Arrays

- In high-level languages, we have several techniques available for constructing data structures:
  - One-dimensional arrays
  - Multi-dimensional arrays
  - Structs
  - Bit sets
  - Linked lists
  - Trees
  - etc
- At the assembly language level, all of these approaches map onto single-dimension arrays alone.
- In order to emulate one of the other structures, we must create a mapping from the high-level approach to an offset into a linear list of memory bytes.
Arrays

- The origin of an array is not always the same:
  - In C, C++, and Java, arrays start with index 0
  - In BASIC and FORTRAN, arrays start with index 1
  - In Pascal and Delphi arrays may start at any index chosen by the programmer
- The simplest data structure is the one-dimensional array
- A one-dimensional array closely matches its equivalent structure in assembly language.
- Associated with each array is the base address, usually denoted by the name of the array
One-dimensional Arrays

Storage allocation

low addresses

\[
\text{address of } a[i] = \text{base address} + i \times \text{element_size}
\]

for 0-origin indexing

high addresses

\[
\text{address of } a[i] = \text{base address} + ((i - \text{origin}) \times \text{element_size})
\]

for nonzero-origin indexing
One-dimensional Arrays

<table>
<thead>
<tr>
<th>base address</th>
<th>base + 4</th>
<th>base + 8</th>
<th>base + i*4</th>
<th>base + (i+1)*4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>other</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stuff</td>
<td>a[0]</td>
<td>a[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a[i]</td>
<td>a[i]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>other stuff</td>
<td>a[n-1]</td>
<td>a[n-1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>other stuff</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(low addresses)

(high addresses)
One-dimensional Arrays

- In order to work with arrays, we must first learn how to represent an array.

- Consider the following C declaration:

  ```c
  int a[100];
  ```

  This is just 100 integers. In ARM, we have to make room for 400 bytes (4 * 100)

  ```asm
  .data
  .align 2
  a: .skip 400
  ```
One-dimensional Arrays

• To reference any element in an array we need to have both the starting address of the array (the base address) and the index of the desired element.
• In ARM, the base address of an array must be in a register.
• The easiest method to get the base address of an array into a register in ARM assembly language is to use the ldr pseudo-instruction

\[ ldr \ rb, =a \]
Review of ARM Addressing Modes

- Accessing an array involves calculating the address of the accessed item.
- Recall that the ARM instruction set provides several indexing modes for accessing array elements:

  \[ [Rn] \quad \text{Register} \]
  \[ \text{ldr } r0, [r1] \]

  \[ [Rn, \#\pm \text{imm}] \quad \text{Immediate offset} \]
  \[ \text{ldr } r2, [r1, \#12] @ r2 \leftarrow *(r1 + 12) \]

  \[ [Rn, \pm Rm] \quad \text{Register offset} \]

  \[ [Rn, \pm Rm, \text{shift}] \quad \text{Scaled register offset} \]
One-dimensional Arrays

\[[Rn, \#\pm\text{imm}]!\] Immediate pre-indexed
\[\text{ldr } r2, [r1, \#12]! \quad @r1 \leftarrow r1 + 12 \text{ then } r2 \leftarrow \ast r1\]

\[[Rn, \pm Rm]!\] Register pre-indexed

\[[Rn, \pm Rm, \text{shift}]!\] Scaled register pre-indexed

\[[Rn], \#\pm\text{imm}\] Immediate post-indexed
\[\text{ldr } r2, [r1], +4 \quad @ r2 \leftarrow \ast r1 \text{ then } r1 \leftarrow r1 + 4\]

\[[Rn], \pm Rm\] Register post-indexed

\[[Rn], \pm Rm, \text{shift}\] Scaled register post-indexed
One-dimensional Arrays

• To access an array element in a stack-allocated array, 0-origin using ldr/str, use one of the following address calculations:

1) ldr rd, [rb, #n] @ ea = value in rb + n
2) ldr rd, [rb, rn] @ ea = value in rb + value in rn
3) ldr rd, [rb, rn, lsl #2] @ ea = value in rb + 4*value in rn
4) ldr rd, [rb, #4]! @automatically updates rb

• Similarly for ldrh, except use lsl #1 in case 3 and use #2 in case 4.
• Access an array element for stores in an analogous manner
Load/Store Exercise

Assume an array of 25 integers. A compiler associates y with r1. Assume that the base address for the array is located in r0. Translate this C statement/assignment:

\[
\text{array}[10] = \text{array}[5] + y;
\]
Assume an array of 25 integers. A compiler associates y with r1. Assume that the base address for the array is located in r0. Translate this C statement/assignment:

```c
```

```assembly
mov r2, #5
mov r3, #10
ldr r4, [r0, r2, lsl #2] // r4 = array[5]
add r4, r4, r1 // r4 = array[5] + y
```
One-dimensional Arrays

/*
   for (i = 0; i < 100; i++)
       a[i] = i;
*/

.text
.globl main
.type %function, main
main:
    ldr r1, =a    /* r1 ← &a */
    mov r2, #0   /* r2 ← 0 */
loop:
    cmp r2, #100 /* Have we reached 100 yet? */
    bge done    /* If so, we're done */
    add r3, r1, r2, lsl #2 /* r3 ← r1 + (r2*4) */
    str r2, [r3] /* *r3 ← r2 */
    add r2, r2, #1 /* r2 ← r2 + 1 */
    b loop
done:
    mov r0, #0
    bx lr
.section .bss
a:  .skip 400
One-dimensional Arrays

/* Another Approach  
   for (i = 0; i < 100; i++)  
     a[i] = i;  
*/

    mov r2, #0
loop:
    cmp r2, #100       /* Have we reached 100 yet? */
    bge done          /* If so, we're done */
    str r2, [r1], +4  /* *r1 ← r2 then r1 ← r1 + 4 */
    add r2, r2, #1    /* r2 ← r2 + 1 */
    b loop

done:
    mov r0, #0
    bx lr

.section .bss
a:    .skip 400
Traversing an array (visiting all elements)

Two options (using ldr):
1) Adding a scaled index to a base address
   // element = a[i];

   add r3, sp, #0  @ can set base address outside loop
   mov index_r, #0
   ...

   loop:
   ...
   ldr element_r, [base_r, index_r, lsl #2]
   ...
   add index_r, index_r, #1
   ...

