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>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1</td>
</tr>
<tr>
<td>word</td>
<td>2</td>
</tr>
<tr>
<td>double</td>
<td>4</td>
</tr>
</tbody>
</table>

Floating Point
- Stored & operated on in floating point registers

<table>
<thead>
<tr>
<th>Type</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>4</td>
</tr>
<tr>
<td>Double</td>
<td>8</td>
</tr>
<tr>
<td>Extended</td>
<td>10/12</td>
</tr>
</tbody>
</table>

Array Allocation

Basic Principle

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

<table>
<thead>
<tr>
<th>char string[12];</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>x+4</td>
</tr>
<tr>
<td>x+8</td>
</tr>
<tr>
<td>x+12</td>
</tr>
<tr>
<td>x+16</td>
</tr>
<tr>
<td>x+20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int val[5];</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>x+4</td>
</tr>
<tr>
<td>x+8</td>
</tr>
<tr>
<td>x+12</td>
</tr>
<tr>
<td>x+16</td>
</tr>
<tr>
<td>x+20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>double a[4];</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>x+4</td>
</tr>
<tr>
<td>x+8</td>
</tr>
<tr>
<td>x+12</td>
</tr>
<tr>
<td>x+16</td>
</tr>
<tr>
<td>x+20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>char *p[3];</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>x+4</td>
</tr>
<tr>
<td>x+8</td>
</tr>
</tbody>
</table>

Array Access

Basic Principle

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

<table>
<thead>
<tr>
<th>int val[5];</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>x+4</td>
</tr>
<tr>
<td>x+8</td>
</tr>
<tr>
<td>x+12</td>
</tr>
<tr>
<td>x+16</td>
</tr>
<tr>
<td>x+20</td>
</tr>
</tbody>
</table>

<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+j</td>
</tr>
</tbody>
</table>
**Register Array Loop Implementation**

- **Registers**
  - \( \text{move } x \)
  - \( \text{move } \text{send} \)

- **Computations**
  - \( 10 * z_i + x \)
  - \( z = 2 * (z + 4) \)
  - \( z + \) increments by 4

**Notes**
- Declaration 
  - `int sz dig[5];`
  - `sz dig cu = (1, 5, 2, 1, 3);`
  - `sz dig mit = (0, 2, 1, 3, 9);`
  - `sz dig uc = (9, 4, 7, 2, 0);`

**Array Accessing Example**

- **Computation**
  - `int get_digit (sz dig x, int dig)`
    - Return `x[dig];`

- **Memory Reference Code**
  - `# define P COUNT 4`
  - `sz dig pgh[P COUNT] =`
    - `(0, 1, 0, 0, 0),`
    - `(1, 5, 2, 1, 3),`
    - `(1, 5, 2, 1, 1),`
  - `sz dig pgh[4];`

- **Nested Array Example**
  - Declaration `sz dig 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 requires K bytes

Array Size
- R*C*K bytes

Arrangement
Row-Major Ordering

int A[R][C];

A[0][0] A[1][0] A[2][0] • • • A[R-1][0]
• • •

Nested Array Row Access

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

int A[R][C];

A[0][0] • • • A[R-1][0]
A[0][1] • • • A[R-1][1]
A[0][2] • • • A[R-1][2]
• • •
A[0][C-1] • • • A[R-1][C-1]

Nested Array Row Access Code

int *get_pgh_digit(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)

# texx = index
lea (texx,texx,4),texx # 5*index
lea pgh[texx,texx,4],texx # pgh + (20*index)

Nested Array Element Access

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

int A[R][C];

A[0][0] • • • A[R-1][0]
A[0][1] • • • A[R-1][1]
A[0][2] • • • A[R-1][2]
• • •
A[0][C-1] • • • A[R-1][C-1]

Nested Array Element Access Code

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

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

# texx = dig
lea (texx,texx,4),texx # 4*dig
lea (texx,texx,4),texx # 5*index
movl pgh[texx,texx,4],texx # *(pgh + 4*dig + 20*index)

Strange Referencing Examples

sig_digit pgh[4];

Reference Address Value Guaranteed?
pgh[3][3] 76+20*3+4*3 = 148 2 Yes
pgh[2][5] 76+20*2+4*5 = 136 1 Yes
pgh[2][-1] 76+20*2+4*-1 = 112 3 Yes
pgh[4][-1] 76+20*4+4*-1 = 152 1 Yes
pgh[0][19] 76+20*0+4*19 = 152 1 Yes
pgh[0][-1] 76+20+4*-1 = 72 ?? No
- 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`

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

```
mit = 16; cmu = 17; ucb = 18;
```

```
xip, dig cmu = ( 1, 2, 1, 3 );
xip, dig mit = ( 0, 2, 1, 3, 9 );
xip, dig ucb = ( 9, 4, 7, 2, 0 );
```

**Element Access in Multi-Level Array**

- 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

```
# hекс = index
# tex = dig
load 0, tex, 4; texa # 4*index
movl texa(texa, 4), tex # Mem[univ+4*index]
```

