Last Time

- Procedures (IA32)
  - call / return
  - %esp, %ebp
  - local variables
  - recursive functions

![Diagram of caller-save, callee-save, and special registers]

Today

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structs
  - Alignment
- Unions
Array Allocation

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

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

Array Access

- Basic Principle
  - \( T \ A[N] \)
  - Array of data type \( T \) and length \( N \)
  - Identifier \( A \) can be used as a pointer to array element 0: Type \( T^* \)

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

- Reference | Type | Value
- --- | --- | ---
- \( \text{val}[4] \) | int | 5
- \( \text{val} \) | int * | \( x \)
- \( \text{val} + 1 \) | int * | \( x + 4 \)
- \( \&\text{val}[2] \) | int * | \( x + 8 \)
- \( \text{val}[5] \) | int | ??
- \( *(\text{val} + 1) \) | int | 8
- \( \text{val} + i \) | int * | \( x + 4 / i \)
Array Example

typedef int zip_dig[5];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

- Declaration “zip_dig uw” equivalent to “int uw[5]”
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

Array Accessing Example

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

IA32

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

- Register %edx contains starting address of array
- Register %eax contains array index
- Desired digit at 4*%eax + %edx
- Use memory reference (%edx, %eax, 4)
Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uw[3]</code></td>
<td>36 + 4* 3 = 48</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td><code>uw[6]</code></td>
<td>36 + 4* 6 = 60</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td><code>uw[-1]</code></td>
<td>36 + 4*-1 = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td><code>cmu[15]</code></td>
<td>16 + 4*15 = 76</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- No bound checking
- Out-of-range behavior implementation-dependent
- No guaranteed relative allocation of different arrays
# Array Loop Example

- **Original**

```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**
  - As generated by GCC
  - Eliminate loop variable \( i \)
  - Convert array code to pointer code
  - Express in do-while form (no 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 (IA32)

```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
```

**Translation?**
Array Loop Implementation (IA32)

- 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;
}
```

Nested Array Example

```
#define PCOUNT 4
zip_dig
sea[PCOUNT] =
{ [ 9, 8, 1, 9, 5 ],
  [ 9, 8, 1, 0, 5 ],
  [ 9, 8, 1, 0, 3 ],
  [ 9, 8, 1, 1, 5 ]};
```

- “zip_dig sea[4]” equivalent to “int sea[4][5]”
  - Variable sea: array of 4 elements, allocated contiguously
  - Each element is an array of 5 ints, allocated contiguously
- “row-major” ordering of all elements guaranteed
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

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

Nested Array Row Access

- **Row vectors**
  - $A[i]$ is array of $C$ elements
  - Each element of type $T$ requires $K$ bytes
  - Starting address $A + i \times (C \times K)$

```c
int A[R][C];
```
Nested Array Row Access Code

```c
int *get_sea_zip(int index) {
    return sea[index];
}
```

```c
#define PCOUNT 4
zip_dig sea[PCOUNT] = {
    { 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 };
```

- What data type is `sea[index]`?
- What is its starting address?

```c
# %eax = index
leal (%eax,%eax,4),%eax
leal sea(,%eax,4),%eax
```

**Translation?**

Nested Array Row Access Code

```c
int *get_sea_zip(int index) {
    return sea[index];
}
```

```c
#define PCOUNT 4
zip_dig sea[PCOUNT] = {
    { 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 };
```

**Row Vector**
- `sea[index]` is array of 5 `int`
- Starting address `sea+20*index`

**IA32 Code**
- Computes and returns address
- Compute as `sea+4*(index+4*index)=sea+20*index`
Nested Array Row Access

- **Array Elements**
  - $A[i][j]$ is element of type $T$, which requires $K$ bytes
  - Address $A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K$

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

```plaintext
  [0]  [0]  [0]
[0]   [1]  [C-1]
```

```plaintext
A + i\times C\times 4
A + (R-1)\times C\times 4
A + i\times C\times 4 + j\times 4
```

Nested Array Element Access Code

```c
int get_sea_digit
(int index, int dig)
{
    return sea[index][dig];
}
```

```plaintext
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx    # 4*dig
leal (%eax,%eax,4),%eax    # 5*index
movl sea(%edx,%eax,4),%eax    # *(sea + 4*dig + 20*index)
```

- **Array Elements**
  - $sea[index][dig]$ is int
  - Address: $sea + 20*index + 4*dig$

