Machine-Level Prog. IV - Structured Data

Today
- Arrays
- Structures
- Unions

Next time
- Buffers
**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>1</td>
<td>[unsigned] char</td>
</tr>
<tr>
<td>word</td>
<td>w</td>
<td>2</td>
<td>2</td>
<td>[unsigned] short</td>
</tr>
<tr>
<td>double word</td>
<td>l</td>
<td>4</td>
<td>4</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>4</td>
<td>float</td>
</tr>
<tr>
<td>Double</td>
<td>l</td>
<td>8</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>Extended</td>
<td>t</td>
<td>10/12</td>
<td>10/12</td>
<td>long double</td>
</tr>
</tbody>
</table>
Array allocation

- Basic principle

\[
T \ A[L];
\]
- Array of data type \( T \) and length \( L \)
- Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes

---

example:

```c
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

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

![Diagram of 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 + i` | `int *` | `x + 4 i`
Array example

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 $4 \times %eax + %edx$
  - Use memory reference (%edx, %eax, 4)

**Memory reference code**

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

```asm
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax # z[dig]
```
Code does not do any bounds checking!

Reference | Address | Value | Guaranteed?
--- | --- | --- | ---
mit[3] | $36 + 4 \times 3 = 48$ | 3 | 
mit[5] | $36 + 4 \times 5 = 56$ | 6 | 
mit[-1] | $36 + 4 \times -1 = 32$ | 3 | 
cmu[15] | $16 + 4 \times 15 = 76$ | ?? | 

– Out of range behavior implementation-dependent
  • No guaranteed relative allocation of different arrays
Referencing examples

- Code does not do any bounds checking!

Reference | Address      | Value | Guaranteed?
-----------|--------------|-------|-------------
mit[3]     | 36 + 4* 3 = 48 | 3     | Yes         
mit[5]     | 36 + 4* 5 = 56 | 6     |             
mit[-1]    | 36 + 4*(-1) = 32 | 3     |             
cmu[15]    | 16 + 4*15 = 76 | ??    |             

- Out of range behavior implementation-dependent
  - No guaranteed relative allocation of different arrays
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</td>
<td>48</td>
<td><strong>Yes</strong></td>
</tr>
<tr>
<td>mit[5]</td>
<td>36 + 4* 5</td>
<td>56</td>
<td><strong>No</strong></td>
</tr>
<tr>
<td>mit[-1]</td>
<td>36 + 4*-1</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15</td>
<td>76</td>
<td>??</td>
</tr>
</tbody>
</table>

- Out of range behavior implementation-dependent
  - No guaranteed relative allocation of different arrays
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>$36 + 4 \times 3 = 48$</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit[5]</td>
<td>$36 + 4 \times 5 = 56$</td>
<td>6</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></td>
</tr>
</tbody>
</table>

– Out of range behavior implementation-dependent
  • No guaranteed relative allocation of different arrays
Code does not do any bounds checking!

### Referencing examples

<table>
<thead>
<tr>
<th>Zip Dig</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmu</td>
<td>36 + 4* 3 = 48</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>mit</td>
<td>36 + 4* 5 = 56</td>
<td>6</td>
<td>No</td>
</tr>
<tr>
<td>mit</td>
<td>36 + 4*-1 = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu</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 zend

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

```c
int zd2int(zip_dig 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

# zi = 0
# zend = z+4
# 5*zi
# *z
# z++
# zi = *z + 2*(5*zi)
# z : zend
# if <= goto loop
```
Array loop implementation

- Registers
  - \%ecx z
  - \%eax zi
  - \%ebx zend

- Computations
  - $10 \times zi + \star z$ implemented as $\star z + 2 \times (zi + 4 \times zi)$
  - $z++$ increments by 4

```assembly
# %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
```

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

```assembly
# \%ecx = z
xoral %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
```

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**
  - $10 \times zi + \ast z$ implemented as $\ast z + 2 \times (zi+4\times zi)$
  - $z++$ increments by 4

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

```asm
# %ecx = z
xorl %eax,%eax  # zi = 0
leal 16(%ecx),%ebx  # zend = z+4
.L59:
    leal (%eax,%eax,4),%edx  # 5\times zi
    movl (%ecx),%eax  # \ast z
    addl $4,%ecx  # z++
    leal (%eax,%edx,2),%eax  # zi = *z + 2 \times (5\times zi)
    cmpl %ebx,%ecx  # z : zend
    jle .L59  # if <= goto loop
```
Array loop implementation

- **Registers**
  - %ecx  z
  - %eax  zi
  - %ebx  zend

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

Wednesday, October 19, 2011
Array loop implementation

- **Registers**
  - `%ecx` z
  - `%eax` zi
  - `%ebx` zend

- **Computations**
  - $10 \cdot zi + *z$ implemented as $*z + 2 \cdot (zi+4*zi)$
  - `z++` increments by 4

```assembly
.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

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**
  - $10 \times zi + *z$ implemented as $*z + 2 \times (zi+4 \times zi)$
  - $z++$ increments by 4

```c
int zd2int(zip_dig 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  # 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
```
Array loop implementation

