Array Indexing

Textbook Section 11.2

Arrays

Remember that an array is defined by:

- its start address
- the number of elements
- the size of an element

1D Array - Memory Storage

```
int array1[4] = { 12, 24, 36, 48 };

i.e.:

array1: dc.w    12,24,36,48 ; assume int = 16 bits (word)
```

1D Array - Memory Storage

If we assume zero-based indexing, then:

0		1	2	3
00	0C			

element	index	offset	address	value
array1[0]	0	0	array1 + 0	12
array1[1]	1	2	array1 + 2	24
array1[2]	2	4	array1 + 4	36
array1[3]	3	6	array1 + 6	48

1D Array - Indexing Formula

Generally, for a zero-based array A with element size “el_size”, A[index] has:

offset	=	index * el_size
address	=	A + index * el_size = address + offset
value	=	dereference(A + index * el_size)

Note: The above should be familiar, this is the indexed address indirect with offset address mode, with offset = 0, and Xn = index*el_size

- `0(An, Xn) or (An, Xn)`

1D Array - Bounds Checking

All code should check if the address (see previous slide) is a valid reference, can be done either by:

1. Index checking
    - 0 <= index < SIZE

or

2. Bounds checking
    - start_address <= address <= end_address
    - where the end_address is computed using SIZE-1

1D Array - Indexing Formula

- We call the lowest legal index the “lower bound”, or “LB”. Similarly, we call the largest legal index the “upper bound”, or “UB”.
- Generally, for an LB-based array, A[index] has:
  - offset = (index – LB) × el_size
  - address = A + (index – LB) × el_size
  - value = dereference(A + (index – LB) × el_size)

Dope Vector

- Arrays may encode their own size information (unlike in C/C++). An array can be implemented as a pair of values:
  1. a dope vector: a record containing these 3 fields:
     - lower bound
     - upper bound
     - element size
  2. the array memory
- A dope vector is a method of storing all relevant information about the array such that it is connected to the array and available at runtime

Dope Vector Example

```
; WORD doped_array [50..149]

MIN_INDEX           equ       50
MAX_INDEX           equ       149
WORD_SIZE           equ       2

LB                  equ       -6              ; dope vector field offsets
UB                  equ       -4              ; (measured from start of
EL_SIZE             equ       -2              ;  data)

                    dc.w      MIN_INDEX       ; why put these here?
                    dc.w      MAX_INDEX
                    dc.w      WORD_SIZE
doped_array:        ds.w      MAX_INDEX-MIN_INDEX+1
```

Dope Vector

The purpose of a dope vector is that a generic accessing function can be created. This function given the starting address of the array and the index will use the previous formula to generate the address of this index. Index or Bounds checking can also be done.

Both of these utilize the array’s dope vector

Dope Vector

Ex. in C the code nums[5] = 1; becomes:

```
subq.l      #6,sp                ; 4 for the address / 2 for error flag
            pea     nums     ; these could be combined
            move.w  #5,-(sp)
            jsr     access1D
            addq.l  #6,sp
            move.l  (sp)+,a0; address of num[5]
            tst.b   (sp)+
            beq     invalid_index_error
            move.w  #1,(a0)
```

```
access1D:       link      a6,#0
                movem.l   d0/a0,-(sp)
                move.l    ARRAY(a6),a0
                move.w    INDEX(a6),d0
                move.b    #TRUE,FLAG(a6)
                cmp.w     LB(a0),d0
                blo       a1D_error           ; both branches are
                cmp.w     UB(a0),d0
                bls       a1d_good            ; data dependent!
a1d_error:      move.b    #FALSE,FLAG(a6)
                bra       a1d_done
a1D_good:       sub.w     LB(a0),d0
                mulu      EL_SIZE(a0),d0
                adda.l    d0,a0
                move.l    a0,ADDRESS(a6)
a1d_done:       movem.l   (sp)+,d0/a0
                unlk      a6
                rts
```

2D Arrays

*N*D arrays can be represented as 1D array of (*N*-1)D arrays.

For example:

- a 2D array can be represented as a 1D array of 1D arrays
- a 3D array can be represented as a 1D array of 2D arrays
- a 4D array can be represented as a 1D array of 3D arrays
- etc.

2D Array - Dope Vector

Generally, for an array:

- type array[LBR..UBR,LBC..UBC]

For a 2D array the dope vector will require:

- UBR, LBR
- UBC, LBC
- element_size

2D Array - Dope Vector

Again this information can be stored before the array memory space, in a fixed format.

For an ND arrays the dope vectors will need:

- a pair of UB & LB fields for each dimension
- one element size field

2D Array - Memory Storage

2D array is a table – but memory is a linear structure, i.e. a 1D array.

Must map the elements of a 2D array into a 1D array!

2D arrays are stored in one of two possible orders:

- row major order: 2D array = 1D array of rows
- column major order: 2D array = 1D array of columns

2D Array - Indexing Formula

Generally, for an array:
type array[LBR..UBR,LBC..UBC]

the offset of array[row,col], stored in row major order, is given by:

how many full rows need to be skipped
# of rows to skip * # of elements in a row

how many elements in the desired row need to be skipped
col_index - LBC

row offset

column offset within row

2D Arrays - Indexing Formula

Putting together the above gives the total number of elements before the one being looked for

Still need to consider the size of each element, which gives:

```
offset = [ (row-LBR) * (UBC-LBC+1) + (col-LBC) ] * sizeof(type)
```

(row–LBR) * (UBC–LBC+1) → is the number of elements in complete rows
(col–LBC) → is the number of prior elements in the current row
sizeof(type) → is the size of each element in the array

For column major simply reverse every row and column reference.

2D Array - Error Checking Indices

Unlike a 1D array checking errors can only be done using Index Checking

Therefore, for a 2D array there will be two checks:

- valid row index
- valid column index

Why does Bounds Checking not work of a 2D array?