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 \( T^* \)

\[
\begin{array}{c|c|c|c|c|c|c}
 & 1 & 2 & 6 & 0 & 3 \\
\hline
x & x+4 & x+8 & x+12 & x+16 & x+20 \\
\end{array}
\]

• 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 * | \( x+4 \cdot i \)
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 };  // Example array 1
int pok2[ZLEN] = { 1, 2, 6, 0, 2 };  // Example array 2
int pok3[ZLEN] = { 1, 2, 6, 0, 3 };  // Example array 3
```

```
16 20 24 28 32 36

36 40 44 48 52 56

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)`)
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
  \[ T \cdot A[R][C] ; \]
  • 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
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 $A + i \times (C \times K)$

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

• Row Vector
  • \texttt{pok[index]} is array of 5 int’s
  • Row vector: starting address \texttt{pok + 20*index}

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

\begin{verbatim}
int *get_pok_zip(int index){
  return pok[index];
}
\end{verbatim}

\begin{verbatim}
# %rdi = index
leaq (%rdi,%rdi,4),%rax  # 5 * index
leaq pok(,%rax,4),%rax   # pok + (20 * index)
\end{verbatim}
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 $A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K$

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

- Address of $A[i][j]$:
  - $A + (i \times C \times 4) + (j \times 4)$

![Diagram](image)
Nested Array Element Access Code

• Array Elements
  • `pok[index][dig]` is type `int`
  • Address: `pok + 20*index + 4*dig = pok + 4*(5*index + dig)`

```c
int get_pok_digit(int index, int dig)
{
    return pok[index][dig];
}
```

```assembly
leaq (%rdi,%rdi,4), %rax  # 5*index
addl %rax, %rsi          # 5*index+dig
movl pok(%rsi,4), %eax   # M[pok+ 4*(5*index+dig)]
```
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 \( r + 4 \times \text{id} \)

```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) {
    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

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>

<table>
<thead>
<tr>
<th>Element i</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td>next</td>
</tr>
</tbody>
</table>

2/26/2019
CMPU 224 -- Computer Organization
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>

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

• Aligned Data

- A primitive data type of \( K \) bytes must have an address that is multiple of \( K \)

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i[0]</th>
<th>i[1]</th>
<th>4 bytes</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+0</td>
<td>p+4</td>
<td>p+8</td>
<td>p+16</td>
<td>p+24</td>
<td></td>
</tr>
</tbody>
</table>

Multiple of 8

Multiple of 4

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: \texttt{char}, ...  
  • no restrictions on address

• 2 bytes: \texttt{short}, ...  
  • lowest 1 bit of address must be 0_2

• 4 bytes: \texttt{int}, \texttt{float}, ...  
  • lowest 2 bits of address must be 00_2

• 8 bytes: \texttt{double}, \texttt{long}, \texttt{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

```c
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;
```
Arrays of Structures

- Overall structure length multiple of $K_{struct}$
- 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`

```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
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;
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>3 bytes</td>
<td>i</td>
<td>d</td>
<td>3 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12 bytes</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>c</td>
<td>d</td>
<td>2 bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8 bytes</td>
<td></td>
</tr>
</tbody>
</table>
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