- **IA32 Code**
  - Computes address $sea + 4*dig + 4*(index+4*index)$
  - `movl` performs memory reference
Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>sea[3][3]</td>
<td></td>
<td>148</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td></td>
<td>136</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td></td>
<td>112</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td></td>
<td>152</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[0][19]</td>
<td></td>
<td>152</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td></td>
<td>72</td>
<td>No</td>
</tr>
</tbody>
</table>

```
zip_dig
sea[4];
```

- 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 ints

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

```c
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

**Element Access in Multi-Level Array**

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

```c
# %ecx = index
# %eax = dig
leal 0(%ecx,4),%edx
movl univ(%edx),%edx
movl (%edx,%eax,4),%eax
Translation?
```
Element Access in Multi-Level Array

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

- **Computation (IA32)**
  - 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
# %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

Nested array

```c
int get_sea_digit(int index, int dig)
{
    return sea[index][dig];
}
```

Multi-level array

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

Access looks similar, but it isn’t:

Strange Referencing Examples

- Reference: `univ[2][3]`
- Address: 56
- Value: 4*3 = 12
- Guaranteed? Yes

- Reference: `univ[1][5]`
- Address: 16
- Value: 4*5 = 20
- Guaranteed? No

- Reference: `univ[2][-1]`
- Address: 56
- Value: 4*-1 = -4
- Guaranteed? No

- Reference: `univ[3][-1]`
- Address: ??
- Value: ??
- Guaranteed? No

- Reference: `univ[1][12]`
- Address: 16
- Value: 4*12 = 48
- Guaranteed? 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 for 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 any 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];
}
```

```asm
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)]
```
Dynamic Array Multiplication

- **Without Optimizations**
  - Multiplies: 3
    - 2 for subscripts
    - 1 for data
  - Adds: 4
    - 2 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;
}
```

Optimizing Dynamic Array Multiplication

- **Optimizations**
  - Performed when set optimization level to -O2
- **Code Motion**
  - Expression i*n can be computed outside loop
- **Strength Reduction**
  - Incrementing j has effect of incrementing j*n+k by n
- **Operations count**
  - 4 adds, 1 mult
- **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;
}
```
### Structures

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

**Concept**
- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

**Accessing structure member**

```c
void set_i(struct rec *r, int val) {
    r->i = val;
    // (*r).i = val;
}
```

**In java:**
```
r.i = val;
```

### Memory Layout

<table>
<thead>
<tr>
<th>i</th>
<th>a</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>16 20</td>
</tr>
</tbody>
</table>

### Generating Pointer to Structure Member

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

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

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

**In java?**
```
// Array access
```
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];
    // (*r).p = &(*((*(r).a+(*r).i)));  
}
```

In java? ...

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 IA32 Linux, x86-64 Linux, and Windows!

- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system-dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory very tricky when datum spans two pages (later...)

- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment (IA32)

- **1 byte:** `char`, ...
  - no restrictions on address
- **2 bytes:** `short`, ...
  - lowest 1 bit of address must be 0_2
- **4 bytes:** `int`, `float`, `char *`, ...
  - lowest 2 bits of address must be 00_2
- **8 bytes:** `double`, ...
  - Windows (and most other OS’s & instruction sets): lowest 3 bits 000_2
  - Linux: lowest 2 bits of address must be 00_2
    - i.e., treated the same as a 4-byte primitive data type
- **12 bytes:** `long double`
  - Windows, Linux: (same as Linux double)

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy element’s alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K

**Example (under Windows or x86-64):**
- K = 8, due to `double` element

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

- **IA32 Windows or x86-64:**
  - $K = 8$, due to `double` element

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

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

- **IA32 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>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+12</td>
</tr>
</tbody>
</table>

Saving Space

- **Put large data types first**

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

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

- **Effect (example x86-64, both have $K=8$)**

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

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

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

<table>
<thead>
<tr>
<th>v</th>
<th>i[0]</th>
<th>i[1]</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+8</td>
<td>p+16</td>
<td></td>
</tr>
</tbody>
</table>
Arrays of Structures

- Satisfy alignment requirement for every element

```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```

Accessing Array Elements

- Compute array offset 12i
- Compute offset 8 with structure
- Assembler gives offset a+8
  - Resolved during linking

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

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

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

Concept
- Allow same regions of memory to be referenced as different types
- Aliases for the same memory location

Structure Layout
- i  a  p
- 0  4  16  20

Union Layout
- i
- a
- p
- 0  4  12

Union Allocation
- Allocate according to largest element
- Can only use one field at a time
Using Union to Access Bit Patterns

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;
}

Same as (float) u ?

Same as (unsigned) f ?

Summary

- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element’s alignment requirement
  - Pointer to first element
  - No bounds checking

- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment

- **Unions**
  - Overlay declarations
  - Way to circumvent type system