- Registers
  \%ecx $z$
  \%eax $zi$
  \%ebx $zend$

- Computations
  - $10 \cdot zi + *z$ implemented as
    $*z + 2 \cdot (zi+4 \cdot zi)$
  - $z++$ increments by 4

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

```assembly
.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
```

Wednesday, October 19, 2011
Array loop implementation

- Registers
  - %ecx z
  - %eax zi
  - %ebx zend

- Computations
  - $10 \times zi + *z$ implemented as
    - $*z + 2 \times (zi+4 \times zi)$
  - $z++$ increments by 4

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

```assembly
# %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
```
Checkpoint
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
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

\[
\begin{array}{cccc}
A[0][0] & \cdots & \cdots & A[0][C-1] \\
\vdots & \ddots & \vdots & \vdots \\
A[R-1][0] & \cdots & \cdots & A[R-1][C-1]
\end{array}
\]

\[
\begin{array}{cccc}
A[0][0] & \cdots & \cdots & A[0][C-1] \\
\vdots & \ddots & \vdots & \vdots \\
A[R-1][0] & \cdots & \cdots & A[R-1][C-1]
\end{array}
\]
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];
```

\[ A[i][0] \quad \cdots \quad A[i][C-1] \]
\[ A[0][0] \quad \cdots \quad A[0][C-1] \]
\[ A[0] \]

\[ A[R-1][0] \quad \cdots \quad A[R-1][C-1] \]
\[ A[i][0] \quad \cdots \quad A[i][C-1] \]
\[ A+i\times C\times 4 \]

\[ A+(R-1)\times C\times 4 \]
Nested array row access code

- **Row vector**
  - \( pgh[index] \) is array of 5 int’s
  - Starting address \( pgh + 20 \times index \)

- **Code**
  - Computes and returns address
  - Compute as \( pgh + 4 \times (index + 4 \times index) \)

```c
int *get_pgh_zip(int index)
{
    return pgh[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 $A + (i * C + j) * K$

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

A[0] $\ldots$ A[i] $\ldots$ A[R-1]

$A[0][0]$ $\ldots$ $A[i][j]$ $\ldots$ $A[R-1][0]$ $\ldots$ $A[R-1][C-1]$

A+i*C*4 $\uparrow$

$A+(i*C+j)*4$ $\uparrow$

$A+(R-1)*C*4$
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];
}

# %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

```plaintext
zipDig

pgh[4];

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgh[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
</tr>
<tr>
<td>pgh[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
```
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>pgh[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
</tr>
<tr>
<td>pgh[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
## Strange referencing examples

**zip_dig**

```c
pgh[4];
```

```text
<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>3</td>
</tr>
<tr>
<td><code>pgh[4][-1]</code></td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
</tr>
<tr>
<td><code>pgh[0][19]</code></td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
</tr>
<tr>
<td><code>pgh[0][-1]</code></td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
</tr>
</tbody>
</table>
```

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
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></td>
</tr>
<tr>
<td><code>pgh[0][-1]</code></td>
<td><code>76+20*0+4*-1 = 72</code></td>
<td>??</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
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>??</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
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>pgh[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
</tr>
<tr>
<td>pgh[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
## 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

```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 };;

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

```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);
}

int var_ele (int *a, int i, int j, int n)
{
    return a[i*n+j];
}
```
Dynamic array multiplication

- Without optimizations
  - **Multiplies**
    - 2 for subscripts
    - 1 for data
  - **Adds**
    - 4 for array indexing
    - 1 for loop index
    - 1 for data

```c
/* Compute element i,k of variable matrix product */
int var_prod_ele (int *a, int *b, int i, int k, int n)
{
    int j;
    int result = 0;
    for (j = 0; j < n; j++)
        result += a[i*n+j] * b[j*n+k];
    return result;
}
```

Row-wise  A  Column-wise  B

(i,*)  (*,k)
Optimizing dynamic array mult.

- **Optimizations**
  - Performed when set optimization level to `-O2`

- **Code motion**
  - Expression $i \times n$ can be computed outside loop

- **Strength reduction**
  - Incrementing $j$ has effect of incrementing $j \times n + k$ by $n$

- **Performance**
  - Compiler can optimize regular access patterns

```c
{ int j; int result = 0; for (j = 0; j < n; j++) { result += a[i*n+j] * b[j*n+k]; } return result; }
```

```c
{ int j; int result = 0; int iTn = i*n; int jTnPk = k; for (j = 0; j < n; j++) { result += a[iTn+j] * b[jTnPk]; jTnPk += n; } return result; }
```
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;
}
```

Memory Layout

![Memory Layout Diagram]

Assembly

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

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

- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time

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

```assembly
# %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
# %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
```
Checkpoint
Alignment

• Aligned data
  – Primitive data type requires K bytes
  – Address must be multiple of K
  – 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
Specific cases of alignment

