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.

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

Array Access

```c
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

Example arrays were allocated in successive 20 byte block.

That's not guaranteed to happen in general.

Array Accessing Example

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

```
#define ZLEN 5
typedef int zip_digit[ZLEN];
zip_digit cmu = { 1, 5, 2, 1, 3, 9 };
zip_digit mit = { 0, 2, 1, 3, 9, 4 };
zip_digit ucb = { 9, 4, 7, 2, 0, 0 };
```

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

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 (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 \( \text{A}[i][j] \), perform the following computation:
\[ A + i \times C \times K + j \times K \]

Nested Array Example

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

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

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).

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

Element Access in a Multi-Level Array

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

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.

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

Array Element Accesses

Nested Array

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

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

```
#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];
}
```

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

```
#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)];
}
```

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

```
#define var
/* 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];
}
```

Variable dimensions, implicit indexing:
- Now supported by gcc
Array Elements
- Address \( A + i \times (C \times K) + j \times K \)
- \( C = 16, K = 4 \)

```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_j ) {
    return a[i][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.

Generating Pointer to Structure Member
- Offset of each structure member determined at compile time
- Compute as \( r + 4 \times i \times dx \)

```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 $(\ast r).x$