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>
</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>
**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.

```
char string[12];
int val[5];
double a[3];
char *p[4];
```
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>

Array Access

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+j int * x + 4j
```c
typedef int zip_dig[5];

type definition:

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

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

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];
}
```

Memory Reference Code

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

Computation

- Register `%edx` contains the starting address of the array.
- Register `%eax` contains the array index.
- The desired digit is at \((4 \times %eax) + %edx\).
- User memory reference \((%edx,%eax,4)\).
- Code does not do any bounds checking!
- Out of range behavior is implementation-dependent.
- There is no guaranteed relative allocation of different arrays.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>mit[3]</td>
<td>$36 + 4 \times 3 = 48$</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit[5]</td>
<td>$36 + 4 \times 5 = 56$</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>mit[-1]</td>
<td>$36 + 4 \times (-1) = 32$</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>$16 + 4 \times 15 = 76$</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>
Array Loop Example

Original Source

```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.
- Eliminates loop variable `i`.
- Converts array code to pointer code.
- Expresses 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

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

- `%ecx` holds `z`
- `%eax` holds `zi`
- `%ebx` holds `zend`

Computations

- "(10*zi + *z)" implemented as "2*(zi+4*zi) + *zi."
- `z++` increments by 4.

```asm
# %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 # compare z : zend
.jle .L59 # if <= goto loop
```
#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]” 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.
**Declaration:** \( T \ A[R][C] \)

- Array of (element) data type \( T \).
- \( R \) rows, \( C \) columns
- Assume type \( T \) element requires \( K \) bytes.

**Array Size:** \( R \times C \times K \)

**Arrangement:** row-major ordering

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

\[
[ A_{0,0}, \ldots, A_{0,C-1}, A_{1,0}, \ldots, A_{1,C-1}, \ldots, A_{R-1,0}, \ldots, A_{R-1,C-1} ]
\]
Given an nested array declaration $A[R][C]$, you can think of this as an array of arrays.

- $A[i]$ is an array of $C$ elements.
- Each element has type $T$.
- The starting address is $A + i \times C \times K$.  

\[ A_0[0] \quad \ldots \quad A_0[C-1] \quad \ldots \quad A_i[0] \quad \ldots \quad A_i[C-1] \quad \ldots \quad A_{R-1}[0] \quad \ldots \quad A_{R-1}[C-1] \]

\[ A + 0 \times C + 4A \quad \ldots \quad A + i \times C + 4A \quad \ldots \quad A + (R-1) \times C \times 4 \]
int *get_pgh_zip(int index)
{
    return pgh[index];
}

Row Vector

- pgh[index] is an array of 5 ints.
- The starting address is pgh + 20*index.

Code

- Computes and returns the address.
- Compute this as pgh + 4*(index + 4*index).

```assembly
# %eax holds the index
leal (%eax,%eax,4),%eax  # 5 * index
leal pgh(%eax,4),%eax    # pgh + (20 * index)
```
**Array Elements**

- $A[i][j]$ is an element of type $T$.
- The address is $A + (i \times C + j) \times K$.
```c
int get_pgh_zip_dig(int index, int dig)
{
    return pgh[index][dig];
}
```

Array Elements

- pgh[index][dig] is an int.
- The address is pgh + 20*index + 4*dig.

Code

- Computes address pgh + 4*dig + 4*(index + 4*index).
- movl then performs the memory reference.

```
# %ecx holds dig
# %eax holds 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 = 1 5 2 0 6 1 5 2 1 3 1 5 2 1 7 1 5 2 2 1
76 96 116 136 156
pgh[4];
```

- Code does not do any bounds checking!
- Ordering of elements within array is guaranteed.

### Reference Table

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pgh[3][3]</code></td>
<td>$76 + 20 \times 3 + 4 \times 3 = 148$</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][5]</code></td>
<td>$76 + 20 \times 2 + 4 \times 5 = 136$</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[2][-1]</code></td>
<td>$76 + 20 \times 2 + 4 \times (-1) = 112$</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[4][-1]</code></td>
<td>$76 + 20 \times 4 + 4 \times (-1) = 152$</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][19]</code></td>
<td>$76 + 20 \times 0 + 4 \times 19 = 152$</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td><code>pgh[0][-1]</code></td>
<td>$76 + 20 \times 0 + 4 \times (-1) = 72$</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>
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.
- Each pointer points to an array of ints.
Element Access in a Multi-Level Array

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

**Computation**

- **Element access**
  \[ \text{Mem}[\text{Mem}[\text{univ}+4\times\text{index}]+4\times\text{dig}] \]

- **Must do two memory reads:**
  - First get pointer to row array.
  - Then access element within the row.

```assembly
# %ecx = index
# %eax = dig
lea 0(,%ecx,4),%edx   # 4 * index
movl univ(%edx),%edx   # Mem[univ+4*index]
movl (%edx,%eax,4),%eax # Mem[...+4*dig]
```

CS429 Slideset 10: 18
Instruction Set Architecture V
Array Element Accesses

Nested Array

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

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

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]

Similar C references, but different address computations.
- Code does not do any bounds checking.
- Ordering of elements in different arrays is not guaranteed.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>univ[2][3]</code></td>
<td>$56 + 4 \times 3 = 68$</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td><code>univ[1][5]</code></td>
<td>$16 + 4 \times 5 = 36$</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[2][-1]</code></td>
<td>$56 + 4 \times (-1) = 52$</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[3][-1]</code></td>
<td>??</td>
<td>??</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[1][12]</code></td>
<td>$16 + 4 \times 12 = 64$</td>
<td>7</td>
<td>No</td>
</tr>
</tbody>
</table>
Using Nested Arrays

**Strengths**
- C compiler handles doubly subscripted arrays.
- Generates very efficient code.

**Limitation**
- It only works with fixed array sizes.

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

/* Compute element i,i of fixed matrix product. */
int fix_prod_elem
(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;
}
```
**Concept**
- Contiguously-allocated region of memory.
- Refer to members within the structure by name.
- Members may be of different types.

```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;
}
```

```
# %eax = val
# %edx = r
movl %eax, (%edx)  # Mem[r] = val
```
Generating Pointer to Struct Member

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

Generating Pointer to an Array Element

- Offset of each structure member is determined at compile time.

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

## Assembly Code

```
# %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];
}
```

```assembly
# %edx = r
movl (%edx),%ecx  # r->i
lea 0(%ecx,4),%eax  # 4*(r->i)
lea 4(%edx,%eax),%eax  # r+4+4*(r->i)
movl %eax,16(%edx)  # update r->p
```