- **Size of Primitive Data Type:**
  - **1 byte** (e.g., `char`)
    - no restrictions on address
  - **2 bytes** (e.g., `short`)
    - lowest 1 bit of address must be 0₂
  - **4 bytes** (e.g., `int`, `float`, `char *`, etc.)
    - lowest 2 bits of address must be 00₂
  - **8 bytes** (e.g., `double`)
    - Windows (and most other OS’s & instruction sets):
      - lowest 3 bits of address must be 000₂
    - Linux:
      - lowest 2 bits of address must be 00₂
      - i.e., treated the same as a 4-byte primitive data type
  - **12 bytes** (`long double`) [only 10 bytes needed]
    - Linux and Windows:
      - lowest 2 bits of address must be 00₂
      - i.e., treated the same as a 4-byte primitive data type
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 (under Windows):
  - K = 8, due to double element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

---

Wednesday, October 19, 2011
Linux vs. Windows

- **Windows (including Cygwin):**
  - \( K = 8 \), due to \texttt{double} element

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

- **Linux:**
  - \( K = 4 \); \texttt{double} treated like a 4-byte data type
Overall alignment requirement

```c
struct S2 {
    double x;
    int i[2];
    char c;
} *p;
```

*p must be multiple of:
8 for Windows
4 for Linux

```c
struct S3 {
    float x[2];
    int i[2];
    char c;
} *p;
```

*p must be multiple of 4 (in either OS)

Windows: p+24
Linux: p+20
Ordering elements within structure

```
struct S4 {
    char c1;
    double v;
    char c2;
    int i;
} *p;
```

10 bytes wasted space in Windows

```
struct S5 {
    double v;
    char c1;
    char c2;
    int i;
} *p;
```

2 bytes wasted space
Arrays of structures

- Principle
  - Allocated by repeating allocation for array type
  - In general, may nest arrays & structures to arbitrary depth

```c
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```
Accessing element within array

– Compute offset to start of structure
  • Compute 12*i as 4*(i+2i)
– Access element according to its offset within structure
  • Offset by 8
  • Assembler gives displacement as a + 8
    – Linker must set actual value

```
short get_j(int idx)
{
  return a[idx].j;
}
```

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```

```c
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

```
a[0]   • • •   a[i]   • • •
a+0

a[i].i  a[i].v  a[i].j
a+12i

a+12i    a+12i+8
```
Satisfying alignment within structure

Achieving Alignment

- Starting address of structure array must be multiple of worst-case alignment for any element
  - \( a \) must be multiple of 4
- Offset of element within structure must be multiple of element’s alignment requirement
  - \( v \)'s offset of 4 is a multiple of 4
- Overall size of structure must be multiple of worst-case alignment for any element
  - Structure padded with unused space to be 12 bytes

```c
struct S6 {
    short i;
    float v;
    short j;
} a[10];
```
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;
```

(Windows alignment)
Using union to access bit patterns

```c
typedef union {
    float f;
    unsigned u;
} bit_float_t;

float bit2float(unsigned u) {
    bit_float_t arg;
    arg.u = u;
    return arg.f;
}

unsigned float2bit(float f) {
    bit_float_t arg;
    arg.f = f;
    return arg.u;
}
```

- 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`
Byte ordering revisited

• Idea
  – Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  – Which is most (least) significant?
  – Can cause problems when exchanging binary data between machines

• Big Endian
  – Most significant byte has lowest address
  – PowerPC, Sparc

• Little Endian
  – Least significant byte has lowest address
  – Intel x86, Alpha
Byte ordering example

union {
    unsigned char c[8];
    unsigned short s[4];
    unsigned int i[2];
    unsigned long l[1];
} dw;

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i[0]</td>
<td>i[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
int j;
for (j = 0; j < 8; j++)
dw.c[j] = 0xf0 + j;

printf("Characters 0-7 == [0x%x, 0x%x, 0x%x, 0x
%x, 0x%x, 0x%x, 0x%x, 0x%x]\n",
dw.c[0], dw.c[1], dw.c[2], dw.c[3],
dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x, 0x%x, 0x%x, 0x%x]
\n",
dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x, 0x%x]\n",
dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
dw.l[0]);
Byte ordering on x86

Little Endian

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>f1</td>
<td>f2</td>
<td>f3</td>
<td>f4</td>
<td>f5</td>
<td>f6</td>
<td>f7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LSB  MSB  LSB  MSB  LSB  MSB  LSB  MSB

LSB  MSB
i[0]  i[1]

LSB  MSB
l[0]

Print

Output on Pentium:

Characters 0-7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0-3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Ints 0-1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [f3f2f1f0]
Byte ordering on sun

Big Endian

Output on Sun:

Characters 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0–3 == [0xf0f1, 0xf2f3, 0xf4f5, 0xf6f7]
Ints 0–1 == [0xf0f1f2f3, 0xf4f5f6f7]
Long 0 == [0xf0f1f2f3]
Byte ordering on alpha

Little Endian

Output on Alpha:

Characters 0–7 == [0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7]
Shorts 0–3 == [0xf1f0, 0xf3f2, 0xf5f4, 0xf7f6]
Ints 0–1 == [0xf3f2f1f0, 0xf7f6f5f4]
Long 0 == [0xf7f6f5f4f3f2f1f0]
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 (`zd2int`)
  – 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