Machine-Level Programming IV: Data

15-213: Introduction to Computer Systems
8th Lecture, Feb. 4, 2015

Instructors:
Franz Franchetti & Seth Copen Goldstein, Ralf Brown, and Brian Railing
Today

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

- Structures
  - Allocation
  - Access
  - Alignment

- Floating Point
Array Allocation

Basic Principle

Array of data type $T$ and length $L$
- Contiguously allocated region of $L \times \text{sizeof}(T)$ bytes in memory

```c
char string[12];

int val[5];

double a[3];

char *p[3];
```

![Diagram showing memory allocation for different types of arrays]
Array Access

**Basic Principle**

\[ T \ A[L]; \]

- Array of data type \( T \) and length \( L \)
- Identifier \( A \) can be used as a pointer to array element 0: Type \( T^* \)

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

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int *</td>
<td>( x )</td>
</tr>
<tr>
<td>val+1</td>
<td>int *</td>
<td>( x + 4 )</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int *</td>
<td>( x + 8 )</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>val + ( i )</td>
<td>int *</td>
<td>( x + 4 ) * ( i )</td>
</tr>
</tbody>
</table>
Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

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

zip_digit cmu;

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</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>

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

x86-64

```c
# %rdi = z
# %rsi = 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
- Use memory reference (%rdi,%rsi,4)
Array Loop Example

```c
void zincr(zip_dig z) {
    size_t i;
    for (i = 0; i < ZLEN; i++)
        z[i]++;
}
```

---

```
# %rdi = z
movl  $0, %eax  #   i = 0
jmp .L3        #   goto middle
.L4:
    addl  $1, (%rdi,%rax,4) #   z[i]++
    addq  $1, %rax          #   i++
.L3:
    cmpq  $4, %rax          #   i:4
    jbe   .L4               #   if <=, goto loop
rep; ret
```
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**
  - \( R \times C \times K \) bytes

- **Arrangement**
  - Row-Major Ordering

\[
\begin{array}{c}
A[0][0] \quad \cdots \quad A[0][C-1] \\
\vdots \\
A[R-1][0] \quad \cdots \quad A[R-1][C-1]
\end{array}
\]

\[
\begin{array}{c}
\text{int } A[R][C];
\end{array}
\]
 Nested Array Example

```c
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
{ {1, 5, 2, 0, 6},
  {1, 5, 2, 1, 3 },
  {1, 5, 2, 1, 7 },
  {1, 5, 2, 2, 1 } }; 
```

- Variable pgh: array of 4 elements, allocated contiguously
- Each element is an array of 5 int's, allocated contiguously

“Row-Major” ordering of all elements in memory
Nested Array Row Access

- **Row Vectors**
  - \( A[i] \) 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 of nested array row access](attachment:image.png)
**Nested Array Row Access Code**

- **Row Vector**
  - `pgh[index]` is array of 5 int’s
  - Starting address `pgh+20*index`

- **Machine Code**
  - Computes and returns address
  - Compute as `pgh + 4*(index+4*index)`

```c
int *get_pgh_zip(int index)
{
    return pgh[index];
}
```

```assembly
# %rdi = index
    leaq (%rdi,%rdi,4),%rax # 5 * index
    leaq pgh(,%rax,4),%rax # pgh + (20 * index)
```
Nested Array Element Access

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

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

```
[0]    [i]    [R-1]
[0]    [j]    [0]
[C-1]  [i]    [C-1]
```

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

**Array Elements**

- \( \text{pgh}[\text{index}][\text{dig}] \) is int
- Address: \( \text{pgh} + 20*\text{index} + 4*\text{dig} \)
  
  \[ \text{pgh} + 4*(5*\text{index} + \text{dig}) \]

```c
int get_pgh_digit(int index, int dig)
{
    return pgh[index][dig];
}
```

```assembly
leaq (%rdi,%rdi,4), %rax  # 5*index
addl %rax, %rsi          # 5*index+dig
movl pgh(%rsi,4), %eax   # M[pgh + 4*(5*index+dig)]
```
Multi-Level Array Example

Variable `univ` denotes array of 3 elements

- Each element is a pointer
  - 8 bytes
- Each pointer points to array of int’s

```c
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```
Element Access in Multi-Level Array

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

```
salq    $2, %rsi            # 4*digit
addq    univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
movl    (%rsi), %eax        # return *p
ret
```

- **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
Array Element Accesses

Nested array

```c
int get_pgh_digit(size_t index, size_t digit)
{
    return pgh[index][digit];
}
```

Multi-level array

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

Accesses looks similar in C, but address computations very different:

\[
\text{Mem}[\text{pgh} + 20 \times \text{index} + 4 \times \text{digit}] \quad \text{Mem}[\text{Mem}[\text{univ} + 8 \times \text{index}] + 4 \times \text{digit}]\
\]
**N X N Matrix**

