CS 105
“Tour of the Black Holes of Computing”

Machine-Level Programming IV:
Structured Data

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>Intel</th>
<th>GAS</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</td>
</tr>
</tbody>
</table>

Floating Point
- Stored & operated on in floating-point registers (not covered in CS 105)

<table>
<thead>
<tr>
<th>Intel</th>
<th>GAS</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>
</tbody>
</table>

Array Allocation

Basic Principle

T A[1];
- Array of data type T and length L
- Contiguously allocated region of L * sizeof(T) bytes in memory

<table>
<thead>
<tr>
<th>char string[12]</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x+12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int val[5]</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x+4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>double a[2]</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x+8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>char *p[2]</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x+8</td>
</tr>
</tbody>
</table>

Array Access

Basic Principle

T A[1];
- 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>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x+4</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[5]</td>
<td>int</td>
<td>x</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int</td>
<td>x+4</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</td>
</tr>
</tbody>
</table>
Array Example

```c
int cmu[5] = {1, 5, 2, 1, 3};
int mit[5] = {0, 2, 1, 3, 9};
int hmc[5] = {9, 1, 7, 1, 1};
```

Note:
- Example arrays were allocated in successive 20-byte blocks
  - Not guaranteed to happen in general
- Here, [5] could be omitted because initializer implies size

Array Accessing Example

```c
int cmu[5];
```

```asm
int get_digit(int z[], int digit) {
  return z[digit];
}
```

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></td>
</tr>
<tr>
<td>mit[5]</td>
<td>36 + 4* 5 = 56</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>mit[-1]</td>
<td>36 + 4*-1 = 32</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15 = 76</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

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

Referencing Examples

```c
int cmu[5];
```

```asm
void zincr(int z[5]){  
  size_t i;  
  for (i = 0; i < 5; i++)  
    z[i]++;  
}
```

Array Loop Example

```asm
void zincr(int z[5]){  
  size_t i;  
  for (i = 0; i < 5; i++)  
    z[i]++;  
}
```

```asm
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  
  # %eax = z[digit]
```

As argument, size of z doesn’t need to be specified
- Register %rdi contains starting address of array
- Register %rsi contains array index
- Desired digit at %rdi + 4*%rsi
- Use memory reference (%rdi, %rsi, 4)
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
  
  int \( A[R][C] \);

Nested Array Element Access

Nested Array Example

```
define PCOUNT 4
int pgh[PCOUNT][5] =
       {{1, 5, 2, 0, 6},
        {1, 5, 2, 1, 3},
        {1, 5, 2, 1, 7},
        {1, 5, 2, 1, 1}};
```

```
typedef int pgh[4][5];
```

```
int pgh[4][5];
```

Variable \( pgh \): array of 4 elements, allocated contiguously
- Each element is an array of 5 ints, allocated contiguously
- “Row-Major” ordering of all elements in memory

Nested Array Row Access

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

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

```
\( A + (i \times (C \times 4)) \)
```

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

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

```
\( A + (i \times (C \times 4)) \)
```

```
\( A + (i \times (C + j) \times 4) \)
```
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></td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>pgh[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</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

```
int cmu[] = {1, 5, 2, 1, 3};
int mit[] = {0, 2, 1, 3, 9};
int hmc[] = {9, 1, 7, 1, 1};
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, hmc};
```

Element Accesses in Multi-Level Array

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

Array Element Accesses

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

Univ 0

```
Pal $2, %rsi # 4*digit
addq  univ(%rdi,8), %rsi # p = univ[index] + 4*digit
movl  (%rsi), %eax # return *p
```

Univ 1

```
Pal $2, %rdi # 8*index
addq  univ(%rdi,8), %rdi # p = univ[index] + 4*digit
```

Element access Mem[Mem[univ+8*index]+4*digit]

- Must do two memory reads
- First get pointer to row array
- Then access element within array
Strange Referencing Examples

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

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

16 x 16 Matrix Access

- Array Elements
  - Address A + i*C + j*K
  - C = 16, K = 4

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

N x N Matrix Access

- Array Elements
  - Address A + j*C + i*K
  - C = n, K = 4
  - Must perform integer multiplication

```
#define N 16
typedef int fix_matrix[N][N];
#define IDX(n, i, j) ((i) * (n) + (j))

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

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

N x N Matrix Code

Fixed dimensions
- Know value of N at compile time

Variable dimensions, explicit indexing
- Traditional way to implement dynamic arrays
- Now supported by gcc

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

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

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

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

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

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

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

Structure represented as block of memory
- Big enough to hold all of the fields

Fields ordered according to declaration
- Even if another ordering could yield a more compact representation

Compiler determines overall size + positions of fields
- Machine-level program has no understanding of the structures in the source code

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

**Generating Pointer to Structure Member**

Generating Pointer to Array Element
- Offset of each structure member determined at compile time
- Compute as \( r + 4 \times idx \)

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

**Alignment Principles**

**Aligned Data**
- Primitive data type requires \( k \) bytes
- Address must be multiple of \( k \)
- Required on some machines; advised on x86-64

**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 trickier when datum spans 2 pages

**Compiler**
- Inserts gaps in structure to ensure correct alignment of fields
Structures & Alignment

Unaligned Data
- Primitive data type requires $K$ bytes
- Address must be multiple of $K$

Aligned Data
- Primitive data type requires $K$ bytes
- Address must be multiple of $K$

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 0
8 bytes: double, long, char *, ...
   - lowest 3 bits of address must be 00
16 bytes: long double (GCC on Linux)
   - lowest 4 bits of address must be 0000

Satisfying Alignment Within Structures
- Within structure:
  - Must satisfy each 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:
- $K = 8$, due to double element

Meeting Overall Alignment Requirement
- For largest alignment requirement $K$
- Overall structure must be multiple of $K$
**Arrays of Structures**

Overall structure length multiple of K
Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```

**Accessing Array Elements**

Compute array offset 12*idx

**Saving Space**

Put large data types first

Effect (K=4)

```
struct S4 {       struct S5 {
  char c;
  int i;
  char d;
} *p;

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

**Union Allocation**

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

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```
**Using Union to Access Bit Patterns**

```c
typedef union {
    float f;
    unsigned int 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;
}
```

**Byte Ordering Revisited**

**Idea**
- Short/long/quad words (x86 terminology; C: short/int/long) stored in memory as
  2/4/8 consecutive bytes
- Which byte is most (least) significant?
- Can cause problems when exchanging binary data between machines

**Big Endian**
- Most significant byte has lowest address
- Sparc; Internet

**Little Endian**
- Least significant byte has lowest address
- Intel x86, ARM Android and IOS

**Bi Endian**
- Can be configured either way
- ARM

---

**Byte Ordering Example**

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

```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]);
```

```c
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]);
```
Summary of Compound Types in C

Arrays
- 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
- Designed to support polymorphic structures
- Way to circumvent type system