Giving credit where credit is due

- Most of slides for this lecture are based on slides created by Drs. Bryant and O'Hallaron, Carnegie Mellon University.
- I have modified them and added new slides.

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>Type</th>
<th>Size</th>
<th>Size in C</th>
<th>Intel</th>
<th>GAS</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>word</td>
<td>2</td>
<td>2</td>
<td>16</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>double</td>
<td>4</td>
<td>4</td>
<td>32</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Floating Point
- Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Size in C</th>
<th>Intel</th>
<th>GAS</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Double</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Extended</td>
<td>16</td>
<td>16</td>
<td>32</td>
<td>16</td>
<td>16</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

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

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>val</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>val+1</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>x+4</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>x+8</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>x+4</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>x</td>
<td>int</td>
<td>x+4/i</td>
</tr>
</tbody>
</table>
Array Example

```
# if <= goto loop

# z : zend
# zi = *z + 2*(5*zi)

# *z

# 5*zi

# zend = z+4
# z++
```

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

```
int get_digit(zip_dig z)
{
    int dig;
    int xi = 0;
    int *send = x + 4;
    do {
        xi = 10 * xi + z[i];
    } while(xi <= send);
    return xi;
}
```

Array Loop Implementation

```
int zd2int(zip_dig z)
{
    int i;
    int xi = 0;
    int *send = x + 4;
    for (i = 0; i < 5; i++)
    {
        xi = 10 * xi + z[i];
    }
    return xi;
}
```

Array Loop Example

```
int zd2int(zip_dig z)
{
    int i;
    int xi = 0;
    int *send = x + 4;
    for (i = 0; i < 5; i++)
    {
        xi = 10 * xi + z[i];
    }
    return xi;
}
```

Nested Array Example

```
#define PCOUNT 4
int get_digit(zip_dig pgh[PCOUNT])
{
    int *send = x + 4;
    for (i = 0; i < 5; i++)
    {
        xi = 10 * xi + z[i];
    }
    return xi;
}
```

Notes

- Declaration "zip_dig cmu[5]" equivalent to "int cmu[5]"
- Each element is an array of 5 int's
- Not guaranteed relative allocation of different arrays
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

\[ R \times C \times K \text{ bytes} \]

Arrangement

- Row-Major Ordering

\[ A[0][0] \quad A[0][C-1] \quad A[R-1][0] \quad A[R-1][C-1] \]

Array Size

\[ R \times C \times K \text{ bytes} \]

Nested Array Row Access

Row Vectors

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

Array Size

\[ R \times C \times K \text{ bytes} \]

Nested Array Row Access Code

\begin{verbatim}
int get_pgh_digit(int index, int dig) {
    return pgh[index][dig];
}
\end{verbatim}

Row Vector

- \( pgh[index] \) is array of \( 5 \) int's
- Starting address \( pgh + 20 \times \text{index} \)

Code

- Computes and returns address
- Compute as \( pgh + 4 \times \text{(index+4*index)} \)

\begin{verbatim}
# %eax = index
lea (pgh, %eax, 4) # 4*dig
lea pgh, %eax # pgh + (20 * index)
\end{verbatim}

Nested Array Element Access

Array Elements

- \( A[i][j] \) is element of type \( T \)
- Address \( A + i \times C \times K \)

Array Size

\[ R \times C \times K \text{ bytes} \]

Nested Array Element Access Code

\begin{verbatim}
int get_pgh_digit(int index, int dig) {
    return pgh[index][dig];
}
\end{verbatim}

Array Elements

- \( pgh[index][dig] \) is int
- Address:
  \[ pgh + 20 \times \text{index} + 4 \times \text{dig} \]

Code

- Computes address
  \[ pgh + 4 \times \text{dig} + 4 \times (\text{index+4*index}) \]
- movl performs memory reference

\begin{verbatim}
# %eax = index
lea (pgh, %eax, 4) # 4*dig
lea pgh, %eax # pgh + (20 * index)
\end{verbatim}

Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( pgh[3][3] )</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
</tr>
<tr>
<td>( pgh[2][5] )</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
</tr>
<tr>
<td>( pgh[2][-1] )</td>
<td>76+20<em>2+4</em>(-1) = 112</td>
<td>3</td>
</tr>
<tr>
<td>( pgh[4][-1] )</td>
<td>76+20<em>4+4</em>(-1) = 152</td>
<td>1</td>
</tr>
<tr>
<td>( pgh[0][18] )</td>
<td>76+20<em>0+4</em>18 = 152</td>
<td>1</td>
</tr>
<tr>
<td>( pgh[0][-1] )</td>
<td>76+20<em>0+4</em>(-1) = 72</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
- Each pointer points to array of int's

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

Element Access in Multi-Level Array

- Element access
  - Mem[univ[univ[4*index]+4*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];
}
```

Array Element Accesses

- Similar C references
- Different address computation

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

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
typedef int fix_matrix[N][N];
```

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

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

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 *p;
};
```

Accessing Structure Member

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

Generating Pointer to Struct. Member

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

Generating Pointer to Array Element

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

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

Optimizing Dynamic Array Mult.

Optimizations
- Performed when set optimization level to -O2
- Code Motion
  - Expression i*n can be computed outside loop
  - Incrementing j has effect of incrementing j*n+k by n

Code Motion
```
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;
jTnK = n;
for (j = 0; j < n; j++)
  result += a[i*n+j] * b[jTnK];
jTnK += n;
return result;
```

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

```
void set_p(struct rec *r)
{
  r->p = &r->a[r->i];
}
```
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
- 4 bytes (e.g., int, float, char *, etc.)
  - lowest 2 bits of address must be 00
- 8 bytes (e.g., double)
  - Windows (and most other OS’s & instruction sets):
    - lowest 2 bits of address must be 00
    - i.e., treated the same as a 4-byte primitive data type
  - 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)
  - lowest 2 bits of address must be 00
  - 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

<table>
<thead>
<tr>
<th>p</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>Multiple of 8</td>
<td>Multiple of 8</td>
<td>Multiple of 8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example (under Linux):
- K = 4; double treated like a 4-byte data type

<table>
<thead>
<tr>
<th>p</th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>Multiple of 8</td>
<td>Multiple of 8</td>
<td>Multiple of 8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ordering Elements Within Structure

<table>
<thead>
<tr>
<th>p</th>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Arrays of Structures

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

<table>
<thead>
<tr>
<th>a[0]</th>
<th>a[1]</th>
<th>a[2]</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td>i[2]</td>
<td></td>
</tr>
<tr>
<td>v</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[0]</td>
<td>s[1]</td>
<td>s[2]</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[0]</td>
<td>s[1]</td>
<td>s[2]</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[0]</td>
<td>s[1]</td>
<td>s[2]</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Accessing Element within Array

- Compute offset to start of structure
- Compute 12i as 4*(i+2)
- Offset by 8
- Access element according to its offset within structure
- Assembler gives displacement as a + 8
- Linker must set actual value

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

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 s[1];
    float f[1];
} s[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;
```

Using Union to Access Bit Patterns

```c
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 (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("Chars 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**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
</tbody>
</table>

**Output on Pentium:**

Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts 0-3 == [0xf0f1f2f3,0xf4f5f6f7]
Ints 0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long 0 == [0x0f0f1f2f3]

---

Byte Ordering on Sun

**Big Endian**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
</tbody>
</table>

**Output on Sun:**

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

---

Byte Ordering on Alpha

**Little Endian**

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
<td>MSB</td>
<td>LSB</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
</tbody>
</table>

**Output on Alpha:**

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

---

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