**Code**

- **Fixed dimensions**
  - Know value of $N$ at compile time

- **Variable dimensions, explicit indexing**
  - Traditional way to implement dynamic arrays

- **Variable dimensions, implicit indexing**
  - Now supported by gcc

```c
#define N 16
typedef int fix_matrix[N][N];
/* Get element A[i][j] */
int fix_ele(fix_matrix A,
            size_t i, size_t j)
{
    return A[i][j];
}
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element A[i][j] */
int vec_ele(size_t n, int *A,
            size_t i, size_t j)
{
    return A[IDX(n,i,j)];
}
/* Get element a[i][j] */
int var_ele(size_t n, int A[n][n],
            size_t i, size_t j) {
    return A[i][j];
}
```
# 16 X 16 Matrix Access

## Array Elements

- `int A[16][16];`
- Address `A + i * (C * K) + j * K`
- `C = 16, K = 4`

```c
/* Get element A[i][j] */
int fix_ele(fix_matrix A, size_t i, size_t j) {
    return A[i][j];
}
```

```assembly
# A in %rdi, i in %rsi, j in %rdx
salq $6, %rsi             # 64*i
addq %rsi, %rdi           # A + 64*i
movl (%rdi,%rdx,4), %eax # M[A + 64*i + 4*j]
ret
```
### $n \times n$ Matrix Access

#### Array Elements

- `size_t n;`
- `int A[n][n];`
- Address $A + i \times (n \times K) + j \times K$
- $C = n$, $K = 4$
- Must perform integer multiplication

```c
/* Get element $A[i][j]$ */
int var_ele(size_t n, int A[n][n], size_t i, size_t j)
{
    return A[i][j];
}
```

```assembly
# n in %rdi, A in %rsi, i in %rdx, j in %rcx
imulq %rdx, %rdi # n*i
leaq (%rsi,%rdi,4), %rax # A + 4*n*i
movl (%rax,%rcx,4), %eax # A + 4*n*i + 4*j
ret
```
Example: Array Access

```c
#include <stdio.h>

#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

void main(void) {
    zip_dig pgh[PCOUNT] =
        {{1, 5, 2, 0, 6},
         {1, 5, 2, 1, 3},
         {1, 5, 2, 1, 7},
         {1, 5, 2, 2, 1}};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
        pgh[0][0] +
        linear_zip[7] +
        *(linear_zip + 8) +
        zip2[1];
    printf("result: %d\n", result);
}
```

```bash
linux> ./array
result: 9
```
Example: Array Access

```c
#include <stdio.h>

#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

void main(void) {
    zip_dig pgh[PCOUNT] =
    {{1, 5, 2, 0, 6},
     {1, 5, 2, 1, 3 },
     {1, 5, 2, 1, 7 },
     {1, 5, 2, 2, 1 }};
    int *linear_zip = (int *) pgh;
    int *zip2 = (int *) pgh[2];
    int result =
    pgh[0][0] +
    linear_zip[7] +
    *(linear_zip + 8) +
    zip2[1];
    printf("result: %d\n", result);
}
```

```
linux> ./array
result: 9
```
Today

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

- Structures
  - Allocation
  - Access
  - Alignment

- Floating Point
Structure Representation

Structure represented as block of memory
- Big enough to hold all of the fields

Fields ordered according to declaration
- Even if another ordering could yield a more compact representation

Compiler determines overall size + positions of fields
- Machine-level program has no understanding of the structures in the source code

```c
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```
Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```

### Generating Pointer to Array Element
- Offset of each structure member determined at compile time
- Compute as `r + 4*idx`

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

```assembly
# r in %rdi, idx in %rsi
leaq (%rdi,%rsi,4), %rax
ret
```
Following Linked List

- C Code

```c
void set_val (struct rec *r, int val) {
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->next;
    }
}
```

### Structure of `struct rec`

- `int a[4]`
- `int i`
- `struct rec *next`

### Element `i`

- `a`:
  - Register: `%rdi`
  - Value: `r`
- `i`:
  - Register: `%rsi`
  - Value: `val`

### Code Snippet

