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};`

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
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 \( T^* \)

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

```assembly
# %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

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

```
A[0][0]  \cdots  A[0][C-1]
   \vdots
   \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} };
```

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

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

- `A[i][0]` to `A[i][C-1]` represent elements within a specific row.
- The diagram illustrates how to access elements using their row and column indices, considering the starting address and the size of one row.
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 \( A + i \times (C \times K) + j \times K = A + (i \times C + j) \times K \)

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

```
A[0][0]  \cdots  A[0][C-1]
\downarrow                        \downarrow
A[0]                      A[0]
\downarrow                        \downarrow
A                        A+(i*C*4)
\downarrow                        \downarrow
\vdots
\downarrow                        \downarrow
A[i][0]  \cdots  A[i][C-1]
\downarrow                        \downarrow
A[i]                      A[i]
\downarrow                        \downarrow
A+(i*C*4)                A+(i*C*4)+(j*4)
```

```
A[R-1][0]  \cdots  A[R-1][C-1]
\downarrow                        \downarrow
\downarrow                        \downarrow
A+(R-1)*C*4                A+(R-1)*C*4)
```
Nested Array Row Access Code

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

• Machine Code
  • Computes and returns address
  • Compute as `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)
```
Nested Array Element Access Code

• Array Elements
  • \texttt{pok[index][dig]} is type \texttt{int}
  • Address: \texttt{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];
}
```

```asm
# 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;
}
```

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

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```
Structures & Alignment

• Unaligned Data

```
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 $K$
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 $K$
    • $K_{\text{struct}} = \text{Largest alignment of any element in struct}$
    • 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;
```

```
v   i[0]   i[1]   c
p+0 p+8 p+16 p+24
```

Multiple of K=8
Arrays of Structures

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

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

```
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;
```
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