Machine-Level Programming IV: Data

15-213/18-213/14-513/15-513/18-613: Introduction to Computer Systems
8th Lecture, September 19, 2019
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

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

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

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

```
char *p[3];
```
Array Access

- **Basic Principle**

  \[ T \text{ A}[L] ; \]

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

- **Reference**

<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>3</td>
</tr>
<tr>
<td>( \text{val} )</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>( \text{val}+1 )</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>&amp;( \text{val}[2] )</td>
<td>int *</td>
<td></td>
</tr>
<tr>
<td>( \text{val}[5] )</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>( \ast(\text{val}+1) )</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>( \text{val} + i )</td>
<td>int *</td>
<td></td>
</tr>
</tbody>
</table>
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 \( T^* \)

\[
\begin{align*}
\text{int } \text{val[5];} \\
\end{align*}
\]

■ 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>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      //\text{val[1]}</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>x + 4 * i  //&amp;val[i]</td>
</tr>
</tbody>
</table>
Array Example

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

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

- 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

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

**x86-64**

```
# %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]++;
}
```

```assembly
# %rdi = z
movl $0, %eax
jmp .L3
.L4:
    addl $1, (%rdi,%rax,4)
    addq $1, %rax
.L3:
    cmpq $4, %rax
    jbe .L4
    rep; ret
```
Understanding Pointers & Arrays #1

<table>
<thead>
<tr>
<th>Decl</th>
<th>A1, A2</th>
<th>*A1, *A2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comp</td>
<td>Bad</td>
</tr>
<tr>
<td>int A1[3]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>int *A2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Comp**: 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>A1 , A2</th>
<th>*A1 , *A2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comp</td>
<td>Bad</td>
</tr>
<tr>
<td>int *A2</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

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

## Declaration Table

<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>
</tbody>
</table>

## Diagram

- **A1**: Declared array with allocated memory.
- **A2**: Declared array with allocated memory.
- **A3**: Declared array with allocated memory.

### Allocations
- **Allocated pointer**
- **Unallocated pointer**
- **Allocated int**
- **Unallocated int**
### 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>
</tbody>
</table>

**Diagram:**
- **A1**
- **A2**
- **A3**

**Legend:**
- Allocated pointer
- Unallocated pointer
- Allocated int
- Unallocated int
Multidimensional (Nested) Arrays

- **Declaration**
  
  
  \[ T \ A[R][C] ; \]

  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns

- **Array Size**
  
  - \( R \times C \times \text{sizeof}(T) \) bytes

- **Arrangement**
  
  - Row-Major Ordering

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

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

A[0][0]    • • •    A[0][C-1]
       •                •
       •                •
A[R-1][0]  • • •    A[R-1][C-1]
```

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

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

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

```
zip_dig pgh[4];
```

- “zip_dig pgh[4]” equivalent to “int pgh[4][5]”
  - 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 of type \( T \)
  - Starting address \( A + i * (C \times \text{sizeof}(T)) \)

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

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

- For \( A[i] \), the starting address is \( A + (i \times C \times 4) \)
- For the \( R-1 \)th row, the starting address is \( A + ((R-1) \times C \times 4) \)
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];
}
```

# %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 $A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K$

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

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A[0]</td>
<td>A[i]</td>
</tr>
<tr>
<td>[0]</td>
<td>[i]</td>
</tr>
</tbody>
</table>

$A[0][0]$, $A[R-1][0]$,
$A_{[i][j]}$,
$A_{[R-1][C-1]}$

$A + (i \times C \times 4)$

$A + ((R-1) \times C \times 4)$

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

Array Elements

- `pgh[index][dig]` is int
- Address: `pgh + 20*index + 4*dig`
  
  \[ = pgh + 4*(5*index + dig) \]
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:

```
Mem[pgh+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

- `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];
}
```

```
# 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 # Mem[A + 64*i + 4*j]
ret
```
**n X n Matrix Access**

- **Array Elements**
  - `size_t n;`
  - `int A[n][n];`
  - Address `A + i * (C * K) + j * 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];

int main(int argc, char** argv) {
    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);
    return 0;
}
```

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

```c
#include <stdio.h>
#define ZLEN 5
#define PCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
    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);
    return 0;
}
```

```
linux> ./array
result: 9
```
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/10968
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

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

```
leaq (%rdi,%rsi,4), %rax
ret
```

### 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];
}
```
Following Linked List #1

**C Code**

```c
long length(struct rec*r) {
    long len = 0L;
    while (r) {
        len ++;
        r = r->next;
    }
    return len;
}
```

**Loop assembly code**

```
.L11:          # loop:
    addq  $1, %rax  #   len ++
    movq  24(%rdi), %rdi  #   r = Mem[r+24]
    testq %rdi, %rdi  #   Test r
    jne   .L11       #   If != 0, goto loop
```

---

**Structure**

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

**Register Table**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>r</td>
</tr>
<tr>
<td>%rax</td>
<td>len</td>
</tr>
</tbody>
</table>
Carnegie Mellon

Following Linked List #2

■ C Code

```c
void set_val(struct rec *r, int val) {
    while (r) {
        size_t i = r->i;
        // No bounds check
        r->a[i] = val;
        r = r->next;
    }
}
```

Following Linked List #2

C Code

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

Element i

<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:  # loop:

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

- **Unaligned Data**

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

- **Aligned Data**

  - Primitive data type requires $B$ bytes implies
  - Address must be multiple of $B$
Alignment Principles

■ Aligned Data
  ▪ Primitive data type requires $B$ bytes
  ▪ Address must be multiple of $B$
  ▪ 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 cache lines (64 bytes).
      Intel states should avoid crossing 16 byte boundaries.
      [Cache lines will be discussed in Lecture 11.]
    ▪ Virtual memory trickier when datum spans 2 pages (4 KB pages)
      [Virtual memory pages will be discussed in Lecture 17.]

■ 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 $0_2$

- **4 bytes: int, float, ...**
  - lowest 2 bits of address must be $00_2$

- **8 bytes: double, long, char *, ...**
  - lowest 3 bits of address must be $000_2$
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;
```

Multiple of K=8

External padding

7 bytes
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
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;
```

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

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

12 bytes

- Effect (largest alignment requirement K=4)

<table>
<thead>
<tr>
<th>i</th>
<th>c</th>
<th>d</th>
<th>2 bytes</th>
</tr>
</thead>
</table>

8 bytes
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.

![Memory Allocation Diagram]

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.

```
<table>
<thead>
<tr>
<th>a</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>X</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
<td>e</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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.

```
+--------+--------+--------+--------+--------+--------+--------+--------+--------+--------+
|   a    |   b    |   c    |   d[3] |   e    |   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.

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

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 (but registers are 32 bytes instead of 16)
  - Documented in book
Programming with SSE4

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;
}
```

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

# 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

```c
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 ZF, PF and CF
  - Zeros OF and SF

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

**Parity Flag**

- UNORDERED: ZF,PF,CF ← 111
- GREATER_THAN: ZF,PF,CF ← 000
- LESS_THAN: ZF,PF,CF ← 001
- EQUAL: ZF,PF,CF ← 100
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 #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`
**Declaration**

- `int A1[3][5]`
- `int *A2[3][5]`
- `int (*A3)[3][5]`
- `int *(A4[3][5])`
- `int *(A5[3])[5]`

**Table Diagram**

- **A1** (Allocated Pointer)
- **A2/A4** (Allocated Pointer to Unallocated Int)
- **A3** (Unallocated Pointer)
- **A5** (Allocated Pointer to Unallocated 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>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`

### Additional Table

<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>