Machine-Level Programming IV: Data

CS351: Systems Programming
Day 9: Sep. 20, 2022

Instructor:
Nik Sultana

Slides adapted from Bryant and O’Hallaron
Next time: recorded lecture

- LEC 9 and LEC 10 will be pre-recorded and circulated on Blackboard.
  - Do not come to SB104 those days – there will not be an in-person lecture.
  - My away-at-a-conference days are marked on the course calendar.
Started to read the CS:APP3e book?

- Do start reading the book – if you delay this it’ll be harder to catch up!
- Read a little bit every day. It’ll add up over time.

(from last week’s survey)

Yes
20 (69%)
K&R book ("The C Programming Language")

- Many respondees aren’t reading it.
- Many respondees don’t have access.
- I contacted the library and they confirmed they have the book, and they’ve enabled requests for scanning: https://i-share-iit.primo.exlibrisgroup.com/permalink/01CARLI_IIT/1ukd1ei/alma991416826105842

- Huge thanks to respondees of the survey, and to the IIT Paul V. Galvin Library.
Questions follow-up

Why was RollerCoaster Tycoon written in assembly? Why not use C?

- Previously (esp up to mid-2000s) not unusual to write (at least parts of) games in assembly when hardware was less resourced – i.e., small or no caches, and low clock rate.
  - C compilers weren’t always great at optimizing.
  - e.g., story of PS1 game hand-optimized to use cache AMAP.
- Portability not a concern? (Marketing choice?)
- Personal (or team!) choice when writing code?
- These days? Often C++, C#, Java
Today

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

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

- **Floating Point**
Array Allocation

**Basic Principle**

\[ T \ A[L]; \]

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

```markdown
char string[12];
```

```
\text{Contiguously allocated region: } x \text{ to } x + 12
```

```markdown
int val[5];
```

```
\text{Contiguously allocated region: } x \text{ to } x + 20
```

```markdown
double a[3];
```

```
\text{Contiguously allocated region: } x \text{ to } x + 24
```

```markdown
char *p[3];
```

```
\text{Contiguously allocated region: } x \text{ to } x + 24
```
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 $T^*$

  ![Array Access Diagram]

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

- **Reference**

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>6</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>0</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>$x + 4i$</td>
</tr>
</tbody>
</table>
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig iit = { 6, 0, 6, 1, 6 };
zip_dig uic = { 6, 0, 6, 0, 7 };
zip_dig nwu = { 6, 0, 2, 0, 8 };

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

zip_dig iit;  

\[
\begin{array}{ccccccc}
\text{16} & \text{20} & \text{24} & \text{28} & \text{32} & \text{36} \\
6 & 0 & 6 & 1 & 6 \\
\end{array}
\]

```c
int get_digit(int zip_dig z, int digit)
{
    return 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)

# %rdi = z
# %rsi = digit
```c
movl (%rdi,%rsi,4), %eax # z[digit]
```
Array Loop Example

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

```assembly
# %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

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

```
| A[0][0] | · · · | A[0][C-1] |
| A[1][0] | · · · | A[1][C-1] |
| ...     |      | ...       |
| A[R-1][0]| · · · | A[R-1][C-1]|
```

4\( R \times C \) Bytes
Nested Array Example

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

- “zip_dig chi[4]” equivalent to “int chi[4][5]”
  - Variable chi: 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](image-url)
### Nested Array Row Access Code

#### Row Vector
- \texttt{chi[index]} is array of 5 int's
- Starting address \texttt{chi+20*index}

#### Machine Code
- Computes and returns address
- Compute as \texttt{chi + 4*(index+4*index)}

```c
int *get_chizi(int index) {
    return chi[index];
}
```
Nested Array Element Access

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

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

![Diagram of nested array access](image)
Nested Array Element Access Code

Array Elements

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

```c
int get_chi_digit (int index, int dig) {
    return chi[index][dig];
}
```

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

- Variable `univ` denotes an array of 3 elements.
- Each element is a pointer:
  - 8 bytes
- Each pointer points to an array of `int`'s.

```c
#define UCOUNT 3
int *univ[UCOUNT] = {iit, uic, nwu};
```

```c
zipDig iit = { 6, 0, 6, 1, 6 }
zipDig uic = { 6, 0, 6, 0, 7 }
zipDig nwu = { 6, 0, 2, 0, 8 }
```
Element Access in Multi-Level Array

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

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

**Computation**

- Element access $\text{Mem[Mem[univ+8*index]+4*digit]}$
- Must do two memory reads
  - First get pointer to row array
  - Then access element within array
Array Element Accesses

Nested array

```c
int get_chi_digit(size_t index, size_t digit)
{
    return chi[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:

