Basic Data Types

Integral
- Stored and 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 int</td>
</tr>
</tbody>
</table>

Floating Point
- Stored and 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

**Basic Principle:** \( T \ A[L] \)
- Array (named A) of data type \( T \) and length \( L \).
- Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes.

```plaintext
char string[12];
int val[5];
double a[3];
char *p[3];
```

Array Access

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

<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+j</td>
<td>int *</td>
<td>x + 4j</td>
</tr>
</tbody>
</table>

*Note the use of pointer arithmetic.*
Array Example

```c
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

Example arrays were allocated in successive 20 byte block. That’s not guaranteed to happen in general.

Declaration `zip_dig cmu` is equivalent to `int cmu[5]`.

Array Accessing Example

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

Computation
- Register `%rdi` contains the starting address of the array.
- Register `%rsi` contains the array index.
- The desired digit is at `%rdi + (4 * %rsi)`.
- User memory reference (%rdi, %rsi, 4).

Memory Reference Code

```c
# %rdi = z
# %rsi = dig
movl (%rdi, %rsi, 4), %eax # z[dig]
```

Array Loop Example

```c
void zincr ( zip_dig z )
{
    size_t i;
    for ( i = 0; i < ZLEN; i++ )
        z[i]++;
}
```

Declarations: `T A[R][C];`
- 2D 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 (guaranteed)

Row major order means the elements are stored in the following order:

\[ [A_0,0, \ldots, A_0,C-1, A_1,0, \ldots, A_1,C-1, \ldots, A_{R-1},0, \ldots, A_{R-1},C-1] \]
Multidimensional Array Access

**Declaration:**  \( T \ A[R][C]; \)
- 2D array of data type \( T \)
- \( R \) rows, \( C \) columns
- Type \( T \) element requires \( K \) bytes

To access element \( A[i][j] \), perform the following computation:

\[
A + i \times C \times K + j \times K
\]

Nested Array Example

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

```
76 86 96 156
```

- Declaration “`zip_digit pgh[4]`” is equivalent to “`int pgh[4][5].`”
- Variable \( pgh \) denotes an array of 4 elements allocated contiguously.
- Each element is an array of 5 ints, which are allocated contiguously.
- This is “row-major” ordering of all elements, guaranteed.

Nested Array Row Access

**Row Vectors:**
Given a nested array declaration \( T \ A[R][C] \), you can think of this as an array of arrays.

- \( A[i] \) is an array of \( C \) elements.
- Each element of \( A[i] \) has type \( T \), and requires \( K \) bytes.
- The starting address of \( A[i] \) is \( A + i \times C \times K \).

Nested Array Element Access

**Array Elements**
- \( A[i][j] \) is an element of type \( T \), which requires \( K \) bytes.
- The address is \( A + (i \times C + j) \times K \).
Multi-Level Array Example

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

Variable `univ` denotes an array of 3 elements.
Each element is a pointer (8 bytes).
Each pointer points to an array of ints (may vary in length; i.e., ragged array is possible).

<table>
<thead>
<tr>
<th>cmu</th>
<th>mit</th>
<th>ucb</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

Computation
- Element access
  - `Mem[Mem[univ+8*index] + 4*dig]`
- Must do two memory reads:
  - First get pointer to row array.
  - Then access element within the row.

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

Element at `Mem[Mem[univ+8*index]+4*dig]`

 Nested Array

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

Element at `Mem[pgh+20*index+4*dig]`

Similar C references, but different address computations.

Multi-Level Array

```c
# define N 16
typed 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];
}
```

Fixed dimensions:
- Know value of `N` at compile time.

Variable dimensions, explicit indexing:
- Traditional way to implement dynamic arrays

Variable dimensions, implicit indexing:
- Now supported by gcc

```c
#define IDX(n, i, j) (((i) * (n)) + (j))
/* Get element a[i][j] */
int vec_ele( size_t n, int *a, size_t i, size_t j ) {
    return a[IDX(n, i, j)];
}
```

```c
#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];
}
```

```c
#define IDX(n, i, j) (((i) * (n)) + (j))
/* Get element a[i][j] */
int vec_ele( size_t n, int *a, size_t i, size_t j ) {
    return a[IDX(n, i, j)];
}
```

```c
#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];
}
```
16 x 16 Matrix Access

Array Elements
- Address $A + i \cdot (C \cdot K) + j \cdot K$
- $C = 16, K = 4$

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

```c
/* 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];
}
```

/*
  # a in %rdi, i in %rsi, j in %rdx
  salq $6, %rsi       # 64*i
  addq %rsi, %rdi     # a + 64*i
  movl (%rdi, %rdx, 4), %eax # M[a + 64*i + 4*j]
*/

Structure Representation
- Structure represented as block of memory
  - Big enough to hold all the fields
- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
- Compiler determines overall size and position of fields
  - Machine-level program has no understanding of the structures in the source code.

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

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

Generating Pointer to Structure Member
- Offset of each structure member determined at compile time
- Compute as $r + 4*idx$

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

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

BTW: why does r->i need 8 bytes? Alignment. (Next slide set)
Aside on Structures: Arrow vs. Dot

If you have a pointer \( r \) to a structure, use \( r->x \) to access component \( x \).

If you have the structure \( s \) itself, use \( s.x \).

\( r->x \) is just syntactic sugar for \( (*r).x \)

```c
void set_val
    (struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->next;
    }
}
```

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

```
.L11:  testq  %rdi, %rdi       # Test r
       .L12   je .L12           # if = 0, goto done
       movq  16(%rdi), %rax    # i = M[r+16]
       movl  %esi, (%rdi, %rax, 4)  # M[r+4*i] = val
       movq  24(%rdi), %rdi     # r = M[r+24]
       jmp .L11
.L12:
```

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

```
.L11:  testq  %rdi, %rdi       # Test r
       .L12   je .L12           # if = 0, goto done
       movq  16(%rdi), %rax    # i = M[r+16]
       movl  %esi, (%rdi, %rax, 4)  # M[r+4*i] = val
       movq  24(%rdi), %rdi     # r = M[r+24]
       jmp .L11
.L12:
```