Machine-Level Programming IV: Data

15-213/18-213/15-513: Introduction to Computer Systems
8th Lecture, February 12, 2019

Instructors:
Seth C. Goldstein, Brandon Lucia, Franz Franchetti, and Brian Raling
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

\( T \ A[L] ; \)

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

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

Reference Type Value

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>val</td>
<td>x</td>
<td>x + 4</td>
</tr>
<tr>
<td>val+1</td>
<td>x + 8</td>
<td>x + 12</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>x + 16</td>
<td>x + 20</td>
</tr>
<tr>
<td>val[5]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*(val+1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>val + i</td>
<td></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 $T^*$

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

- Reference | Type | Value
  - $val$ | int * | $x$
  - $val+1$ | int * | $x + 4$
  - $&val[2]$ | int * | $x + 8$
  - $val[5]$ | int | ??
  - $(val+1)$ | int | 5 //val[1]
  - $val + i$ | int * | $x + 4 * i$ //&val[i]
Array Example

```c
#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

```c
int get_digit(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)`
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

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

- **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></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>

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

- **A2**
  - Allocated pointer
  - Unallocated pointer
  - Allocated int
  - Unallocated int

- **A3**
  - 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**: Allocated int
- **A2**: Unallocated pointer
- **A3**: Allocated int
- **Allocated pointer**: Allocated int
- **Unallocated pointer**: 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];
```
Nested Array Example

```c
#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]” 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 $A + i \times (C \times \text{sizeof}(T))$

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

![Diagram showing array access](attachment:image.png)
Nested Array Row Access Code

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

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

int \texttt{*get_pgh_zip(\texttt{int index})}
{\texttt{
    return pgh[index];
}}

Machine Code
\begin{verbatim}
# %rdi = index
    leaq (%rdi,%rdi,4),%rax # 5 * index
    leaq pgh(,%rax,4),%rax # pgh + (20 * index)
\end{verbatim}
Nested Array Element Access

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

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

```plaintext
A+(i*C*4)  A+(i*C*4)+(j*4)
```

```
A[0]
A[0]  ...  A[C-1]
```

```
A[i]
A[i]  ...  A[i]
```

```
A[R-1]
A[R-1]  ...  A[R-1]
```

```
A  
A[0]  ...  A[0]
```
Nested Array Element Access Code

Array Elements

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

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

```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 \texttt{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];
}
```

```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
```
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
lea (%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;
}
```
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
#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;
}
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`
Allocated pointer
Allocated pointer to unallocated int
Unallocated pointer
Allocated int
Unallocated int

A1

A2/A4

A3

A5

Declaration

int A1[3][5]
int *A2[3][5]
int (*A3)[3][5]
int *(A4[3][5])
int (*A5[3])[5]
# 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 A1[3][5]</td>
<td>N</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>
Break Time!

whipsaw: "to beset or victimize in two opposite ways at once"

Check out:

Quiz: day 8: Machine Data

https://canvas.cmu.edu/courses/8555
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

```
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;
};
```

### Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as \( r + 4 \times \text{idx} \)

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

```
# 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 Code

```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>

### Element i

0 16 24 32

```assembly
.L11:  
    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

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

```c
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 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 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;
```

![Diagram showing alignment and padding](image)
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

```
short get_j(int idx)
{
    return a[idx].j;
}
```

```
# %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;
```

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

- Effect (largest alignment requirement K=4)
Example Struct Exam Question

Problem 5. (8 points):

Struct alignment. Consider the following C struct declaration:

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.

```
+-------------------+-------------------+-------------------+-------------------+-------------------+-------------------+
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
|                   |                   |                   |                   |                   |                   |
```

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.

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

```
adddddccccccbBBBBBBBBBB
+--------------------------+
|                         |
enario 000000000000000000
|                         |
+--------------------------+
```

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 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;
}
```

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

```assembly
# 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;
}
```

```bash
# 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

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