```
Mem[chi+20*index+4*digit]  Mem[Mem[univ+8*index]+4*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**
  - Address $A + i \times (C \times K) + j \times K$
  - $C = 16, K = 4$

```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 X n Matrix Access

- **Array Elements**
  - Address $A + i \times (C \times K) + j \times K$
  - $C = n, K = 4$
  - 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
```
Today

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

- Structures
  - Allocation
  - Access
  - Alignment

- Floating Point
Structure Representation

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

- **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
Generating Pointer to Structure Member

```c
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;
  }
}
```

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

---

### Register Value

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>r</td>
</tr>
<tr>
<td>%rsi</td>
<td>val</td>
</tr>
</tbody>
</table>

---

```
.L11:
  movslq 16(%rdi), %rax
  movl %esi, (%rdi,%rax,4)
  movq 24(%rdi), %rdi
  testq %rdi, %rdi
  jne .L11
```

---

# loop:

```
#   i = M[r+16]
#   M[r+4*i] = val
#   r = M[r+24]
#   Test r
#   if !=0 goto loop
```
Structures & Alignment

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

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

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

- **16 bytes: long double (GCC on Linux)**
  - lowest 4 bits of address must be 0000₂
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

```c
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;
```

Multiple of $K=8$
Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
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
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

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

- Effect (K=4)

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>3 bytes</td>
<td>i</td>
<td>d</td>
</tr>
<tr>
<td>i</td>
<td></td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>
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 modern Intel x86 machines
  - Special case use of vector instructions
- AVX FP
  - Newest version
  - Similar to SSE
  - Documented in textbook
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**
  - \texttt{addss} \%xmm0, \%xmm1
  - %xmm0
  - %xmm1

- **SIMD Operations: Single Precision**
  - \texttt{addps} %xmm0, %xmm1
  - %xmm0
  - %xmm1

- **Scalar Operations: Double Precision**
  - \texttt{addsd} %xmm0, %xmm1
  - %xmm0
  - %xmm1
FP Basics

- Arguments passed in \( \%\text{xmm0}, \%\text{xmm1}, \ldots \)
- Result returned in \( \%\text{xmm0} \)
- All XMM registers caller-saved

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

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

```assembly
# x in \%\text{xmm0}, y in \%\text{xmm1}
addss \%\text{xmm1}, \%\text{xmm0}
ret

# x in \%\text{xmm0}, y in \%\text{xmm1}
addsd \%\text{xmm1}, \%\text{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

```
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>
<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>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int *A2</td>
<td></td>
<td></td>
<td></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>
<th>Cmp</th>
<th>Bad</th>
<th>Size</th>
<th>Cmp</th>
<th>Bad</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>int *A2</td>
<td>Y</td>
<td>N</td>
<td>8</td>
<td>Y</td>
<td>Y</td>
<td>4</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></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>

#### Diagrams

- **A1**
- **A2/A4**
- **A3**

**Legend**
- Allocated pointer
- Unallocated pointer
- Allocated int
- Unallocated int

---

Illinois Tech CS351 Fall 2022
### 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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (*A3)[3][5]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int (*A5[3])[5]</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`

<table>
<thead>
<tr>
<th>Decl</th>
<th>***An</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cmp</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>
<tr>
<td>Declaration</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td></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>

---

**Diagram:**

- **Allocated pointer**
- **Allocated pointer to unallocated int**
- **Unallocated pointer**
- **Allocated int**
- **Unallocated int**

**Memory Layout:**

- **A1**
- **A2/A4**
- **A3**
- **A5**
# 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`

<table>
<thead>
<tr>
<th>Decl</th>
<th>***An</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cmp</td>
</tr>
<tr>
<td>int *A2[3][5]</td>
<td>Y</td>
</tr>
<tr>
<td>int (**A3)[3][5]</td>
<td>Y</td>
</tr>
<tr>
<td>int *(A4[3][5])</td>
<td>Y</td>
</tr>
<tr>
<td>int (**A5[3]) [5]</td>
<td>Y</td>
</tr>
</tbody>
</table>
LEC 9 and LEC 10 will be pre-recorded and circulated on Blackboard.

- Do not come to SB104 those days – there will not be an in-person lecture.
- My away-at-a-conference days are marked on the course calendar.
Per-lecture feedback

- Better sooner rather than later!
- I can help with issues sooner.
- There is a per-lecture feedback form.
- **The form is anonymous.**
  (It checks that you’re at Illinois Tech to filter abuse, but I don’t see who submitted any of the forms.)
- [https://forms.gle/qoeEbBuTYXo5FiU1A](https://forms.gle/qoeEbBuTYXo5FiU1A)
- I’ll remind about this at each lecture.