Machine-Level Programming: Arrays and Structures

CMPU 224 – Computer Organization
Jason Waterman
Arrays in C

• Declaring arrays
  • type array_name [ array_size ];
  • Example: int count [5];
  • All elements of the array have the same type

• Declaring and initializing
  • int count[] = {4, 2, 9, 3, 5};

```
    4       2         9        3         5
```
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

```c
char string[12];
```

```
x x + 12
```

```c
int val[5];
```

```
x x + 4 x + 8 x + 12 x + 16 x + 20
```

```c
double a[3];
```

```
x x + 8 x + 16 x + 24
```

```c
char *p[3];
```

```
x x + 8 x + 16 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 \( T^* \)

```
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
• Return value stored in %rax
• Desired digit at %rdi + 4*%rsi
• Use memory reference (%rdi, %rsi, 4)

```c
int pok1[ZLEN];

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

```assembly
x86-64

# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```
Multidimensional (Nested) Arrays

- Declaration
  \[ T \ A[R][C]; \]
  - 2D array of data type \( T \)
  - \( R \) rows, \( C \) columns
  - Type \( T \) element requires \( K \) bytes

- Arrangement
  - Row-Major Ordering

- Array Size
  - \( R \times C \times K \) bytes

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

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

\[
\begin{array}{c|c|c}
\text{A[0][0]} & \cdots & \text{A[0][C-1]} \\
\hline
\text{A[1][0]} & \cdots & \text{A[1][C-1]} \\
\hline
\text{A[R-1][0]} & \cdots & \text{A[R-1][C-1]}
\end{array}
\]

\[ 4\times R\times C \text{ Bytes} \]
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
               // 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)`

![Diagram of nested array row access](image)
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];
```

```
A[0]   A[0][0]  A[0][C-1]
A[0]   A[0][0]  A[0][C-1]
   . .           .      .
A     A[i][0]   A[i][C-1]
A+(i*C*4)  A[i][0]   A[i][C-1]
A+(i*C*4)+(j*4)
A+(i*C*4)+(j*4)
A+(R-1)*C*4)
A+(R-1)*C*4)
```
Nested Array Row Access Code

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

- **Machine Code**
  - Computes and returns address
  - Compute as $\text{pok} + 4*(\text{index} + 4*\text{index})$

```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)
```
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

```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 $r + 4*\text{id}x$

```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
movl (%rdi,%rsi,4), %eax
ret
```
Next Linked List

- Return address of next node in the linked list

```c
struct rec* get_next(struct rec *r) {
    return r->next;
}
```

```assembly
movq 24(%rdi), %rax
ret
```

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>r</td>
</tr>
</tbody>
</table>

Next Linked List: 10/3/2023

• Return address of next node in the linked list

```c
struct rec* get_next(struct rec *r) {
    return r->next;
}
```

```assembly
movq 24(%rdi), %rax
ret
```

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>r</td>
</tr>
</tbody>
</table>
Structures & Alignment

- Unaligned Data

- 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 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
Satisfying Alignment with Structures

- Within structure:
  - Must satisfy each element’s alignment requirement

- Overall structure placement
  - Each structure has alignment requirement $K$
    - $K_{struct} = $ Largest alignment of any element in struct
  - Initial address & structure length must be multiples of $K_{struct}$

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

Multiple of $K=8$
Arrays of Structures

- Overall structure length multiple of $K_{\text{struct}}$
- Satisfy alignment requirement for every element
Accessing Array Elements of a Structure

• 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;
}
```

```assembly
# %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;
```

<table>
<thead>
<tr>
<th>c</th>
<th>3 bytes</th>
<th>i</th>
<th>d</th>
<th>3 bytes</th>
</tr>
</thead>
</table>

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