# Referencing Examples

<table>
<thead>
<tr>
<th>zip_dig cmu;</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>zip_dig mit;</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td>52</td>
<td>56</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>zip_dig nwu;</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
<th>🔼</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>60</td>
<td>64</td>
<td>68</td>
<td>72</td>
<td>76</td>
<td></td>
</tr>
</tbody>
</table>

## 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>No</td>
</tr>
</tbody>
</table>

- Out of range behavior implementation-dependent
  - No guaranteed relative allocation of different arrays
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

- Eliminate loop variable \(i\)
- Convert array code to pointer code
- 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
- $10\times zi + *z$
  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
```
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

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

```
| A[0][0] | ⋮ | ⋮ | A[0][C-1] |
| ⋮       |   |   | ⋮         |
| ⋮       |   |   | ⋮         |
| A[R-1][0] | ⋮ | ⋮ | A[R-1][C-1] |
```

$4 \times R \times C$ Bytes
Nested Array Row Access

Row Vectors

- $A[i]$ is array of $C$ elements
- Each element of type $T$
- Starting address $A + i \times C \times K$

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>[0]</td>
<td>[i]</td>
<td>[i]</td>
<td>[R-1]</td>
<td>[R-1]</td>
<td>[R-1]</td>
</tr>
<tr>
<td>[0]</td>
<td>[C-1]</td>
<td>[0]</td>
<td>[C-1]</td>
<td>[0]</td>
<td>[C-1]</td>
<td>[C-1]</td>
</tr>
<tr>
<td>$A$</td>
<td></td>
<td>$A+i*4$</td>
<td></td>
<td>$A+(R-1)*4$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)`

```asm
# %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];
```

![Diagram showing array access]

- \( A \)
- \( A+i*C*4 \)
- \( A+(i*C+j)*4 \)
- \( A+(R-1)*C*4 \)
Nested Array Element Access Code

Array Elements
- \( pgh[\text{index}][\text{dig}] \) is int
- Address:
  \[ pgh + 20*\text{index} + 4*\text{dig} \]

Code
- Computes address
  \[ pgh + 4*\text{dig} + 4*(\text{index}+4*\text{index}) \]
- \text{movl} performs memory reference

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

Note: One Memory Fetch

```assembly
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx     # 4*\text{dig}
leal (%eax,%eax,4),%eax  # 5*\text{index}
movl pgh(%edx,%eax,4),%eax  # *(pgh + 4*\text{dig} + 20*\text{index})
```
Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgh[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>pgh[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</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 nwu = { 6, 0, 2, 0, 1 };

#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, nwu};
```
Referencing “Row” in Multi-Level Array

**Row Vector**
- `univ[index]` is pointer to array of `int`'s
- Starting address `Mem[univ+4*index]`

**Code**
- Computes address within `univ`
- Reads pointer from memory and returns it

```assembly
# %edx = index
leal 0(%edx,4),%eax  # 4*index
movl univ(%eax),%eax  # *(univ+4*index)
```

```c
int* get_univ_zip(int index)
{
    return univ[index];
}
```
Accessing Element in Multi-Level Array

Computation

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

```
# %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]
```

Note: Two Memory Fetches
Strange Referencing Examples

Reference | Address     | Value | Guaranteed?
----------|-------------|-------|-------------
univ[2][3] | 56+4*3 = 68 | 0     | Yes
univ[1][5] | 16+4*5 = 36 | 0     | No
univ[2][-1] | 56+4*-1 = 52 | 9     | No
univ[3][-1] | ??          | ??    | No
univ[1][12] | 16+4*12 = 64 | 2     | No

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed
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];
}
```

```assembly
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

- Contiguously_allocated region of memory
- Refer to members within structure by names
- 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;
}
```

Memory Layout

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

Assembly

```
# %eax = val
# %edx = r
movl %eax, (%edx)     # Mem[r] = val
```
Generating Pointer to Structure 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];
}

# %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];
}
```

```
# %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
```
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)
  – Linux:
    » 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

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
## Linux vs. Windows

### Windows (including Cygwin):

- $K = 8$, due to double element

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>p+0</td>
</tr>
<tr>
<td>i[0]</td>
<td>p+4</td>
</tr>
<tr>
<td>i[1]</td>
<td>p+8</td>
</tr>
<tr>
<td>v</td>
<td>p+16</td>
</tr>
<tr>
<td>v</td>
<td>p+24</td>
</tr>
</tbody>
</table>