```
.L11:  # loop:
    movslq  16(%rdi), %rax  #   i = M[r+16]
    movl    %esi, (%rdi,%rax,4) #   M[r+4*i] = val
    movq    24(%rdi), %rdi  #   r = M[r+24]
    testq   %rdi, %rdi      #   Test r
    jne     .L11            #   if !=0 goto loop
```
Structures & Alignment

- **Unaligned Data**
  - 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 $K$

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

- **Aligned Data**
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
  - Required on some machines; advised on x86-64

- **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 trickier when datum spans 2 pages

- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields
Specific Cases of Alignment (x86-64)

- **1 byte: char, ...**
  - no restrictions on address

- **2 bytes: short, ...**
  - lowest 1 bit of address must be 0₂

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

- **8 bytes: double, long, char *, ...**
  - lowest 3 bits of address must be 000₂
Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each 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:**
  - $K = 8$, due to `double` element

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Meeting Overall Alignment Requirement

- For largest alignment requirement $K$
- Overall structure must be multiple of $K$

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

<table>
<thead>
<tr>
<th></th>
<th>v</th>
<th>i[0]</th>
<th>i[1]</th>
<th>c</th>
<th>7 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p+8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p+16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Multiple of $K=8$
Arrays of Structures

- Overall structure length multiple of K
- 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*idx
  - `sizeof(S3)`, including alignment spacers
- Element j is at offset 8 within 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;
}
```

```assembly
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```
Saving Space

- Put large data types first

```
struct S4 {  
    char c;  
    int i;  
    char d;  
} *p;
```

- Effect (K=4)

```
struct S5 {  
    int i;  
    char c;  
    char d;  
} *p;
```

```c
<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>c</th>
<th>d</th>
<th>2 bytes</th>
</tr>
</thead>
</table>
Example Struct Exam Question

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

```c
typedef struct {
    char a;
    long b;
    float c;
    char d[3];
    int *e;
    short *f;
} foo;
```

1. Show how `foo` would be allocated in memory on an x86-64 Linux system. Label the bytes with the names of the various fields and clearly mark the end of the struct. Use an X to denote space that is allocated in the struct as padding.

```
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
|    a    |    X    |    X    |    X    |    X    |    b    |    b    |    b    |    b    |    b    |    b    |    b    |    b    |    b    |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
|    c    |    c    |    c    |    d    |    d    |    d    |    X    |    e    |    e    |    e    |    e    |    e    |    e    |    e    |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
|    f    |    f    |    f    |    f    |    f    |    f    |    f    |    f    |    f    |    f    |    f    |    f    |    f    |    f    |
+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+---------+
```

Example Struct Exam Question (Cont’d)

Problem 5. (8 points):

*Struct alignment.* Consider the following C struct declaration:

```c
typedef struct {
    char a;
    long b;
    float c;
    char d[3];
    int *e;
    short *f;
} foo;
```

2. Rearrange the elements of `foo` to conserve the most space in memory. Label the bytes with the names of the various fields and clearly *mark the end of the struct*. Use an X to denote space that is allocated in the struct as padding.

Today

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

- **Structures**
  - Allocation
  - Access
  - Alignment

- **Floating Point**
Background

- History
  - x87 FP
    - Legacy, very ugly
  - SSE FP
    - Supported by Shark machines
    - Special case use of vector instructions
  - AVX FP
    - Newest version
    - Similar to SSE
    - Documented in book
### Programming with SSE3

#### XMM Registers

- **16 total, each 16 bytes**
- **16 single-byte integers**
- **8 16-bit integers**
- **4 32-bit integers**
- **4 single-precision floats**
- **2 double-precision floats**
- **1 single-precision float**
- **1 double-precision float**
Scalar & SIMD Operations

- **Scalar Operations: Single Precision**
  - `addss %xmm0, %xmm1`
  - `%xmm0`
  - `%xmm1`

- **SIMD Operations: Single Precision**
  - `addps %xmm0, %xmm1`
  - `%xmm0`
  - `%xmm1`

- **Scalar Operations: Double Precision**
  - `addsd %xmm0, %xmm1`
  - `%xmm0`
  - `%xmm1`
FP Basics

- Arguments passed in %xmm0, %xmm1, ...
- Result returned in %xmm0
- All XMM registers caller-saved

```c
float fadd(float x, float y)
{
    return x + y;
}

double dadd(double x, double y)
{
    return x + y;
}
```

```c
# x in %xmm0, y in %xmm1
addss  %xmm1, %xmm0
ret
```

```c
# x in %xmm0, y in %xmm1
addsd %xmm1, %xmm0
ret
```
FP Memory Referencing

- Integer (and pointer) arguments passed in regular registers
- FP values passed in XMM registers
- Different mov instructions to move between XMM registers, and between memory and XMM registers

```cpp
double dincr(double *p, double v) {
    double x = *p;
    *p = x + v;
    return x;
}
```

```
# p in %rdi, v in %xmm0
movapd %xmm0, %xmm1    # Copy v
movsd (%rdi), %xmm0    # x = *p
addsd %xmm0, %xmm1    # t = x + v
movsd %xmm1, (%rdi)    # *p = t
ret
```
Other Aspects of FP Code

- **Lots of instructions**
  - Different operations, different formats, ...

