiguously
- "Row-Major" ordering of all elements guaranteed

```c
zip_dig mpls[4];
```

```
5 5 4 5 6 5 5 4 1 3 5 5 4 1 4 5 4 5 5 4 4 5
```

Nested Array Allocation

Declaration: `T A[R][C];`

- Array of data type `T`
- `R` rows, `C` columns
- Type `T` element requires `K` bytes

Array Size

- `R * C` bytes

Arrangement

- Row-Major Ordering

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

```
A[0][0]   ...   A[0][C-1]
    .       .       .
    .       .       .
A[R-1][0] ... 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 $A + i \times C \times K$

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

**Diagram:**

```
| 0 | 1 | 2 | 3 | ...
|---|---|---|---|---
| 0 |   |   |   |   |
|   |   |   |   |   |
| C-1| C-1| C-1| C-1| C-1|
```

A $\rightarrow$ A[0] $\rightarrow$ A+i*C*4 $\rightarrow$ A+(R-1)*C*4

2/12/15  CSCI 2021  13

Nested Array Row Access Code

**Code:**

```c
int *get_mpls_zip(int index) {
    return mpls[index];
}
```

Row Vector

- mpls[index] is array of 5 int's
- Starting address mpls+20*index

**Code:**

- Computes and returns address
- Compute as mpls + 4*(index+4*index)

```asm
  # %eax = index
  leal (%eax,%eax,4),%eax # 5 * index
  leal mpls(,%eax,4),%eax # mpls + (20 * index)
```

2/12/15  CSCI 2021  14
## Nested Array Element Access

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

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

### Nested Array Element Access Code

**Array Elements**
- $mpls[index][dig]$ is int
- **Address**: $mpls + 20 \ast index + 4 \ast dig$

**Code**
- Computes address
- $mpls + 4 \ast dig + 4 \ast (index + 4 \ast index)$
- `movl` performs memory reference

```plaintext
int get_mpls_digit(int index, int dig) {
    return mpls[index][dig];
}
```

```plaintext
# %ecx = dig
# %eax = index
leal 0(%ecx,4),%edx # 4*dig
leal (%eax,%eax,4),%eax # 5*index
movl mpls(%edx,%eax,4),%eax # *(mpls + 4*dig + 20*index)
```
Strange Referencing Examples

zip_dig mpls[4];

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

Code does not do any bounds checking
Ordering of elements within array guaranteed

Multi-Level Array Example

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

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

```c
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig umn = { 5, 5, 4, 5, 5 };
```
Element Access in Multi-Level Array

Element access $\text{Mem[Mem[univ } + 4 \times \text{index} ] + 4 \times \text{dig]}$

Must do two memory reads
• First get pointer to row array
• Then access element within array

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

```c
movl 8(%ebp), %eax  # index
movl univ(,%eax,4), %edx  # p = univ[index]
movl 12(%ebp), %eax  # dig
movl (%edx,%eax,4), %eax  # p[dig]
```

Array Element Accesses

Similar C references

Different address computation

Nested Array

Multi-Level Array

```
int get_mpls_digit (int index, int dig)
{
    return mpls[index][dig];
}
```

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

Element at
$\text{Mem[mpls+20*index+4*dig]}$

Element at
$\text{Mem[univ+4*index+4*dig]}$
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></td>
<td>Yes</td>
</tr>
<tr>
<td>univ[1][5]</td>
<td>16+4*5 = 36</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>univ[2][-1]</td>
<td>56+4*-1 = 52</td>
<td></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></td>
<td>No</td>
</tr>
</tbody>
</table>

Code does not do any bounds checking
Ordering of elements in different arrays not guaranteed

N X N Matrix

- Fixed dimensions
  - Know value of N at compile time

- Variable dimensions, explicit indexing
  - Traditional way to implement dynamic arrays

- Variable dimensions, implicit indexing
  - Now supported by gcc

