The Hardware/Software Interface
CSE351 Spring 2015

Lecture 12

Instructor:
Katelin Bailey

Teaching Assistants:
• I hear that some of you don’t like questions
  • Answers are always on the slides if you look
  • But, here’s a new format to help you find it
  • (I won’t stop asking)
  • If one of these doesn’t pop up, ask and I’ll insert it after class in the slides.
  • Answers to questions from previous lectures won’t be put in the slides.
  • Try to think through the questions, and ask if you can’t figure it out.

Why do I ask questions?
- Questions keep you paying attention
- This class isn’t about memorizing material
- Questions make you think about the “why”
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
110000011111101000011111

Computer system:

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

OS:

Windows 8
Mac
Data Structures in Assembly

- **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 * sizeof(T) bytes

```c
char string[12];
```

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

```c
double a[3];
```

```c
char* p[3];
(or char *p[3];)
```

- 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];
  |
  x | x + 4 | x + 8 | x + 12 | x + 16 | x + 20 |
  ---|-------|-------|--------|--------|--------|
  |  9   |  8    |  1    |  9    |  5    |
  ```

• Reference | Type | Value
- `val[4]` | int | 
- `val` | int * |
- `val+1` | int * |
- `&val[2]` | int * |
- `val[5]` | int |
- `*(val+1)` | int |
- `val + i` | int * |
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
  - \( \text{int} \ \text{val}[5]; \)

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{val}[4] )</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>( \text{val} )</td>
<td>int *</td>
<td>( x )</td>
</tr>
<tr>
<td>( \text{val}+1 )</td>
<td>int *</td>
<td>( x + 4 )</td>
</tr>
<tr>
<td>( &amp;\text{val}[2] )</td>
<td>int *</td>
<td>( x + 8 )</td>
</tr>
<tr>
<td>( \text{val}[5] )</td>
<td>int</td>
<td>?? (value in memory at address ( x + 20 ))</td>
</tr>
<tr>
<td>( *(\text{val}+1) )</td>
<td>int</td>
<td>8</td>
</tr>
<tr>
<td>( \text{val} + i )</td>
<td>int *</td>
<td>( x + 4 \times i )</td>
</tr>
</tbody>
</table>

Why are some ints and some int pointers/addresses?
- Anything with \& is an address (or pointer)
- The array itself (\( \text{val} \)) is the address of the first element
- Anything added to an address is also an address
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 };  

initialization

same as int uw[5] ...
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 };
```

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

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

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

IA32

```assembly
# %edx = z
# %eax = dig
movl (%edx,%eax,4),%eax  # z[dig]
```
## Referencing Examples

Why are these not guaranteed?:
- The arrays may not be one after another in memory

### 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>$36 + 4 \times 3 = 48$</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td><code>uw[6]</code></td>
<td>$36 + 4 \times 6 = 60$</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td><code>uw[-1]</code></td>
<td>$36 + 4 \times -1 = 32$</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td><code>cmu[15]</code></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
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}

zi = 10*0 + 9 = 9
zi = 10*9 + 8 = 98
zi = 10*98 + 1 = 981
zi = 10*981 + 9 = 9819
zi = 10*9819 + 5 = 98195
Array Loop Example

- **Original**

- **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 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;
}
```

Why do we convert to do-while format?
- It helps you think through the x86 representation
Array Loop Implementation (IA32)

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

- Computations
  - `10*zi + *z` implemented as `*z + 2*(5*zi)`
  - `z++` increments by 4

```assembly
.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
```

```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

```c
zip_dig sea[4] =
{{ 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 elements of type `T`, with length `N`.

**same as:**

```c
int sea[4][5];
```

What is the layout in memory?
- Do we place all the elements of a row together?
- Or all the elements of a column together?
  
  (... Answer is on the next slide)
Nested Array Example

```cpp
zip_dig sea[4] =
{ { 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 elements of type \( T \), with length \( N \).
Two-Dimensional (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?

```
A[0][0]  •  •  •  A[0][C-1]

  •
  •
  •

A[R-1][0]  •  •  •  A[R-1][C-1]
```
Two-Dimensional (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 is the answer to the last question

```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];
```

![Description of nested array access](image)
int *get_sea_zip(int index) {
    return sea[index];
}

int sea[4][5] =
    {{ 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[2] (note the single index)? … guesses?

What is the starting address for sea[2]? … guesses?
Nested Array Row Access Code

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

```c
int sea[4][5] = {
    { 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[2]` (note the single index)?
... guesses?

What is the starting address for `sea[2]`?
... guesses?

