Computer Systems
CSE 410 Autumn 2013
8 – Data Structures: Arrays and Structs
Roadmap

C:
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);

Java:
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();

Assembly language:
get_mpg:
pushq %rbp
movq %rsp, %rbp
...
popq %rbp
ret

Machine code:
0111010000011000
100011010000010000000010
1000100111000010
11000000111111010100001111

OS:
Windows 8
Mac

Computer system:
Arrays

Memory & data
Integers & floats
Machine code & C
x86 assembly
Procedures & stacks
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C
Arrays & Other Data Structures

- Array allocation and access in memory
- Multi-dimensional or nested arrays
- Multi-level arrays
- Other structures in memory
- Data structures and alignment
### Array Allocation

**Basic Principle**

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

- `char string[12];`
  - `x` to `x + 12`
- `int val[5];`
  - `x` to `x + 4`
  - `x + 4` to `x + 8`
  - `x + 8` to `x + 12`
  - `x + 12` to `x + 16`
  - `x + 16` to `x + 20`
- `double a[3];`
  - `x` to `x + 8`
  - `x + 8` to `x + 16`
  - `x + 16` to `x + 24`
- `char* p[3];`
  - `(or char *p[3];)`
  - `x` to `x + 4`
  - `x + 4` to `x + 8`
  - `x + 8` to `x + 12`

 **IA32**

 **x86-64**
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];
```

Reference | Type | Value
---|---|---
val[4] | int | 5
val | int * | x
val+1 | int * | x + 4
&val[2] | int * | x + 8
val[5] | int | ?? (whatever is in memory at address x + 20)
*(val+1) | int | 8
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 };
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

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

### IA32

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at $4\times%eax + %edx$
- Use memory reference $(%edx, %eax, 4)$
**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>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</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>36</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td>52</td>
<td>56</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>56</td>
<td>60</td>
<td>64</td>
<td>68</td>
<td>72</td>
<td>76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>uw[3]</td>
<td>$36 + 4 \times 3 = 48$</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>uw[6]</td>
<td>$36 + 4 \times 6 = 60$</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>uw[-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>

- No bounds checking
- Location of each separate array in memory is not guaranteed
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**

```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**

- Eliminate loop variable `i`, use pointer `zend` instead
- Convert array code to pointer code
  - Pointer arithmetic on `z`
- 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)

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

- Computations
  - $10\times zi + *z$ implemented as $*z + 2\times(5\times zi)$
  - 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  # zi + 4*zi = 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
```
Arrays & Other Data Structures

- Array allocation and access in memory
- Multi-dimensional or nested arrays
- Multi-level arrays
- Other structures in memory
- Data structures and alignment
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 }};
```

Remember, \( T \ A[N] \) is an array with \( N \) elements of type \( T \).
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 }};
```

Remember, `T A[N]` is an array with `N` elements of type `T`.

- “Row-major” ordering of all elements
- This is 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?**
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

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

- **Row vectors**
  - $T$ \( A[R][C] \): \( A[i] \) 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];
```

![Diagram](nested_arrays.png)
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 };
```
## 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 an array of 5 integers (a `zip_dig` data type)
- 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];
```

![Diagram of nested arrays and row access]

- `A[R-1][C]`: Accessing the last element in the last row.

Nested Arrays
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];
```

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

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

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

- **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>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
Arrays & Other Data Structures

- Array allocation and access in memory
- Multi-dimensional or nested arrays
- Multi-level arrays
- Other structures in memory
- Data structures and alignment
# 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 a 2D array?
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`

```c
#include <stdio.h>

#define UCOUNT 3

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

Note: this is how Java represents multi-dimensional arrays.
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        # 4*index
movl univ(%edx),%edx       # Mem[univ+4*index]
movl (%edx,%eax,4),%eax   # Mem[...+4*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:

```
```
Strange Referencing Examples

- **Reference**
- **Address**
  - `univ[2][3]` 56+4*3 = 68
  - `univ[1][5]` 16+4*5 = 36
  - `univ[2][-1]` 56+4*-1 = 52
  - `univ[3][-1]` ??
  - `univ[1][12]` 16+4*12 = 64
- **Value**
  - 2
  - 9
  - 5
  - ??
  - 7
- **Guaranteed?**
  - Yes
  - No
  - No
  - No
  - No

- Code does not do any bounds checking
- Location of each lower-level array in memory is not guaranteed
Arrays in C

- Contiguous allocations of memory
- No bounds checking
- Can usually be treated like a pointer to first element (elements are offset from start of array)
- Nested (multi-dimensional) arrays are contiguous in memory (row-major order)
- Multi-level arrays are not contiguous (pointers used between levels)
Arrays & Other Data Structures

- Array allocation and access in memory
- Multi-dimensional or nested arrays
- Multi-level arrays
- Other structures in memory
- Data structures and alignment
Structures

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

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

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

Memory Layout

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

Accessing Structure Member

- Given an instance of the struct, we can use the . operator, just like Java:
  - `struct rec r1; r1.i = val;`
- What if we have a pointer to a struct: `struct rec *r = &r1;`
  - Using * and . operators: `(*r).i = val;`
  - Or, use -> operator for short: `r->i = val;`
- Pointer indicates first byte of structure; access members with offsets

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

IA32 Assembly

```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;
};
```

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

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

```c
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax  # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```
Arrays & Other Data Structures

- Array allocation and access in memory
- Multi-dimensional or nested arrays
- Multi-level arrays
- Other structures in memory
- Data structures and alignment
Structures & Alignment

- **Unaligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$

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

  - Multiple of 4
  - Multiple of 8
  - Multiple of 8
  - Multiple of 8
Alignment Principles

- **Aligned Data**
  - Primitive data type requires K bytes
  - Address must be multiple of K

- **Aligned data is required on some machines; it is advised on IA32**
  - Treated differently by IA32 Linux, x86-64 Linux, and Windows!

- **What is the motivation for alignment?**
Alignment Principles

**Aligned Data**
- Primitive data type requires K bytes
- Address must be multiple of K

**Aligned data is required on some machines; it is *advised* on IA32**
- Treated differently by IA32 Linux, x86-64 Linux, and Windows!

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

**Compiler**
- Inserts padding in structure to ensure correct alignment of fields
- `sizeof()` should be used to get true size of structs
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 OSs & 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: lowest 2 bits of address must be $00_2$
Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy every member’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):** $K = ?$
  - $K = 8$, due to `double` member

\[
\begin{array}{cccccc}
\text{c} & \text{3 bytes} & \text{i[0]} & \text{i[1]} & \text{4 bytes} & \text{v} \\
p1+0 & \uparrow & p1+4 & \uparrow & p1+16 & \uparrow & p1+24 \\
\text{Multiple of 8} & \text{Multiple of 4} & \text{Multiple of 8} & \text{Multiple of 8} \\
\end{array}
\]
Different Alignment Conventions

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

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

- **IA32 Linux:** $K = 4$
  - $K = 4$; `double` aligned like a 4-byte data type
Saving Space

- Put large data types first:

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

```c
struct S2 {
    double v;
    int i[2];
    char c;
} *p2;
```

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

Unfortunately, doesn’t satisfy requirement that struct’s total size is a multiple of K
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 12i (sizeof(S3))
- Element j is at offset 8 within structure
- Since a is static array, assembler gives offset a+8

```c
// Global:
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 # a+12*idx+8
```