```c
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele
  (fix_matrix a, int i, int j){
    return a[i][j];
}
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele
  (int n, int *a, int i, int j) {
    return a[IDX(n,i,j)];
}
/* Get element a[i][j] */
int var_ele
  (int n, int a[n][n], int i, int j) {
    return a[i][j];
}
Dynamic Nested Arrays

Can create matrix of arbitrary size
Must do index computation explicitly
• 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];
}
```

```assembly
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)]
```

16 X 16 Matrix Access

**Array Elements**
- Address $A + i \times (C \times K) + j \times K$
- $C = 16, K = 4$

```c
/* Get element a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
    return a[i][j];
}
```

```assembly
movl 12(%ebp), %edx   # i
sall $6, %edx          # i*64
movl 16(%ebp), %eax   # j
sall $2, %eax          # j*4
addl 8(%ebp), %eax    # a + j*4
movl (%eax,%edx), %eax # *(a + j*4 + i*64)
```
**n x n Matrix Access**

- **Array Elements**
  - Address $A + i \cdot (C \cdot K) + j \cdot K$
  - $C = n, K = 4$

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

```assembly
movl 8(%ebp), %eax  # n
sall $2, %eax      # n*4
movl %eax, %edx    # n*4
imull 16(%ebp), %edx # i*n*4
movl 20(%ebp), %eax # j
sall $2, %eax      # j*4
addl 12(%ebp), %eax # a + j*4
movl (%eax,%edx), %eax # *(a + j*4 + i*n*4)
```

---

**Optimizing Fixed Array Access**

- **Computation**
  - Step through all elements in column $j$

- **Optimization**
  - Retrieving successive elements from single column

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

/* Retrieve column $j$ from array */
void fix_column (fix_matrix a, int j, int *dest) {
    int i;
    for (i = 0; i < N; i++)
        dest[i] = a[i][j];
}
```

Fixed array: Addr + 64
Optimizing Fixed Array Access

**Optimization**
- Compute \( ajp = \&a[i][j] \)
- Initially = \( a + 4*j \)
- Increment by \( 4*N \)

```c
/* Retrieve column j from array */
void fix_column(fix_matrix a, int j, int *dest {
    int i;
    for (i = 0; i < N; i++)
        dest[i] = a[i][j];
}
```

**Register Value**
- %ecx ajp
- %ebx dest
- %edx i

```
.L8:  # loop:
    movl (%ecx), %eax  # Read *ajp
    movl %eax, (%ebx,%edx,4) # Save in dest[i]
    addl $1, %edx  # i++
    addl $64, %ecx  # ajp += 4*N
    cmpl $16, %edx  # i:N
    jne .L8  # if !=, goto loop
```

Optimizing Variable Array Access

**Compute** \( ajp = \&a[i][j] \)
- Initially = \( a + 4*j \)
- Increment by \( 4*n \)

```c
/* Retrieve column j from array */
void var_column (int n, int a[n][n],
    int j, int *dest)
{
    int i;
    for (i = 0; i < n; i++)
        dest[i] = a[i][j];
}
```

**Register Value**
- %ecx ajp
- %edi dest
- %edx i
- %ebx 4*n
- %esi n

```
.L18:  # loop:
    movl (%ecx), %eax  # Read *ajp
    movl %eax, (%edi,%edx,4) # Save in dest[i]
    addl $1, %edx  # i++
    addl $ebx, %ecx  # ajp += 4*n
    cmpl $esi, %edx  # n:i
    jg .L18  # if >, goto loop
```
Memory Layout

Memory Allocation Example

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 ... */
}   Where does everything go?
### IA32 Example Addresses

*address range ~2^32*

- `$esp` 0xffffbcd0
- `p3` 0x65586008
- `p1` 0x55585008
- `p4` 0x1904a110
- `p2` 0x1904a008
- `&p2` 0x18049760
- `&beyond` 0x08049744
- `big_array` 0x18049780
- `huge_array` 0x08049760
- `main()` 0x080483c6
- `useless()` 0x08049744
- `final malloc()` 0x006be166

