Machine-Level Programming: Data

CMPU 224 – Computer Organization
Jason Waterman
**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 \( T^* \)

• Reference | Type | Value
---|---|---
\( val \) | int * | \( x \)
\( val[4] \) | int | 3
\( val+1 \) | int * | \( x + 4 \)
\&\( val[2] \) | int * | \( x + 8 \)
\( val[5] \) | int | ??
\( *(val+1) \) | int | 2
\( val + i \) | int * | \( x + 4 \ i \)
Array Example

- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general!

```c
#define ZLEN 5

int pok1[ZLEN] = { 1, 2, 6, 0, 1 };
int pok2[ZLEN] = { 1, 2, 6, 0, 2 };
int pok3[ZLEN] = { 1, 2, 6, 0, 3 };
```
Array Accessing Example

- Register %rdi contains starting address of array
- Register %rsi contains array index
- Desired digit at %rdi + 4*%rsi
- Use memory reference (%rdi,%rsi,4)

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

x86-64
```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```
Array Loop Example

```c
void zincr(int z[]) {
    size_t i;
    for (i = 0; i < ZLEN; i++)
        z[i] += 1;
}
```

```assembly
# %rdi = z
movl $0, %eax  #  i = 0
jmp .L3        #  goto test
.L4:           #  loop:
    addl $1, (%rdi,%rax,4) #  z[i] += 1
    addq $1, %rax  #  i++
.L3:           #  test
    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

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

\[
\text{int } A[R][C];
\]

\[
\begin{array}{cccc}
A[0][0] & \cdots & A[0][C-1] & A[1][0] \\
\cdots & \ddots & \cdots & \cdots \\
A[R-1][0] & \cdots & A[R-1][C-1]
\end{array}
\]

4*R*C Bytes
Nested Array Example

• `int pok[4][5];`
  • Variable `pok`: 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

```c
int pok[4][5] =
    {{1, 2, 6, 0, 1},
     {1, 2, 6, 0, 2},
     {1, 2, 6, 0, 3},
     {1, 2, 6, 0, 4}};
```

```
int pok[4][5];
```

```
1 2 6 0 1 1 2 6 0 2 1 2 6 0 3 1 2 6 0 4
```

3/22/2021

CMPU 224 -- Computer Organization
Nested Array Row Access

• Row Vectors
  • `int A[R][C];`
  • \( A[i] \) is array of \( C \) elements
  • Each element of type \( T \) requires \( K \) bytes
  • Starting address \( A + i \times (C \times K) \)

\[
\begin{align*}
A[0] & \quad \vdots \quad A[0][C-1] \\
A[0][0] & \quad \vdots \quad A[0][C-1] \\
\end{align*}
\]

\[
\begin{align*}
A[i] & \quad \vdots \quad A[i][C-1] \\
A[i][0] & \quad \vdots \quad A[i][C-1] \\
A+ (i \times C \times 4) & \quad \vdots \quad A+ ((R-1) \times C \times 4) \\
\end{align*}
\]

\[
\begin{align*}
A[R-1] & \quad \vdots \quad A[R-1][C-1] \\
A[R-1][0] & \quad \vdots \quad A[R-1][C-1] \\
\end{align*}
\]
Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type \( T \), which requires \( K \) bytes and has \( R \) rows and \( C \) cols
  - Address \( A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K \)

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

![Diagram showing nested array access](image)
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;
};
struct rec r;
```
Generating Pointer to Structure Member

• Generating Pointer to Array Element
  • Offset of each structure member determined at compile time
  • Accessing an element in array a: compute as $r + 4 \times \text{idx}$

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

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

```
# r in %rdi, idx in %rsi
movq (%rdi,%rsi,4), %rax
ret
```
Following Linked List

```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];
    size_t i;
    struct rec *next;
};
```

```assembly
jmp .L2 # goto test
.L3:
    # 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]
.L2:
    testq %rdi, %rdi # Test r
    jne .L3 # if !=0 goto loop
```

```
<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>
```
Structures & Alignment

• Unaligned Data

<table>
<thead>
<tr>
<th>c</th>
<th>i[0]</th>
<th>i[1]</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>p+1</td>
<td>p+5</td>
<td>p+9</td>
</tr>
</tbody>
</table>

• Aligned Data

• A primitive data type of $K$ bytes must have an address that is 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 data that spans quad word boundaries

• 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 $K$
    - $K_{\text{struct}} =$ Largest alignment of any element in struct
    - Initial address & structure length must be multiples of $K_{\text{struct}}$

- **Example:**
  - $K_{\text{struct}} = 8$, due to `double` element

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

- Largest alignment requirement $K_{\text{struct}}$
- Overall structure must be multiple of $K_{\text{struct}}$

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


Multiple of K=8
Arrays of Structures

- Overall structure length multiple of $K_{\text{struct}}$
- Satisfy alignment requirement for every element

```c
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```
Accessing Array Elements

• Compute array offset as `sizeof(s3) * idx`
  • `sizeof(S3) = 12`, including alignment spacers
• Element `j` is at offset 8 within structure
• Assembler gives offset `a+8`

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```

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

```
# %rdi = idx
lea (%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></th>
<th>3 bytes</th>
<th>3 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>i</td>
<td>d</td>
</tr>
</tbody>
</table>

12 bytes

<table>
<thead>
<tr>
<th></th>
<th>2 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>c</td>
</tr>
</tbody>
</table>

8 bytes
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