Arrays
CSE 410 Winter 2017

Instructor: Justin Hsia

Teaching Assistants: Kathryn Chan, Kevin Bi, Ryan Wong, Waylon Huang, Xinyu Sui
Administrivia

- Homework 3 due Thursday (2/9)
- Lab 2 due next Monday (2/13)
- Lab 3 released next Monday (2/13)

- Midterm Friday (2/10) in lecture
  - Make a cheat sheet! – two-sided letter page, *handwritten*
  - Midterm details Piazza post: [66]
  - Expect average around 70% – comparable to 351 Au16 MT

- Midterm review session
  - 5-7pm on Tuesday (2/7) in BAG 261
Roadmap

C:

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

Java:

```java
    Car c = new Car();
c.setMiles(100);
c.setGals(17);
    float mpg = c.getMPG();
```

Assembly language:

```assembly
    get_mpg:
        pushq %rbp
        movq %rsp, %rbp
        ...
        popq %rbp
        ret
```

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
1100000111111101000011111
```

Computer system:

OS:

Windows 8

Mac

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Operating Systems
Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structs**
  - Alignment

- **Unions**

- Also: Some C details and how they relate to Java and assembly
  - C arrays are convenient but with some unique/strange rules
Array Allocation

- Basic Principle
  - `T A[N];` → array of data type `T` and length `N`
  - *Contiguously* allocated region of `N*sizeof(T)` bytes
  - Identifier `A` returns address of array (type `T*`)

```c
char msg[12];
int val[5];
double a[3];
char* p[3];
(or char *p[3];)
```
Array Access

- **Basic Principle**
  - \( T \ A[N] ; \) → array of data type \( T \) and length \( N \)
  - Identifier \( A \) returns address of array (type \( T^* \))

```
int x[5];
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[4]</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>x</td>
<td>int *</td>
<td>a</td>
</tr>
<tr>
<td>x+1</td>
<td>int *</td>
<td>a + 4</td>
</tr>
<tr>
<td>&amp;x[2]</td>
<td>int *</td>
<td>a + 8</td>
</tr>
<tr>
<td>x[5]</td>
<td>int</td>
<td>?? (whatever’s in memory at addr x+20)</td>
</tr>
<tr>
<td>*(x+1)</td>
<td>int</td>
<td>7</td>
</tr>
<tr>
<td>x+i</td>
<td>int *</td>
<td>a + 4*i</td>
</tr>
</tbody>
</table>
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 };

- typedef: Declaration “zip_dig uw” equivalent to “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 };
```

- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
Array Accessing Example

```c
typedef int zip_dig[5];

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

```
get_digit:
    movl (%rdi,%rsi,4), %eax  # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi+4*%rsi`, so use memory reference (`%rdi,%rsi,4`)
## Referencing Examples

```c
typedef int zip_dig[5];
```

```c
zip_dig cmu;
zip_dig uw;
zip_dig ucb;
```

<table>
<thead>
<tr>
<th>Reference</th>
<th>Address</th>
<th>Value</th>
<th>Guaranteed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>uw[3]</td>
<td>16</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>uw[6]</td>
<td>20</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>uw[-1]</td>
<td>24</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>cmu[15]</td>
<td>28</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Referencing Examples

```c
typedef int zip_dig[5];
```

```c
typedef int zip_dig[5];
```

```c
zip_dig cmu;
```

```c
zip_dig uw;
```

```c
zip_dig ucb;
```

<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* 3 = 48</td>
<td>9</td>
<td>Yes</td>
</tr>
<tr>
<td>uw[6]</td>
<td>36 + 4* 6 = 60</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td>uw[-1]</td>
<td>36 + 4*(-1) = 32</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>cmu[15]</td>
<td>16 + 4*15 = 76</td>
<td>??</td>
<td>No</td>
</tr>
</tbody>
</table>

- No bounds checking
- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general
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;
}
```

typedef int zip_dig[5];

\[
\begin{align*}
zi &= 10 \times 0 + 9 = 9 \\
zi &= 10 \times 9 + 8 = 98 \\
zi &= 10 \times 98 + 1 = 981 \\
zi &= 10 \times 981 + 9 = 9819 \\
zi &= 10 \times 9819 + 5 = 98195
\end{align*}
\]
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)
Array Loop Implementation

