COMP201
Computer Systems & Programming

Lecture #26 – Data and Stack Frames

Aykut Erdem // Koç University // Fall 2020
About your midterm exam

• **Zoom** and **Blackboard**

• It will become **active at 11:50am**, and it will be released in **3 parts**:
  • Each contains multi-part questions
  • 30 mins to complete in one sitting.
  • Backtracking allowed.

• **Open books, open notes.** You can use your own lecture notes, lecture slides and textbook, but nothing else.

• **We won’t take any questions during the exam.** Please write “ERROR” if you think something is wrong.
Recap

- Revisiting `%rip`
- Calling Functions
  - The Stack
  - Passing Control
  - Passing Data
  - Local Storage
- Register Restrictions
- Pulling it all together: recursion example
Plan for Today

• Arrays
• Structures
• Floating Point

Disclaimer: Slides for this lecture were borrowed from
—Randal E. Bryant and David R. O’Hallaroni’s CMU 15-213 class
Lecture Plan

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

• Structures
• 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

```c
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 $T^*$

<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</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 ku = {3,4,4,5,0};

zip_dig cmu;
16 20 24 28 32 36

zip_dig mit;
36 40 44 48 52 56

zip_dig ku;
56 60 64 68 72 76

• 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

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)

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

zip_dig ku;

# %rdi = z
# %rsi = digit
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
    jmp   .L3               # i = 0
    .L3:                        # goto middle
        addl  $1, (%rdi,%rax,4) # z[i]++
        addq  $1, %rax
        .L4:                        # loop:
            cmpq  $4, %rax
            jbe   .L4               # i:4
        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 * C * K bytes

Arrangement
• Row-Major Ordering

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

1

```
A[0][0]  • • •  A[0][C-1]

A[0][1]  • • •  A[0][C-1]

A[R-1][0]  • • •  A[R-1][C-1]
```

4*R*C Bytes

```
#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 }};

• “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
- Each element of type $T$ requires $K$ bytes
- Starting address $A + i \times (C \times K)$

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

![Diagram showing nested array row access]
Nested Array Row Access Code

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

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)`
Nested Array Element Access

Array Elements

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

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

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

```
leaq (%rdi,%rdi,4), %rax  # 5*index
addl %rax, %rsi          # 5*index+dig
movl pgh(%rsi,4), %eax   # M[pgh + 4*(5*index+dig)]
```

Array Elements

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

zip_dig cmu = { 1, 5, 2, 1, 3 };  
zip_dig mit = { 0, 2, 1, 3, 9 };  
zip_dig ku = { 3, 4, 4, 5, 0 };  

#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ku};

- Variable univ denotes array of 3 elements
- Each element is a pointer
  - 8 bytes
- Each pointer points to array of int's
Element Access in Multi-Level Array

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

- **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

```
salq $2, %rsi                 # 4*digit
addq univ(%rdi,8), %rsi      # p = univ[index] + 4*digit
movl (%rsi), %eax            # return *p
ret
```
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 × 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 × 16 Matrix Access

/* 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  # M[a + 64*i + 4*j]
ret

Array Elements
• Address A + i*(C*K) + j*K
• C = 16, K = 4
n × n Matrix Access

/* 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];
}

Array Elements

- Address $A + i \cdot (C \cdot K) + j \cdot K$
- $C = 16, K = 4$
- Must perform integer multiplication
Practice 1: Reverse Engineering

#define M ??
#define N ??

long P[M][N];
long Q[N][M];
long sum_elem(long i, long j)
{
    return P[i][j] + Q[j][i];
}

What is the value of M and N?
M = 5 and N = 7

Event code: 73165
Lecture Plan

• Arrays

• 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 \( r + 4*\text{idx} \)

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

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

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

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

.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
```
Practice 2: Reverse Engineering

Fill in the blanks by inspecting the assembly code generated by gcc.

struct test {
    short *p;
    struct {
        short x;
        short y;
    } s;
    struct test *next;
};

void st_init(struct test *st) {
    st->s.y = __________;  // Get st->s.x
    st->p = __________;    // Save in st->s.y
    st->next = __________; // Compute &(st->s.y)
}

# void st_init(struct test *st)
# st in %rdi
1 st_init:
  2 movl 8(%rdi), %eax         Get st->s.x
  3 movl %eax, 10(%rdi)        Save in st->s.y
  4 leaq 10(%rdi), %rax       Compute &(st->s.y)
  5 movq %rax, (%rdi)          Store in st->p
  6 movq %rdi, 12(%rdi)        Store st in st->next
  7 ret
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 $K$

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

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

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```
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 (K=4)

```
<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>c</td>
<td>d</td>
<td></td>
<td>2 bytes</td>
</tr>
</tbody>
</table>
```
Determine the offset of each field, the total size of the structure, and its alignment requirement for x86-64.

```c
struct mystruct {
    int *a;
    float b;
    char c;
    short d;
    long e;
    double f;
    int g;
    char *h;
};
```

<table>
<thead>
<tr>
<th>Field</th>
<th>*a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>*h</th>
<th>Total</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>8</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>56</td>
<td>8</td>
</tr>
<tr>
<td>Offset</td>
<td>0</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>8</td>
</tr>
</tbody>
</table>

Rearranged structure with minimum wasted space:

<table>
<thead>
<tr>
<th>Field</th>
<th>a</th>
<th>h</th>
<th>f</th>
<th>e</th>
<th>b</th>
<th>g</th>
<th>d</th>
<th>c</th>
<th>Total</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>48</td>
<td>8</td>
</tr>
<tr>
<td>Offset</td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>36</td>
<td>40</td>
<td>42</td>
<td>56</td>
<td>8</td>
</tr>
</tbody>
</table>

5 bytes padded to satisfy alignment requirement
Lecture Plan

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

• Structures

• Floating Point
Background

• History
  • x87 FP
    • Legacy, very ugly
  • Streaming SIMD Extensions (SSE) FP
    • SIMD: single instruction, multiple data
    • 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

• SIMD Operations: Single Precision
  - addps %xmm0, %xmm1
  - addsd %xmm0, %xmm1

• Scalar Operations: Double Precision
  - addss %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

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

```
    # p in %rdi, v in %xmm0
    movapd %xmm0, %ymm1      # Copy v
    movsd (%rdi), %ymm0      # x = *p
    addsd %ymm0, %ymm1       # t = x + v
    movsd %ymm1, (%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
Recap

• Arrays
• Structures
• Floating Point

That’s it for assembly!

Next time: security vulnerabilities, memory hierarchy