

Machine Programming Data

CSC 235 - Computer Organization

References

- Slides adapted from CMU

Outline

- Arrays
 - One-dimensional
 - Two-dimensional
 - Multi-level
- Structures
 - Allocation
 - Access
 - Alignment

Array Allocation

- Basic Principle
 - C syntax: `T A[L]`
 - Array of type `T` and length `L`
 - Contiguously allocated region of `L * sizeof(T)` bytes in memory
- Examples:
 - `char string[12]`: `12 * sizeof(char) = 12` bytes
 - `int val[5]`: `5 * sizeof(int) = 20` bytes

Array Access

- Basic Principle

- The array identifier can be used as a pointer to array element 0:
type T*

- Examples: `int val[5] = {1, 5, 2, 3, 1};`

Reference	Type	Value
<code>val[4]</code>	<code>int</code>	<code>1</code>
<code>val</code>	<code>int *</code>	<code>x</code>
<code>val+1</code>	<code>int *</code>	<code>x+4</code>
<code>&val[2]</code>	<code>int *</code>	<code>x+8</code>
<code>*(val+1)</code>	<code>int</code>	<code>5</code>

where `x` is the address of the first byte of `val`

Array Example

- C code

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

zip_dig_ku = {1, 9, 5, 3, 0};

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

Array Example

- Assembly code

```
# %rdi = z (starting address)
# %rsi = digit (index)
movl (%rdi, %rsi, 4), %eax # z[digit]
```

Array Loop Example

- C code

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


Array Loop Example

■ Assembly code

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

Multidimensional (Nested) Arrays

- Declaration: `T A[R][C];`
 - 2D array of type T
 - R rows, C columns
- Array Size
 - `R * C * sizeof(T)`
- Arrangement
 - Row-major ordering

$$\begin{bmatrix} A[0][0] & \dots & A[0][C-1] \\ \vdots & \ddots & \vdots \\ A[R-1][0] & \dots & A[R-1][C-1] \end{bmatrix}$$

Nested Array Example

- C code

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

zip_dig pgh[PCOUNT] =
    {{1, 9, 5, 1, 1},
     {1, 9, 5, 3, 6},
     {1, 9, 5, 6, 2},
     {1, 9, 5, 2, 9}};
```

Nested Array Row Access

- Row Vectors
 - $A[i]$ is array C elements of type T
 - Starting address $A + i * (C * \text{sizeof}(T))$
- Array Elements
 - $A[i][j]$ is element of type T , which requires K bytes
 - Address $A + i * (C * K) + j * k \rightarrow A + (i * C + j) * K$

Nested Array Row Access Code

- C code

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

- `pgh[index]` is array of 5 int values with starting address `pgh + 20 * index`

- Assembly code

```
# %rdi = index  
leaq (%rdi, %rdi, 4), %rax # 5 + index  
leaq pgh(,%rax,4), %rax   # pgh + (20 * index)
```

- Computes return address as `pgh + 4 * (index + 4 * index)`

Nested Array Element Access Code

■ C code

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

■ Assembly code

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

■ Array elements

- `pgh[index][digit]` is a value of type `int`
- Address: `pgh + 20 * index + 4 * dig`

Multi-Level Array Example

- C code

```
zip_dig b = { 1, 9, 5, 1, 1};  
zip_dig k = { 1, 9, 5, 3, 0};  
zip_dig t = { 1, 9, 5, 6, 2};  
  
#define TCOUNT 3  
int *town[TCOUNT] = {b, k, t}
```

- Variable `town` denotes an array of 3 elements
- Each element is a pointer (8 bytes)
- Each pointer points to array of `int` values

Element Access in Multi-Level Array

■ C Code

```
int get_town_digit(size_t index, size_t digit) {  
    return town[index][digit]  
}
```

■ Assembly Code

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

■ Computation

- Element access $\text{Mem}[\text{Mem}[\text{town}+8*\text{index}]+4*\text{digit}]$
- Must perform two memory reads

N × N Matrix Code

- Fixed dimensions (know at compile time)

```
#define N 16
typedef int fix_matrix[N][N];

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

N × N Matrix Code

- Variable dimensions, explicit indexing

```
#define IDX(n, i, j) ((i)*(n)+(j))
```

```
/* Get element at A[i][j] */
```

```
int vec_ele(size_t n, int *A, size_t i, size_t j) {  
    return A[IDX(n,i,j)];  
}
```