Traversing an array (visiting all elements)

2) dereferencing a pointer and incrementing the pointer
   
   ```c
   // int *ptr = a;
   // element = *ptr++
   
   add ptr, sp, #0 //init ptr to base_addr outside loop
   ...
   
   loop:
   ...
   
   ldr element_r, [ptr] // dereference ptr -- *ptr
   add ptr, ptr, #4     // postincrement ptr - ptr++
   ...
   ```
Traversing an array (visiting all elements)

2b) an alternate approach to dereferencing a pointer and incrementing the pointer in ARM

// int *ptr = a;
// element = *ptr++

add ptr, sp, #-4    // init ptr to base_addr - 4 outside
// the loop

... loop:
    ...

ldr element_r, [ptr, #4]!  // dereference ptr -- *ptr
// and postincrement ptr
    ptr++

...
Traversing an array (visiting all elements)

- note that in the second approach you can choose to make the loop control a test against an address limit rather than testing a loop counter

```c++
// int a[N];

//
// int *ptr = a; -- like a.begin() for a C++ vector
// int *limit = a+N; -- like a.end() for a C++ vector

// while( ptr != limit )
// {
//   ...
//   element = *ptr++;
//   ...
// }
```
Traversing an array (visiting all elements)

add ptr_r, sp, #-4 @ init ptr to base_address-4
mov r0, N, lsl #2 @ init limit to point to
    @ one element beyond the last
add limit_r, ptr_r, r0 @ element in the array
...
loop:
cmp ptr_r, limit_r
beq done
...
ldr element_r, [ptr_r, #4]! @ dereference ptr -- *ptr
    @ and postincrement ptr
    @ ptr++
...
ba  loop
done:
...
**Two-dimensional Arrays**

- storage allocation is now more difficult – row-major or column-major

<table>
<thead>
<tr>
<th>row major</th>
<th>column major</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(0,0)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(1,0)</td>
</tr>
<tr>
<td>(0,2)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(1,0)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(1,1)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>(1,2)</td>
<td>(1,2)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Two-dimensional Arrays

C is row-major, FORTRAN is column-major
e.g., in C

```c
int mat[2][2];
```

<table>
<thead>
<tr>
<th>mat[0][0]</th>
<th>mat[0][1]</th>
<th>mat[1][0]</th>
<th>mat[1][1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mat + 0</td>
<td>mat + 4</td>
<td>mat + 8</td>
<td>mat + 12</td>
</tr>
</tbody>
</table>

\[
\text{address of } a[i][j] \text{ in row-major storage order} = \\
\text{base_address + } [i \times \#_{\text{elements_per_row}} + j] \times \text{element_size}
\]

for 0-origin indexing

\[
\text{address of } a[i][j] \text{ in row-major storage order} = \text{base_address + } \\
[(i-\text{origin1}) \times \#_{\text{elements_per_row}} + (j-\text{origin2})] \times \text{element_size}
\]

for non-zero origin indexing
Two-dimensional Arrays

consider this example C code

```c
int main(void) {
    int a[5][5];
    register int i=3, j=4;
    a[i][j] = 0;
}
```
Two-dimensional Arrays

annotated assembly output from compiler

.global main
main:
push    {r4, r5, r7}           // save space for 5x5 array
sub     sp, sp, #108
add     r7, sp, #0
mov     r4, #3                // r4 = i = 3
mov     r5, #4                // r5 = j = 4
mov     r3, r4                // r3 = i
lsr     r3, r3, #2            // r3 = 4*i
adds    r3, r3, r4            // r3 = 4*i + i = 5*i
adds    r3, r3, r5            // r3 = 5*i + j
lsr     r3, r3, #2            // r3 = 4*(5*i + j)
add      r2, r7, #104         // r2 = r7 + 104
adds     r3, r2, r3           // r3 = r7 + 104 + 4*(5*i + j)
mov      r2, #0               // r2 = 0
str      r2, [r3, #-100]      // eff addr = r3 - 100
          // r7 + 104 + 4*(5*i+j) - 100
mov      r0, r3               // r0 = r7 + 4 + 4*(5*i+j)
add      r7, r7, #108
mov      sp, r7
pop      {r4, r5, r7}
bx       lr
Two-dimensional Arrays

A cleaner approach to do $a[i][j] = 0$;

```
sub    sp,   sp,   #108
add    base_r, sp,   #0
mov    r0,   i_r,    lsl #2 @ 4*i
add    r0,   r0,   i_r    @ 5*i
add    r0,   r0,   j_r    @ 5*i + j
mov    offset_r, r0,    lsl #2 @ 4*( 5*i + j )

mov    r0,   #0
str    r0,   [base_r, offset_r]
```

assuming base_r, i_r, j_r, offset_r are register names for the base address, i, j, and the offset.