- **Floating-point comparisons**
  - Instructions `ucomiss` and `ucomisd`
  - Set condition codes CF, ZF, and PF

- **Using constant values**
  - Set XMM0 register to 0 with instruction `xorpd %xmm0, %xmm0`
  - Others loaded from memory
Summary

- **Arrays**
  - Elements packed into contiguous region of memory
  - Use index arithmetic to locate individual elements

- **Structures**
  - Elements packed into single region of memory
  - Access using offsets determined by compiler
  - Possible require internal and external padding to ensure alignment

- **Combinations**
  - Can nest structure and array code arbitrarily

- **Floating Point**
  - Data held and operated on in XMM registers
Understanding Pointers & Arrays #1

<table>
<thead>
<tr>
<th>Decl</th>
<th>An</th>
<th>*An</th>
<th>Cmp</th>
<th>Bad</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>int A1[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int *A2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Cmp**: Compiles (Y/N)
- **Bad**: Possible bad pointer reference (Y/N)
- **Size**: Value returned by `sizeof`
# Understanding Pointers & Arrays #1

<table>
<thead>
<tr>
<th>Decl</th>
<th>An</th>
<th>*An</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cmp</td>
<td>Bad</td>
</tr>
<tr>
<td>int *A2</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

- **Cmp:** Compiles (Y/N)
- **Bad:** Possible bad pointer reference (Y/N)
- **Size:** Value returned by `sizeof`
# Understanding Pointers & Arrays #2

<table>
<thead>
<tr>
<th>Decl</th>
<th>An</th>
<th>*An</th>
<th>**An</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cmp</td>
<td>Bad</td>
<td>Size</td>
</tr>
<tr>
<td>int A1[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int *A2[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (*A3)[3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (*A4[3])</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Cmp**: Compiles (Y/N)
- **Bad**: Possible bad pointer reference (Y/N)
- **Size**: Value returned by `sizeof`
# Understanding Pointers & Arrays #2

<table>
<thead>
<tr>
<th>Decl</th>
<th>An</th>
<th>*An</th>
<th>**An</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cmp</td>
<td>Bad</td>
<td>Size</td>
</tr>
<tr>
<td>int A1[3]</td>
<td>Y</td>
<td>N</td>
<td>12</td>
</tr>
<tr>
<td>int *A2[3]</td>
<td>Y</td>
<td>N</td>
<td>24</td>
</tr>
<tr>
<td>int (*A3)[3]</td>
<td>Y</td>
<td>N</td>
<td>8</td>
</tr>
<tr>
<td>int (*A4[3])</td>
<td>Y</td>
<td>N</td>
<td>24</td>
</tr>
</tbody>
</table>

### Diagram

- **A1**
  - Allocated int
  - Unallocated pointer

- **A2/A4**
  - Allocated pointer
  - Unallocated pointer
  - Allocated int
  - Unallocated int

- **A3**
  - Allocated int
## Understanding Pointers & Arrays #3

<table>
<thead>
<tr>
<th>Decl</th>
<th>( An )</th>
<th>( *An )</th>
<th>( **An )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cm p</td>
<td>Bad</td>
<td>Size</td>
</tr>
<tr>
<td></td>
<td>Cm p</td>
<td>Bad</td>
<td>Size</td>
</tr>
<tr>
<td></td>
<td>Cm p</td>
<td>Bad</td>
<td>Size</td>
</tr>
</tbody>
</table>

- **Cmp:** Compiles (Y/N)
- **Bad:** Possible bad pointer reference (Y/N)
- **Size:** Value returned by `sizeof`

### Examples:

<table>
<thead>
<tr>
<th>Decl</th>
<th>( ***An )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cm p</td>
</tr>
<tr>
<td>int A1[3][5]</td>
<td></td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td></td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td></td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td></td>
</tr>
<tr>
<td>int (*A5[3])[5]</td>
<td></td>
</tr>
</tbody>
</table>
Carnegie Mellon

### Understanding Pointers & Arrays #3

<table>
<thead>
<tr>
<th>Decl</th>
<th>An</th>
<th>*An</th>
<th>**An</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cmp</td>
<td>Bad</td>
<td>Size</td>
</tr>
<tr>
<td>int A1[3][5]</td>
<td>Y</td>
<td>N</td>
<td>60</td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td>Y</td>
<td>N</td>
<td>120</td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td>Y</td>
<td>N</td>
<td>8</td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td>Y</td>
<td>N</td>
<td>120</td>
</tr>
<tr>
<td>int (*A5[3])[5]</td>
<td>Y</td>
<td>N</td>
<td>24</td>
</tr>
</tbody>
</table>

- **Cmp**: Compiles (Y/N)
- **Bad**: Possible bad pointer reference (Y/N)
- **Size**: Value returned by `sizeof`