- **Registers:**
  - `%rdi` `z`
  - `%rax` `zi`
  - `%rcx` `zend`

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

```c
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 5;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z < zend);
    return zi;
}
```

```assembly
# %rdi = z
leaq 20(%rdi),%rcx # %rcx = zend = z+5
movl $0,%eax # %rax = zi = 0
.L17:
leal (%rax,%rax,4),%edx # zi + 4*zi = 5*zi
movl (%rdi),%eax # %eax = *z
leal (%rax,%rdx,2),%eax # zi = *z + 2*(5*zi)
addq $4,%rdi # z++
cmpq %rdi,%rcx # zend : z
jne .L17 # if != goto loop
```
C Details: Arrays and Pointers

- Arrays are (almost) identical to pointers
  - `char *string and char string[]` are nearly identical declarations
  - Differ in subtle ways: initialization, `sizeof()`, etc.

- An array variable looks like a pointer to the first (0th) element
  - `ar[0] same as *ar; ar[2] same as *(ar+2)`

- An array variable is read-only (no assignment)
  - Cannot use “`ar = <anything>`”
C Details: Arrays and Functions

- Declared arrays only allocated while the scope is valid:

  ```c
  char *foo() {
    char string[32]; ...; return string;
  }
  ```

- An array is passed to a function as a pointer:

  ```c
  int foo(int ar[], unsigned int size) {
    ... ar[size-1] ...
  }
  ```

   **BAD!**

   Really `int *ar`

   Must explicitly pass the size!
C Details: Arrays and Functions

- Array size gets lost when passed to a function
- What prints in the following code:

```c
int foo(int array[], unsigned int size) {
    ...
    printf("%d\n", sizeof(array));
}

int main(void) {
    int a[10], b[5];
    ... foo(a, 10) ...
    printf("%d\n", sizeof(a));
}
```

- `sizeof(int *)`  
- `10*sizeof(int)`
Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level

- **Structs**
  - Alignment

- **Unions**

- Also: Some C details and how they relate to Java and assembly
  - C arrays are convenient but with some unique/strange rules
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 });
```

same as:

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

What is the layout in memory?

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

```c
typedef int zip_dig[5];
```
Nested Array Example

```c
typedef int zip_dig[5];

zip_dig sea[4] =
    {{ 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
- Elements in the same row are contiguous
- Guaranteed (in C)
Two-Dimensional (Nested) Arrays

- Declaration: \( T \ A[R][C] ; \)
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Each element requires \( \text{sizeof}(T) \) bytes

- Array size?
Two-Dimensional (Nested) Arrays

- Declaration: \( T \ A[R][C] \);
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Each element requires \( \text{sizeof}(T) \) bytes

- Array size:
  - \( R \times C \times \text{sizeof}(T) \) bytes

- Arrangement: **row-major** ordering

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

![Array arrangement diagram]

- \( A[0][0] \) \( \ldots \) \( A[0][C-1] \)
- \( \vdots \)
- \( \vdots \)
- \( A[R-1][0] \) \( \ldots \) \( A[R-1][C-1] \)

- Total: \( 4 \times R \times C \) bytes
Nested Array Row Access

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

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

![Diagram of nested array row access](image)

\[ A \]
\[ A[0] \]
\[ A[0][0] \]
\[ A[i] \]
\[ A[i][0] \]
\[ A[i][C-1] \]
\[ A[R-1] \]
\[ A[R-1][0] \]
\[ A[R-1][C-1] \]
\[ A+i*C*4 \]
\[ A+(R-1)*C*4 \]
Nested Array **Row Access Code**

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

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

```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 } }
```

```
get_sea_zip(int):
    movslq %edi, %rdi
    leaq (%rdi, %rdi, 4), %rdx
    leaq 0(,%rdx, 4), %rax
    addq $sea, %rax
    ret

sea:
    .long 9
    .long 8
    .long 1
    .long 9
    .long 5
    .long 9
    .long 8
    ...
```
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[index]`?
- What is its starting address?

```asm
# %rdi = index
leaq (%rdi,%rdi,4),%rax  
leaq sea(,%rax,4),%rax
```

