The Hardware/Software Interface
CSE351 Spring 2015
Lecture 13

Instructor:
Katelin Bailey

Teaching Assistants:
Announcements

• **Assignments**
  • HW1 feedback should have been emailed to you (check your junk folder if you don’t see it)
  • HW2 is due Wednesday
  • Lab3 will be released on Wednesday and due the following Friday

• **Midterm is Friday in class**
  • If you need special accommodations for the exam, please talk to me ASAP.
  • Midterm review tomorrow 6-8pm in CSE 403. BYO questions: I’ll just be answering whatever you throw at me.
  • Sections this week are intended to help you with Lab3, **not** be midterm review, but the TAs may answer some questions for you.

• **Moving back toward C programming**
  • K&R readings will be posted on the website for various topics (e.g. pointers)
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:

OS:
Windows 8
Mac

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

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structs**
  - Alignment
- **Unions**
Nested Array Example (review)

```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`
Nested Array Row Access (review)

- **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 $A[i][j]$?

$A + i \times C \times 4 + j \times 4$
### Strange Referencing Examples (review)

```plaintext
zip_dig sea[4];

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>sea[3][3]</td>
<td>76+20<em>3+4</em>3 = 148</td>
<td>1 Yes</td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>76+20<em>2+4</em>5 = 136</td>
<td>9 Yes</td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>76+20<em>2+4</em>-1 = 112</td>
<td>5 Yes</td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>76+20<em>4+4</em>-1 = 152</td>
<td>5 Yes</td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>76+20<em>0+4</em>19 = 152</td>
<td>5 Yes</td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td>76+20<em>0+4</em>-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 integers.

```
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

```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:

```
```
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
Using Nested Arrays

#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
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
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;`
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**

```c
# %eax = val
# %edx = r
movl %eax,0(%edx)  # Mem[r+0] = val
```
Generating Pointer to Structure Member

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

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

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

```assembly
# %ecx = idx
# %edx = r
leal 4(%edx,%ecx,4),%eax # r+4*idx+4
```

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

```
r r+4+4*idx
```

```
i a p
0 4 16 20
```

```
int* find_address_of_elem (struct rec* r, int idx)
{
  return &r->a[idx];
}
```
Accessing to Structure Member

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

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

### Reading Array Element

- Offset of each structure member still determined at compile time

```assembly
# %ecx = idx
# %edx = r
movl 4(%edx,%ecx,4),%eax # Mem[r+4*idx+4]
```
How would it look if data items inside the struct were **aligned** (address is multiple of type size)?

... answer on next page
Structures & Alignment

• Unaligned Data

\[
\begin{align*}
\text{c} & \quad \text{v} \quad \text{i} \\
p & \quad p+1 & \quad p+9 & \quad p+13
\end{align*}
\]

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

• Aligned Data

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

\[
\begin{align*}
\text{c} & \quad \text{v} \quad \text{i} \\
p+0 & \quad p+8 & \quad p+16 & \quad p+20
\end{align*}
\]

\[
\text{struct S1} \ {\ }
\begin{align*}
\text{char c;} & \\
\text{double v;} & \\
\text{int i;} & \\
\text{)} & \text{* p;}
\end{align*}
\]

internal fragmentation
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, Windows, Mac OS X, ...
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, Windows, Mac OS X, ... 

- **Motivation for Aligning Data**
  - Physical memory is accessed by aligned chunks of 4 or 8 bytes
    - Inefficient to load or store datum that spans these 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 like 2 contiguous 4-byte primitive data items
Saving Space with Alignment

- Put large data types first:

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

  ```c
  struct S2 {
    double v;
    int i;
    char c;
  } * q;
  ```

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

  ```plaintext
  But actually...
  ```
Struct Alignment Principles

- Size must be a multiple of the largest primitive type inside.

\[ K = 8 \quad \text{so} \quad \text{size mod } 8 = 0 \]
Arrays of Structures

- Satisfy alignment requirement for every element

How would accessing an element work?

- declares a new type “struct S2”
- then declares an array a that contains 10 “struct S2” elements
- get to an element a[2], or a member of the struct a[2].i

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

- Allocated according to largest element
- Can only use one member at a time

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

```c
struct S {
    char c;
    int i[2];
    double v;
} *sp;
```
What Are Unions Good For?

- Unions allow the same region of memory to be referenced as different types
  - Different “views” of the same memory location
  - Can be used to circumvent C’s type system (bad idea)
- Better idea: use a struct inside a union to access some memory location either as a whole or by its parts
- But watch out for endianness at a small scale…
- Layout details are implementation/machine-specific…

```c
union int_or_bytes {
    int i;
    struct bytes {
        char b0, b1, b2, b3;
    }
}
```
Summary

- **Arrays in C**
  - Contiguous allocations of memory
  - No bounds checking
  - Can usually be treated like a pointer to first element

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

- **Unions**
  - Provide different views of the same memory location