Data Structures in Memory!

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

Data Structures in Assembly...

- Arrays?
- Strings?
- Structs?
### Array Allocation

**Basic Principle**
- `T A[N];`
- Array of data type `T` and length `N`
- Contiguously allocated region of `N * sizeof(T)` bytes

### 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*`

#### Reference | Type | Value
--- | --- | ---
`val[4]` | int |  
`val` | int * |  
`val+1` | int * |  
`&val[2]` | int * |  
`val[5]` | int |  
`*(val+1)` | int |  
`val + i` | int * |  

---

**University of Washington**

23 April 2012  Data Structures
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*

```
int val[5];        
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>1</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>
```

- **Reference**  | **Type** | **Value**
- val            | int *    | x
- val+1          | int *    | x + 4
- &val[2]        | int *    | x + 8
- val[5]         | int      | ??
- *(val+1)       | int      | 8
- val + i        | int *    | x + 4

**Array Example**

```c
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 };```

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];
}

# 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>zip_dig cmu;</th>
<th>1</th>
<th>5</th>
<th>2</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>zip_dig uw ;</th>
<th>9</th>
<th>8</th>
<th>1</th>
<th>9</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td>52</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>zip_dig ucb;</th>
<th>9</th>
<th>4</th>
<th>7</th>
<th>2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>56</td>
<td>60</td>
<td>64</td>
<td>68</td>
<td>72</td>
</tr>
</tbody>
</table>

### Reference

- `uw[6]` 36 + 4* 6 = 60 4
- `uw[-1]` 36 + 4*-1 = 32 3
- `cmu[15]` 16 + 4*15 = 76 ??

- No bound checking
- Out-of-range behavior implementation-dependent
- No guaranteed relative allocation of different arrays
Referencing Examples

| zip_dig cmu; | 1 5 2 1 3 |
| zip_dig uw; | 9 8 1 9 5 |
| zip_dig ucb; | 9 4 7 2 0 |

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

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

Array Loop Example

```c
int zd2int(zip_dig z) {
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```
Array Loop Example

- **Original**

- **Transformed**
  - 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 i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

```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
.L59:
    leal (%eax,%edx,2),%eax
    cmpl %ebx,%ecx
    jle .L59
```

Translation?
Array Loop Implementation (IA32)

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

- Computations
  - 10*z + *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 }};
```
Nested Array Example

```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-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 \* C \* K bytes

- **Arrangement**
  - Row-major ordering
Nested Array Row Access

int A[R][C];

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

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?

```asm
# %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 } };
```

```c
# %eax = index
leal (%eax,%eax,4),%eax # 5 * index
leal sea(,%eax,4),%eax # sea + (20 * index)
```

- **Row Vector**
  - `sea[index]` is array of 5 ints
  - 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

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

A + i*C*4

A + (R-1)*C*4
Nested Array Row Access

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

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

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

```
# %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

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

<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>76+20<em>3+4</em>3 = 148</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td>76+20<em>0+4</em>-1 = 72</td>
<td>??</td>
<td></td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
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>76+20<em>3+4</em>3 = 148</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>5</td>
<td>Yes</td>
</tr>
<tr>
<td>sea[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

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

Same thing as Multi-level array?
Multi-Level Array Example

```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};

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

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of ints
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
### 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:

- `Mem[sea+20*index+4*dig]`
- `Mem[Mem[univ+4*index]+4*dig]`

---

### Strange Referencing Examples

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>univ[2][3]</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>univ[1][5]</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>univ[2][-1]</td>
<td>56</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>univ[3][-1]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>univ[1][12]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What values go here?
### Strange Referencing Examples

<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>$56+4\times3 = 68$</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td><code>univ[1][5]</code></td>
<td>$16+4\times5 = 36$</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td><code>univ[2][-1]</code></td>
<td>$56+4\times-1 = 52$</td>
<td>5</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>$16+4\times12 = 64$</td>
<td>7</td>
<td>No</td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed
Using Nested Arrays

```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;
}
```

**Strengths**
- C compiler handles doubly subscripted arrays
- Generates very efficient code
- Avoids multiply in index computation

**Limitation**
- Only works for fixed array size
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);
}
```

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

```c
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

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

**IA32 Assembly**

```
# %eax = val
# %edx = r
movl %eax, (%edx)  # Mem[r] = val
```

---

Generating Pointer to Structure Member

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

**Memory Layout**

```
0 4 16 20
```

```plaintext
i a p
```
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));
}
```

```
# %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];
    // (*r).p = &(*(*r).a+(*r).i));
}
```

```
# %edx = r
movl (%edx),%ecx # r->i
leal 0(%ecx),%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 IA32 Linux, x86-64 Linux, and Windows!

- What is the motivation for alignment?
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 02

- **4 bytes: int, float, char *, ...**
  - lowest 2 bits of address must be 002

- **8 bytes: double, ...**
  - Windows (and most other OS’s & instruction sets): lowest 3 bits 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**
  - 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

  ![Alignment Diagram]

Different Alignment Conventions

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

  ![Alignment Diagram]

- **IA32 Linux**
  - $K = 4$; double treated like a 4-byte data type

  ![Alignment Diagram]
Saving Space

- Put large data types first

```c
struct S1 {
  char c;
  int i[2];
  double v;
} *p1;
```

- Effect (example x86-64, both have K=8)
Arrays of Structures

- Satisfy alignment requirement for every element

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

---

Arrays of Structures

- Satisfy alignment requirement for every element

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

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

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

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

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

Unions

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

```c
union urec {
    int i;
    int a[3];
    int *p;
};
```

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

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

Structure Layout

Union Layout
Union Allocation

- 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;
```

Using Union to Access Bit Patterns

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

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

```c
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