**Translation?**
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 }};
```

```assembly
# %rdi = index
leaq (%rdi,%rdi,4),%rax  # 5 * index
leaq sea(,%rax,4),%rax  # sea + (20 * index)
```

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

- **Assembly Code**
  - Computes and returns address
  - Compute as: sea+4*(index+4*index) = sea+20*index
int A[R][C];

\[ A[0][0] \ldots A[0][C-1] \]

\[ A[i][j] \]

\[ A[R-1][0] \ldots A[R-1][C-1] \]

A + i*C*4

A + (R-1)*C*4

Nested Array Element Access
Nested Array Element Access

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

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

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

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

- **Array Elements**
  - `sea[index][digit]` is an int (sizeof(int)=4)
  - Address = `sea + 5*4*index + 4*digit`

- **Assembly Code**
  - Computes address as: `sea + ((index+4*index) + digit)*4`
  - `movl` performs memory reference

```assembly
leaq (%rdi,%rdi,4), %rax  # 5*index
addl %rax, %rsi           # 5*index+digit
movl sea(,%rsi,4), %eax  # *(sea + 4*(5*index+digit))
```
Strange Referencing Examples

```c
typedef int zip_dig[5];
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>81</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>sea[2][5]</td>
<td>10</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sea[2][-1]</td>
<td>-1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>sea[4][-1]</td>
<td>-1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>sea[0][19]</td>
<td>19</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>sea[0][-1]</td>
<td>-1</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
Strange Referencing Examples

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

### Reference | Address | Value | Guaranteed?
--- | --- | --- | ---
sea[3][3] | 76+20*3+4*3 = 148 | 1 | Yes
sea[2][5] | 76+20*2+4*5 = 136 | 9 | Yes
sea[2][-1] | 76+20*2+4*-1 = 112 | 5 | Yes
sea[4][-1] | 76+20*4+4*-1 = 152 | 5 | Yes
sea[0][19] | 76+20*0+4*19 = 152 | 5 | Yes
sea[0][-1] | 76+20*0+4*-1 = 72 | ?? | No

- Code does not do any bounds checking
- Ordering of elements within array guaranteed
Data Structures in Assembly

❖ Arrays
  ▪ One-dimensional
  ▪ Multi-dimensional (nested)
  ▪ Multi-level

❖ Structs
  ▪ Alignment

❖ Unions

❖ Also: Some C details and how they relate to Java and assembly
  ▪ C arrays are convenient but with some unique/strange rules
Multi-Level Array Example

Multi-Level Array Declaration(s):

```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};
```

Is a multi-level array the same thing as a 2D array? **NO**

2D Array Declaration:

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

One array declaration = one contiguous block of memory
Multi-Level Array Example

- **Variable `univ` denotes array of 3 elements**
- Each element is a pointer
  - 8 bytes each
- Each pointer points to array of `ints`

```java
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 digit) {
    return univ[index][digit];
}
```

- **Computation**
  - Element access: `Mem[Mem[univ+8*index]+4*digit]`
  - Must do **two memory reads**
    - First get pointer to row array
    - Then access element within array
  - But allows inner arrays to be different lengths (not in this example)
Array Element Accesses

Nested array

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

Multi-level array

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

Access looks the same, but it isn’t:

\[
\text{Mem[sea+20*index+4*digit]} \quad \text{Mem[Mem[univ+8*index]+4*digit]}
\]
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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>univ[1][5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>univ[2][-2]</td>
<td></td>
<td></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>

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

C Code does not do any bounds checking

Location of each lower-level array in memory is not guaranteed

<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>60+4*3 = 72</td>
<td>2</td>
<td>Yes</td>
</tr>
<tr>
<td>univ[1][5]</td>
<td>16+4*5 = 36</td>
<td>9</td>
<td>No</td>
</tr>
<tr>
<td>univ[2][-2]</td>
<td>60+4*-2 = 52</td>
<td>5</td>
<td>No</td>
</tr>
<tr>
<td>univ[3][-1]</td>
<td>#@$%^??</td>
<td>??</td>
<td>No</td>
</tr>
<tr>
<td>univ[1][12]</td>
<td>16+4*12 = 64</td>
<td>4</td>
<td>No</td>
</tr>
</tbody>
</table>
Summary

- Contiguous allocations of memory
- No bounds checking (and no default initialization)
- Can usually be treated like a pointer to first element

```c
int a[4][5];  // array of arrays
  ▪ all levels in one contiguous block of memory
```

```c
int* b[4];  // array of pointers to arrays
  ▪ First level in one contiguous block of memory
  ▪ Each element in the first level points to another “sub” array
  ▪ Parts anywhere in memory
```