`malloc()` is dynamically linked
address determined at runtime

### x86-64 Example Addresses

*address range ~2^67*

- `$rsp` 0x000007ff0008d1f8
- `p3` 0x00002aaabaadd010
- `p1` 0x00002aaaaaadc010
- `p4` 0x0000000011501120
- `p2` 0x0000000011501010
- `&p2` 0x0000000010500a60
- `&beyond` 0x00000000000500a44
- `big_array` 0x0000000010500a80
- `huge_array` 0x00000000000500a50
- `main()` 0x00000000000400510
- `useless()` 0x00000000000400500
- `final malloc()` 0x0000000386ae6a170

not drawn to scale
Heterogeneous Data Structures

Structure Allocation

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

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

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

- **Accessing Structure Member**
  - Pointer indicates first byte of structure
  - Access elements with offsets

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

### Generating Pointer to Structure Member

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

```asm
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```
Following Linked List

- C Code

```c
void set_val(struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->n;
    }
}
```

```
.L17:
    # loop:
    movl 12(%edx), %eax  # r->i
    movl %ecx, (%edx,%eax,4)  # r->a[i] = val
    movl 16(%edx), %edx  # r = r->n
    testl %edx, %edx  # Test r
    jne .L17  # If != 0 goto loop
```

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 (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

Specific Cases of Alignment (x86-64)

- **1 byte:** char, ...
  - no restrictions on address
- **2 bytes:** short, ...
  - lowest 1 bit of address must be 0
- **4 bytes:** int, float, ...
  - lowest 2 bits of address must be 00
- **8 bytes:** double, char *, ...
  - Windows & Linux:
    - lowest 3 bits of address must be 000
  - Linux:
    - lowest 3 bits of address must be 000
    - i.e., treated the same as a 8-byte primitive data type
- **16 bytes:** long double
  - Linux:
    - lowest 3 bits of address must be 000
    - i.e., treated the same as a 8-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

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

Different Alignment Conventions

x86-64 or IA32 Windows:
- K = 8, due to double element

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

IA32 Linux
- K = 4; double treated like a 4-byte data type

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

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

- `p` must be a multiple of 4 for Linux.

<table>
<thead>
<tr>
<th>x</th>
<th>i[0]</th>
<th>i[1]</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+8</td>
<td>p+12</td>
<td>p+16</td>
</tr>
</tbody>
</table>

Windows: `p+24`

Linux: `p+20`

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

- `p` must be a multiple of 4.

<table>
<thead>
<tr>
<th>x[0]</th>
<th>x[1]</th>
<th>i[0]</th>
<th>i[1]</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+12</td>
<td>p+16</td>
</tr>
</tbody>
</table>

### 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];
```

```
a[1].i  a[1].v  a[1].j |
+---------------------+
| a+12 | a+16 | a+20 | a+24 |

<table>
<thead>
<tr>
<th>a[0]</th>
<th>a[1]</th>
<th>a[2]</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+0</td>
<td>a+12</td>
<td>a+24</td>
<td>a+36</td>
</tr>
</tbody>
</table>
```
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

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

(Note: `movswl` loads a 16-bit value into a 32-bit register with sign-extension)

Satisfying Alignment within Structure

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];
```
Union Allocation

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;

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

(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;
}

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
  - 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;
```

<table>
<thead>
<tr>
<th></th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>c[0]</td>
<td>c[1]</td>
<td>c[2]</td>
</tr>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td>i[0]</td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32-bit

<table>
<thead>
<tr>
<th></th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>c[0]</td>
<td>c[1]</td>
<td>c[2]</td>
</tr>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td>i[0]</td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

64-bit
### 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%x]\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>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></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>

**Output:**

- **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

Big 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></td>
<td>i[0]</td>
<td></td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output on Sun:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
Ints 0-1 == [0xf0f1f2f3, 0xf4f5f6f7]
Long 0 == [0xf0f1f2f3]

Byte Ordering on x86-64

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></td>
<td>i[0]</td>
<td></td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output on x86-64:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf0f1f2f3, 0xf4f5f6f7]
Ints 0-1 == [0xf0f1f2f3f4, 0xf7f6f5f4]
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
- 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