N × N Matrix Code

- Variable dimensions, implicit indexing (now supported by gcc)

```
/* Get element at A[i][j] */  
int vec_ele(size_t n, int A[n][n], size_t i, size_t j)  
    return A[i][j];  
}
```

16 × 16 Fixed Matrix Access

- Array elements

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

- Assembly Code

```
# 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
```

$n \times n$ Variable 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

- Assembly Code

```
# 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
```

Structure Representation

- Structure represented as block of memory big enough to hold all the fields
- Fields ordered according to declaration
 - Even if another ordering could yield a more compact representation
- Compiler determines overall size + positions of fields
- Example

```
struct rec {  
    int a[4];           // 16 bytes  
    size_t i;          // 8 bytes  
    struct rec *next; // 8 bytes  
}
```

Generating Pointer to Structure Member

- C code

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

- Assembly code

```
# r in %rdi, idx in %rsi  
leaq (%rdi, %rsi, 4), %rax  
ret
```

- Offset of each structure member is determined at compile time
- Compute as $r + 4 * idx$

Following Linked List Example

■ C Code

```
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
```


Following Linked List Example 2

■ C Code

```
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;
    }
}
```

■ Loop Assembly Code

```
.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 and Alignment

- Example Code

```
struct S1 {  
    char c;    // 1 byte  
    int i[2]; // 2*4 bytes  
    double v; // 8 bytes  
} *p;
```

- Unaligned Data

- c starts at address p
- i[0] starts at address p+1
- i[1] starts at address p+5
- v starts at address p+9

Structures and Alignment

- Example Code

```
struct S1 {  
    char c;    // 1 byte  
    int i[2]; // 2*4 bytes  
    double v; // 8 bytes  
} *p;
```

- Aligned Data

- Primitive data type requires **B** bytes implies address must be multiple of **B**
- `c` starts at address `p+0`
- `i[0]` starts at address `p+4` (3-byte gap)
- `i[1]` starts at address `p+8`
- `v` starts at address `p+16` (4-byte gap)

Alignment Principles

- Aligned Data
 - Primitive data type requires **B** bytes implies address must be multiple of **B**
 - Required on some machines; advised on x86-64
- Motivation for Aligning Data
 - Memory is accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
 - Inefficient to load or store data that spans cache lines
 - Virtual memory is tricky when data 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 0
- 4 bytes: `int`, `float`, ...
 - lowest 2 bits address must be 00
- 8 bytes: `double`, `long`, `char *`, ...
 - lowest 3 bits address must be 000

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 and structure length must be multiples of **K**
- Example:
 - **K** = 8 due to double element

```
struct S1 {  
    char c;    // 1 byte  
    int i[2]; // 2*4 bytes  
    double v; // 8 bytes  
} *p;
```

Meeting Overall Alignment Requirement

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

```
struct S2 {  
    double v; // 8 bytes  
    int i[2]; // 2*4 bytes  
    char c;   // 1 byte  
} *p;
```

- Since the char is last, the overall structure needs 7 bytes of padding at the end.

Arrays of Structures

- Overall structure length multiple of **K**
- Satisfy alignment requirement for every element.
- Example:

```
struct S2 {  
    double v; // 8 bytes  
    int i[2]; // 2*4 bytes  
    char c;   // 1 byte  
} a[10];
```

- Each array element is aligned to 24 bytes.

Accessing Array Elements

■ C code

```
struct S3 {  
    short i; // 2 bytes (2 byte gap)  
    float v; // 4 bytes  
    short j; // 2 bytes (2 byte padding)  
} a[10];
```

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

■ Assembly code

```
# %rdi = idx  
leaq (%rdi, %rdi, 2), %rax # 3*idx  
movzwl a+8(,%rax,4), %eax # a+8 resolved at link time
```

Saving Space in Structures

- Put large data types first
- Example 1 (12 bytes total):

```
struct S4 {  
    char c; // 1 byte (3 byte gap)  
    int i;  // 4 bytes  
    char d; // 1 byte (3 bytes padding)  
} *p;
```

- Example 2 (8 bytes total):

```
struct S5 {  
    int i; // 4 bytes  
    char c; // 1 byte (no gap)  
    char d; // 1 byte (2 bytes padding)  
} *p;
```

Summary

- Arrays
 - One-dimensional
 - Two-dimensional
 - Multi-level
- Structures
 - Allocation
 - Access
 - Alignment