```assembly
# %eax = index
leal (%eax,%eax,4),%eax  # 5 * index
leal sea(,%eax,4),%eax    # sea + (20 * index)
```
Nested Array Row Access Code

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

int sea[4][5] =
{  { 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[2] (note the single index)?
- An array of 5 ints or row vector

What is the starting address for sea[2]?
Starting address is
sea + 5 * sizeof(element) * index = sea + 20 * 2

IA32 Code

- Computes and returns address
- Compute as sea + 4 * (index + 4 * index) = sea + 20 * index

# %eax = index
lea (%eax, %eax, 4), %eax    # 5 * index
lea sea(, %eax, 4), %eax    # sea + (20 * index)
int A[R][C];

What is the starting address for A[i][j]?
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];
```

What is the starting address for $A[i][j]$?

$A + i \times C \times 4 + j \times 4$
Nested Array Element Access Code

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

`int sea[4][5] =`  
`{{ 9, 8, 1, 9, 5 },`  
`{ 9, 8, 1, 0, 5 },`  
`{ 9, 8, 1, 0, 3 },`  
`{ 9, 8, 1, 1, 5 }};`  

```assembly
# %ecx = dig  
# %eax = index  
leal 0(%ecx,4),%edx  
leal (%eax,%eax,4),%eax  
movl sea(%edx,%eax,4),%eax  
```

- **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 Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sea[3][3]</code></td>
<td>$76+20\cdot3+4\cdot3 = 148$</td>
<td>1 Yes</td>
</tr>
<tr>
<td><code>sea[2][5]</code></td>
<td>$76+20\cdot2+4\cdot5 = 136$</td>
<td>9 Yes</td>
</tr>
<tr>
<td><code>sea[2][-1]</code></td>
<td>$76+20\cdot2+4\cdot(-1) = 112$</td>
<td>5 Yes</td>
</tr>
<tr>
<td><code>sea[4][-1]</code></td>
<td>$76+20\cdot4+4\cdot(-1) = 152$</td>
<td>5 Yes</td>
</tr>
<tr>
<td><code>sea[0][19]</code></td>
<td>$76+20\cdot0+4\cdot19 = 152$</td>
<td>5 Yes</td>
</tr>
<tr>
<td><code>sea[0][-1]</code></td>
<td>$76+20\cdot0+4\cdot(-1) = 72$</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
int cmu[5] = { 1, 5, 2, 1, 3 };  
int uw[5] = { 9, 8, 1, 9, 5 };  
int ucb[5] = { 9, 4, 7, 2, 0 };  

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

```
int univ2D[3] = {  
{ 9, 8, 1, 9, 5 },  
{ 1, 5, 2, 1, 3 },  
{ 9, 4, 7, 2, 0 }  
};
```

Is this declaration the same thing as a 2D array? No! One array declaration is guaranteed to be one contiguous block of memory. An array of arrays is not.
Multi-Level Array Example

- Variable `univ` denotes an array of 3 elements
- Each element is a pointer (4 bytes)
- Each pointer points to an array of ints

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

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

Note: this is how Java represents multi-dimensional arrays.
Element Access in Multi-Level Array

```
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]
```

- **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

- Reference Address Value Guaranteed?
  - `univ[2][3]` 60+4*3 = 72 2 Yes
  - `univ[1][5]` 16+4*5 = 36 9 No
  - `univ[2][-2]` 60+4*-2 = 52 5 No
  - `univ[3][-1]` #@@!^?? ?? No
  - `univ[1][12]` 16+4*12 = 64 4 No

- Code does not do any bounds checking
- Location of each lower-level array in memory is not guaranteed
#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;
}
Using Nested Arrays: arrays of arrays

- **Strengths**
  - Generates very efficient assembly 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: arrays of pointers to 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];
}
```

```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)]
```
Summary: Arrays in C

- Contiguous allocations of memory
- No bounds checking
- Can usually be treated like a pointer to first element
- `int a[4][5]` => array of arrays
  - all levels in one contiguous block of memory
- `int* b[4]` => array of pointers to arrays
  - first level in one contiguous block of memory
  - parts anywhere in memory
Announcements

• Lab 2 due tonight
  • If you end up stuck on a phase, submit an explanation of what you figured out in a separate file.

• Midterm Exam: Friday, May 1, in class.
  • HW/SW concepts, numerical representation, pointers, stacks/procedures, arrays/structs, C-asm-C mapping
  • We will provide a cheat sheet 😊!
    • Powers of 2, x86 assembly instructions
  • Some review (BYO questions) on Tuesday, April 28