Machine-Level Programming: Data

CMPU 224 – Computer Organization
Jason Waterman
Array Allocation

• Basic Principle

\[ T \text{ A}[L]; \]
• Array of data type \( T \) and length \( L \)
• Contiguously allocated region of \( L \times \text{sizeof}(T) \) bytes in memory

char string[12];

\[ x \quad x + 12 \]

int val[5];

\[ x \quad x + 4 \quad x + 8 \quad x + 12 \quad x + 16 \quad x + 20 \]

double a[3];

\[ x \quad x + 8 \quad x + 16 \quad x + 24 \]

char *p[3];

\[ x \quad x + 8 \quad x + 16 \quad x + 24 \]
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 \( T^* \)

```plaintext
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 )</td>
</tr>
</tbody>
</table>

### Pointer arithmetic
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)

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

```assembly
# %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
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];
```

10/12/2018CMPU 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 * (C * K)

\[ A[i] = A + (i * C * 4) \]

\[ A[R-1] = A + ((R-1) * C * 4) \]
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)}

```c
int *get_pok_zip(int index){
    return pok[index];
}
```

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

10/12/2018
CMPU 224 -- Computer Organization
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];
```

![Diagram showing array access in nested arrays](image)

$A + (i \times C \times 4) + (j \times 4)$
Nested Array Element Access Code

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

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

```
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

```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*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;
};
```

```plaintext
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>
```

Element i

<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

- A primitive data type of $K$ bytes must have an address that is multiple of $K$

• Aligned Data

- A primitive data type of $K$ bytes must have an address that is 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 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 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 $K_{\text{struct}}$
    - $K_{\text{struct}} = \text{Largest alignment of any element in struct}$
    - Initial address & structure length must be multiples of $K_{\text{struct}}$

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

\[
\begin{array}{c|c|c|c|c}
\text{c} & \text{3 bytes} & \text{i[0]} & \text{i[1]} & \text{4 bytes} \\
\hline
\text{p+0} & \text{p+4} & \text{p+8} & \text{p+16} & \text{p+24} \\
\end{array}
\]

- All elements must be aligned to their respective boundaries:
  - Multiple of 4
  - Multiple of 8

\[
\begin{array}{c|c|c|c|c}
\text{c} & \text{3 bytes} & \text{i[0]} & \text{i[1]} & \text{4 bytes} \\
\hline
\text{p+0} & \text{p+4} & \text{p+8} & \text{p+16} & \text{p+24} \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c}
\text{c} & \text{3 bytes} & \text{i[0]} & \text{i[1]} & \text{4 bytes} \\
\hline
\text{p+0} & \text{p+4} & \text{p+8} & \text{p+16} & \text{p+24} \\
\end{array}
\]
Meeting Overall Alignment Requirement

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

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

struct S5 {
    int i;
    char c;
    char d;
} *p;
```

```
c 3 bytes i d 3 bytes
```

12 bytes

```
i c d 2 bytes
```

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