**Array Element Accesses**

- Similar C references
- Nested Array
- Multi-Level Array

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

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

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

**Strange Referencing Examples**

```
Reference Address Value Guaranteed?
univ(2)[3] 564*5 = 68 2 Yes
univ(1)[5] 164*5 = 36 0 No
univ(2)[-1] 564*1 = 52 9 No
univ(1)[-1] 164*12 = 64 7 No

* 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

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

```
/* Compute element k,k of fixed matrix product */
int fix_prodAle ( 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;
}
```

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

```
int get_pgh_digit (int index, int dig)
{ return pgh[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
Dynamic Array Multiplication

Without Optimizations
- Multiplicities
  - 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 vec_prodc Ale 
(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

- Multiplicities
  - 2 for subscripts
  - 1 for data
- Adds
  - 4 for array indexing
  - 1 for loop index
  - 1 for data

```
int v a r _ p r o d _ e l e 
(int * a ,  int * b ,
  int i ,  int k ,  int n )
{
  int j ;
  int r e s u l t  = 0 ;
  f o r  ( j  = 0 ;  j < n ;  j + + )
    r e s u l t  + =
      a [ i * n + j ]  *  b [ j * n + k ] ;
  r e t u r n  r e s u l t ;
}
```

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

```
0 4 16
```

Accessing Structure Member

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

Assembly

```
$ teax = val$
$ tedx = r$
movl teax,(tedx)  $ Mem[r] = val
```

Generating Pointer to Struct. Member

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

Alignement

```
$ teax = idx$
$ tdx = r$
leal 0,(teax),teax $ 4*idx
leal 4,(teax,teax),teax $ r+4*idx4
movl teax,16(tedx) $ Update r->p
```

Alignement

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

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

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

```
$ teax = idx$
$ tdx = r$
leal 0,(teax),teax $ 4*idx
leal 4,(teax,teax),teax $ r+4*idx4
movl teax,16(tedx) $ Update r->p
```

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

```
int j ;
int result = 0 ;
for ( j = 0 ;  j < n ;  j++)
  result +=
    a[i*n+j] * b[j*n+k]; 
return result;
```
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 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)
  - lowest 2 bits of address must be 00
  - i.e., treated the same as a 4-byte primitive data type

Satiation with Structures

Offsets Within Structure
- Must satisfy element’s alignment requirements

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
- i.e., must be multiple of 8

Overall Alignment Requirement

Arrays of Structures

Principle
- Allocated by repeating allocation
- In general, may nest arrays & structures to arbitrary depth
Accessing Element within Array

- Compute offset to start of structure
  - Compute 12\'s as 4\(\times\)4
- 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;
}
```

Satisfying Alignment within Structure

Achieving Alignment

- Starting address of structure array must be multiple of worst-case alignment for any element
  - \(\alpha\) must be multiple of 4
- Offset of element within structure must be multiple of element's alignment requirement
  - \(\nu\)'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;
} a[10];
```

Union Allocation

Principles

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

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

Byte Ordering Revisited

Idea

- Short/long/quad words stored in memory as 2/4/8
- 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

Using Union to Access Bit Patterns

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

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

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

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

```
((0) | (1) | (2) | (3) | (4) | (5) | (6) | (7)
| (8) | (9) | (10) | (11) | (12) | (13) | (14) | (15)
| (16) | (17) | (18) | (19) | (20) | (21) | (22) | (23)
| (32) | (33) | (34) | (35) | (36) | (37) | (38) | (39)
| (40) | (41) | (42) | (43) | (44) | (45) | (46) | (47)
| (48) | (49) | (50) | (51) | (52) | (53) | (54) | (55)
| (56) | (57) | (58) | (59) | (60) | (61) | (62) | (63)

```
Byte Ordering Example (Cont.).

```c
int j;
for (j = 0; j < 8; j++)
dw.c[j] = 0xf0 + j;
printf("Characters 0-7 ==
[Data, Data, Data, Data, Data, Data, Data, Data]\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 ==
[Data, Data, Data, Data]\n",
dw.s[0], dw.s[1], dw.s[2], dw.s[3]);
printf("Ints 0-1 ==
[Data, Data]\n",
dw.i[0], dw.i[1]);
printf("Long 0 ==
[Data]\n",
dw.l[0]);
```

Byte Ordering on x86

Little Endian

```
Output on Pentium:
Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf60, 0xf62, 0xf64, 0xf66, 0xf68]
Ints 0-1 == [0xf3f3f1f0, 0xf7f6f5f4]
Long 0 == [f3f2f1f0]
```

Byte Ordering on Sun

Big Endian

```
Output on Sun:
Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf60, 0xf62, 0xf64, 0xf66, 0xf68]
Ints 0-1 == [0xf3f3f1f0, 0xf7f6f5f4]
Long 0 == [f3f2f1f0]
```

Byte Ordering on Alpha

Little Endian

```
Output on Alpha:
Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf60, 0xf62, 0xf64, 0xf66, 0xf68]
Ints 0-1 == [0xf3f3f1f0, 0xf7f6f5f4]
Long 0 == [f3f2f1f0]
```

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 (indirect)
- 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