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></th>
<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></td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td>2</td>
<td></td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>l</td>
<td>4</td>
<td></td>
<td>[unsigned] int</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<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></td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td>8</td>
<td></td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12</td>
<td></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];
```

- **Reference**

<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>
```

Diagram:
```
\[\begin{array}{cccccccc}
1 & 5 & 2 & 1 & 3 \\
& x & x + 4 & x + 8 & x + 12 & x + 16 & x + 20
\end{array}\]"
Array example

```c
typedef 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 * 4`
  - Use memory reference `(%edx, %eax, 4)`

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

Memory reference code

```
# %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>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>6</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
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  zendi

• Computations
  – 10*zi + *z implemented as
    *z + 2*(zi+4*zi)
  – z++ increments by 4

```c
int zd2int(zip_digit z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

```assembly
; %ecx = z
xorl %eax,%eax
leal 16(%ecx),%ebx
.L59:
    leal (%eax,%eax,4),%edx
    movl (%ecx),%eax
    addl $4,%ecx
    leal (%eax,%edx,2),%eax
    cmpl %ebx,%ecx
    jle .L59
```

# %ecx = z
# zi = 0
# zend = z+4

# 5*zi
# *z
# z++
# zi = *z + 2*(5*zi)
# z = zend
# 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 }};
```

  - 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**
  
  - \( R \times C \times K \) bytes

- **Arrangement**
  
  - Row-Major Ordering

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

| A[0][0] | ⋯ | ⋯ | A[0][C-1] |
| A[1][0] | ⋯ | ⋯ | ⋯ |
| A[R-1][0] | ⋯ | ⋯ | ⋯ |

4*R*C Bytes
Nested array row access

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

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

![Diagram showing array access](image)
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];
```

```
<table>
<thead>
<tr>
<th>A[0]</th>
<th>⋮</th>
<th>A[0][C-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>⋮</td>
<td>A[0][C-1]</td>
</tr>
<tr>
<td>A</td>
<td>⋮</td>
<td>A[R-1][C-1]</td>
</tr>
</tbody>
</table>
```

```
A+i*C*4
A+(i*C+j)*4
A+(R-1)*C*4
```
Nested array element access code

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

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

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

```asm
# %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

```c
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><code>76 + 20*3 + 4*3 = 148</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][5]</code></td>
<td><code>76 + 20*2 + 4*5 = 136</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][-1]</code></td>
<td><code>76 + 20*2 + 4*-1 = 112</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[4][-1]</code></td>
<td><code>76 + 20*4 + 4*-1 = 152</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][19]</code></td>
<td><code>76 + 20*0 + 4*19 = 152</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][-1]</code></td>
<td><code>76 + 20*0 + 4*-1 = 72</code></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

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

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, nu};
```
Element access in multi-level array

- **Computation**
  - **Element access** $\text{Mem[Mem[univ+4*index]+4*dig]}$
  - 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];
}
```

```assembly
# %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
#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

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

- **Accessing structure member**

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

**Memory Layout**

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

**Assembly**

```
# %eax = val
# %edx = r
movl %eax, (%edx)  # Mem[r] = val
```

Since the offset of `i` is 0, the address of `r->i` is `r`
Generating pointer to struct. member

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

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

```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
```
Structure referencing (Cont.)

- C Code

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

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

- C Code (Continued)

```
# %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
```

- Structure referencing (Cont.)

```
<table>
<thead>
<tr>
<th>Element i</th>
<th>i</th>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>
```

- Structure referencing (Cont.)

```
<table>
<thead>
<tr>
<th>Element i</th>
<th>i</th>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>
```

- Structure referencing (Cont.)

```
<table>
<thead>
<tr>
<th>Element i</th>
<th>i</th>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>
```

- Structure referencing (Cont.)

```
<table>
<thead>
<tr>
<th>Element i</th>
<th>i</th>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>
```
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**

- **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;
```

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

- Store it using one type & access it with another one
- Get direct access to bit representation of float
- `float2bit` generates bit pattern from float
  - Same code as the one generated for `copy`
  - NOT the same as `(unsigned) f`

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

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

There’s no type info in assembly code!

```
pushl %ebp
movl %esp, %ebp
movl 8(%ebp), %eax
popl %ebp
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