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

\[
\begin{align*}
\text{char string[12];} & \quad x \quad \ldots \quad x + 12 \\
\text{int val[5];} & \quad x \quad x + 4 \quad x + 8 \quad x + 12 \quad x + 16 \quad x + 20 \\
\text{double a[3];} & \quad x \quad x + 8 \quad x + 16 \quad x + 24 \\
\text{char *p[3];} & \quad x \quad x + 8 \quad x + 16 \quad x + 24
\end{align*}
\]
Array Access

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

\[
\text{int val[5];}
\]

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>int *</td>
<td>( x )</td>
</tr>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</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>2</td>
</tr>
<tr>
<td>val + i</td>
<td>int *</td>
<td>( x + 4 \ i )</td>
</tr>
</tbody>
</table>
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 };  
```

```
   int pok1[ZLEN];
   1  2  6  0  1
   16 20 24 28 32 36

   int pok2[ZLEN];
   1  2  6  0  2
   36 40 44 48 52 56

   int pok3[ZLEN];
   1  2  6  0  3
   56 60 64 68 72 76
```
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

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

# %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

10/3/2019

CMPU 224 -- Computer Organization
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

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

```
A[0][0]  \cdots  A[0][C-1]

\vdots

A[R-1][0]  \cdots  A[R-1][C-1]
```
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}};
```
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 * (C * K)`

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

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

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

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

```
A + ((R-1)*C*4)
```
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 \( A + i \cdot (C \cdot K) + j \cdot K = A + (i \cdot C + j) \cdot K \)

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

\[ A + (i \cdot C \cdot 4) + (j \cdot 4) \]
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;
};
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;
};
```

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

struct rec {
    int a[4];
    size_t 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;
    }
}

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

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;
};
```

```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;
};
```

```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;
};
```

```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;
};
```

```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;
};
```

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

• Unaligned 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;
```

• Aligned Data

- Multiple of 4
- Multiple of 8
- Multiple of 8
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 $K$
    - $K_{\text{struct}} = \text{Largest alignment of any element in struct}$
    - 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 $K_{\text{struct}}$

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

![Diagram showing alignment of struct S2](image)

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];
```

![Diagram showing memory layout of an array of structures](image)
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`

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

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

```
# %rdi = idx
lea (%rdi,%rdi,2),%rax  # 3*idx
movzlwl a+8(%rax,%eax),%eax
```
Saving Space

• Put large data types first

```c
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>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
<td>i</td>
<td>d</td>
<td></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