Machine-Level Prog. IV - Structured Data

Today
- Arrays
- Structures
- Unions

Next time
- Buffer overflow, x86-64
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>
</tbody>
</table>

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

<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>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12</td>
<td>long double</td>
</tr>
</tbody>
</table>
Array allocation

- $T \ A[L]$;
  - 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
  
  $T \ A[L]$;
  
  – Identifier $A$ can be used as a pointer to array element 0

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

  ![Array access diagram]

  Reference | Type     | Value
  ----------|----------|------
  val[4]    | int      | 3
  val       | int *    | x
  val+1     | int *    | x + 4
  &val[2]   | int *    | x + 8
  val[5]    | int      | ?
  *(val+1)  | int      | 5
  val + i   | int *    | x + 4 i
Array example

defined int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig nu = { 6, 0, 2, 0, 8 };

Notes
- Declaration “zip_dig nu” equivalent to “int nu[5]”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array accessing example

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

---

**Memory reference code**

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

```assembly
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```
**Referencing examples**

- 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><code>mit[3]</code></td>
<td>$36 + 4 \times 3 = 48$</td>
<td>3</td>
<td><strong>Yes</strong></td>
</tr>
<tr>
<td><code>mit[5]</code></td>
<td>$36 + 4 \times 5 = 56$</td>
<td>6</td>
<td><strong>No</strong></td>
</tr>
<tr>
<td><code>mit[-1]</code></td>
<td>$36 + 4 \times (-1) = 32$</td>
<td>3</td>
<td><strong>No</strong></td>
</tr>
<tr>
<td><code>cmu[15]</code></td>
<td>$16 + 4 \times 15 = 76$</td>
<td>??</td>
<td><strong>No</strong></td>
</tr>
</tbody>
</table>

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

- **Original Source**
  Computes the integer represented by an array of 5 decimal digits.

```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 version**
  As generated by GCC
  - Eliminate loop variable `i` and uses pointer arithmetic
  - Computes address of final element and uses that for test
  - Express in do-while form
    - No need to test at entrance

```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

- **Registers**
  - \%ecx  \( z \)
  - \%eax  \( zi \)
  - \%ebx  \( zend \)

- **Computations**
  - \( 10 \times zi + \times z \) implemented as \( \times z + 2 \times (zi+4 \times zi) \)
  - \( z++ \) 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    # zi = 0
leal 16(%ecx),%ebx # zend = z+4
.L59:
    leal (%eax,%eax,4),%edx # 5*zi
    movl (%ecx),%eax     # *z
    addl $4,%ecx         # z++
    leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
    cmpl %ebx,%ecx       # z : zend
    jle .L59             # if <= goto loop
```
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} };
```

• Declaration "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  – Variable `pgh` denotes array of 4 elements
    • Allocated contiguously
  – 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];
  ```
Nested array row access

- Row vectors
  - \( A[i] \) is array of \( C \) elements
  - Each element of type \( T \)
  - Starting address \( A + i \times C \times K \) \( (\text{sizeof}(T) = K) \)

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

Diagram:

- \( A[0] \)
- \( A[i] \)
- \( A[R-1] \)
Nested array row access code

```c
int *get_pgh_zip(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)`

```assembly
# %eax = index
leal (%eax,%eax,4),%eax    # 5 * index
leal pgh(,%eax,4),%eax     # pgh + (20 * index)
```
Nested array element access

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

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

```
A[0][0]  ...  A[0][C-1]
A[1][0]  ...  A[i][j]  ...  A[R-1][0]
A[1][C-1]  ...  A[i][j]  ...  A[R-1][C-1]
```

- Address calculation:
  - \( A + i \times C \times 4 \)
  - \( A + (i \times C + j) \times 4 \)
  - \( A + (R-1) \times C \times 4 \)
Nested array element access code

- **Array Elements**
  - pgh[index][dig] is int
  - Address:
    
    \[ \text{pgh} + 4 \times (5 \times \text{index} + \text{dig}) = \text{pgh} + 20 \times \text{index} + 4 \times \text{dig} \]

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

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

```assembly
# %ecx = dig
# %eax = index
leal 0(%ecx,4),%edx       # 4*dig
leal (%eax,%eax,4),%eax   # 5*index
movl pgh(%edx,%eax,4),%eax # *(pgh + 4*dig + 20*index)
```
Strange referencing examples

```
zip_dig pgh[4];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pgh[3][3]</code></td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][5]</code></td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][-1]</code></td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[4][-1]</code></td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][19]</code></td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][-1]</code></td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>No</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
#include <stdio.h>

#define UCOUNT 3

int *univ[UCOUNT] = {mit, cmu, nu};
```

```c
zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig mit = { 0, 2, 1, 3, 9 };  
zip_dig nu = { 6, 0, 2, 0, 8 };  
```
Element access in multi-level array

- **Computation**
  - **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

```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]
```
Array element accesses

Similar C references

● Nested Array

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

● Element at

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

Different address computation

● Multi-Level Array

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

● Element at

Mem[Mem[univ+4*index]+4*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
#include <stdio.h>

int main()
{
    int i, j, k;
    fix_matrix a, b;
    int result = 0;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
            {
                result += a[i][j] * b[j][k];
            }
    printf("Result: %d\n", result);
    return 0;
}
```

```c
#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 arbitrary 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];
}
```

```
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)]
```
**Structures**

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

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

- **Accessing structure member**

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

**Memory Layout**

```
0  4  16  20
  i  a  p
```

**Assembly**

```
# %eax = val
# %edx = r
movl %eax,(%edx)  # Mem[r] = val
```
Generating pointer to struct. member

- Generating pointer to array element
  - Offset of each structure member determined at compile time

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

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

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```
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
```
Alignment

- **Aligned data**
  - Primitive data type requires K bytes
  - Address must be multiple of K (typically 2, 4 or 8)
  - 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
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

```
struct S1 {
  int i;
  char c;
  int j;
} *p;
```

Impossible to satisfy 4-byte alignment requirement for both $i$ and $j$

Compiler inserts a 3-byte gap to solve this
Arrays of structures

- Principle
  - Allocated by repeating allocation for array type
  - In general, may nest arrays & structures to arbitrary depth
  - Compiler may need to add padding to ensure each element satisfies its alignment requirements

```c
struct S2 {
    int i;
    int j;
    char c;
} a[4];
```
Union allocation

- 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;
```

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *sp;
```
Using union to access bit patterns

```c
unsigned copy (unsigned u) {
    return u;
}
```

```c
unsigned float2bit(float f) {
    union {
        float f;
        unsigned u;
    } temp;
    temp.f = f;
    return temp.u;
}
```

- Store it using one type & access it with another one
- 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`

There's no type info in assembly code!

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %eax
leave
ret
```
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