### Linux:

- $K = 4$; double treated like a 4-byte data type

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>p+0</td>
</tr>
<tr>
<td>i[0]</td>
<td>p+4</td>
</tr>
<tr>
<td>i[1]</td>
<td>p+8</td>
</tr>
<tr>
<td>v</td>
<td>p+12</td>
</tr>
<tr>
<td>v</td>
<td>p+20</td>
</tr>
</tbody>
</table>

Multiple of 4 | Multiple of 8 | Multiple of 8 | Multiple of 8 | Multiple of 4 | Multiple of 4 | Multiple of 4 | Multiple of 4 |
Effect of 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)

```
x          i[0]  i[1]  c
p+0         p+8   p+12  p+16
```

Windows: `p+24`
Linux: `p+20`
Ordering Elements Within Structure

```c
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

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;
}
```

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

```
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

union U1 {
    char c;
    int i[2];
    double v;
} *up;

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

(Windows alignment)
Implementing “Tagged” Union

- Structure can hold 3 kinds of data
- Only one form at any given time
- Identify particular kind with flag `type`

```c
typedef enum { CHAR, INT, DBL } utype;

typedef struct {
  utype type;
  union {
    char c;
    int i[2];
    double v;
  } e;
} store_ele, *store_ptr;

store_ele k;
```

```
  k.type
    k.e
      k.e.c
        k.e.i[0] k.e.i[1]
      k.e.v
    k.e
```
IA32 Floating Point

History
- 8086: first computer to implement IEEE FP
  - separate 8087 FPU (floating point unit)
- 486: merged FPU and Integer Unit onto one chip

Summary
- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

Floating Point Formats
- single precision (C float): 32 bits
- double precision (C double): 64 bits
- extended precision (C long double): 80 bits
FPU Data Register Stack

FPU register format (extended precision)

```
79  78  64 63

s  exp  frac
```

FPU register “stack”

- stack grows down
  - wraps around from R0 -> R7
- FPU registers are typically referenced relative to top of stack
  - st(0) is top of stack (Top)
  - followed by st(1), st(2),…
- push: increment Top, load
- pop: store, decrement Top
- Run out of stack? Overwrite!
FPU instructions

Large number of floating point instructions and formats

- ~50 basic instruction types
- load, store, add, multiply
- sin, cos, tan, arctan, and log!

Sampling of instructions:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fldz</td>
<td>push 0.0</td>
<td>Load zero</td>
</tr>
<tr>
<td>flds S</td>
<td>push S</td>
<td>Load single precision real</td>
</tr>
<tr>
<td>fmuls S</td>
<td>st(0) ← st(0)*S</td>
<td>Multiply</td>
</tr>
<tr>
<td>faddp</td>
<td>st(1) ← st(0)+st(1); pop</td>
<td>Add and pop</td>
</tr>
</tbody>
</table>
Floating Point Code Example

Compute Inner Product of Two Vectors

- Single precision arithmetic
- Scientific computing and signal processing workhorse

```c
float ipf (float x[], float y[], int n)
{
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++) {
        result += x[i] * y[i];
    }
    return result;
}
```

```assembly
pushl %ebp          # setup
movl %esp,%ebp
pushl %ebx
movl 8(%ebp),%ebx   # %ebx=&x
movl 12(%ebp),%ecx  # %ecx=&y
movl 16(%ebp),%edx  # %edx=n
fldz               # push +0.0
xorl %eax,%eax     # i=0
cmpl %edx,%eax      # if i>=n done
jge .L3
.L5:
    flds (%ebx,%eax,4)    # push x[i]
    fmuls (%ecx,%eax,4)   # st(0)*=y[i]
    faddp
    incl %eax
    cmpl %edx,%eax        # if i<n repeat
    jl .L5
.L3:
    movl -4(%ebp),%ebx   # finish
    leave
    ret                   # st(0) = result
```
Inner product stack trace

1. fldz

2. flds (%ebx,%eax,4)

3. fmuls (%ecx,%eax,4)

4. faddp %st,%st(1)

5. flds (%ebx,%eax,4)

6. fmuls (%ecx,%eax,4)

7. faddp %st,%st(1)