2-Dimensional Arrays

So far we have dealt only with 1-dimensional arrays. Theoretically there is no limit on the dimension of an array. The only difficulty in implementing arrays of higher dimension is calculating the correct index values. A 2-dimensional array is made up of rows and columns. These rows and columns are mapped into the 1-dimensional memory layout.

Example:

Char array[6][8]:

This is a 6*8 array of bytes and would be implemented in 68000.

array dc.b v11, v12, v13, v14, v15, v16, v17, v18 "Row1"
dc.b v21, v22, v23, v24, v25, v26, v27, v28 "Row2"
... dc.b v61, v63, v64, v65, v66, v67, v68 "Row6"

The array can be stored either in row or in column order.
2D Array

The most common is the row order method and it is this method we shall use.

If we assume that $a0$ is pointing to the start of an array of word sized data values, and we use $d0.w$ as the index register, to access individual elements of the 2D array requires a bit more effort than with the 1D counterpart.

We wish to access:

$$ \text{array [row][col]} $$

Index

We need to calculate the appropriate index value. The value is located at:

$$ a0 + \text{index where index} = (\text{row} \times 6 + \text{col}) \times \text{size} $$

Example: What index is required to access array[3][4]?

$$ \text{Index} = (3 \times 6 + 4) \times 1 $$

$$ = 22 $$

In general for a 2D array of element-size $S$, the correct index value for the element located at $\text{array[row, col]}$ is:

$$ (\text{row} \times \text{RowSize} + \text{col}) \times S $$

Higher Dimension Array

Extending to higher dimensions can be implemented by extending the dimension of the above formula.

For example the correct index for the value at $\text{array[w,x,y,z]}$ of a 4D array, of elements size $S$ and of dimension $SW \times SW \times SY \times SZ$ is given by the expression:

$$ (z \times SZ \times SY \times SX + y \times SY \times SX + x \times SX + w) \times S $$

Higher Dimension Array

Again, be wary of odd aligned rows and columns. If element size $S$ are words or greater there is no chance of errors.

With elements of size byte, however, an array with odd dimensions will cause data following the array to be odd aligned, possibly requiring padding of a byte.
Arrays of Records

The arrays so far have been composed of simple types, bytes, words, longwords, integer etc.

For the case of arrays of records, the situation is very similar.

We now use the displacement part of the \( d(A_n, X_i) \) addressing mode to access the individual member of the records in the array.

\( A_n \), as before points to the base of the array, and \( X_i \) is used as an index to point at the start of the records.

In this example, to access the 1ba3Mark field of the 2nd Record in the array we use the following code:

\[
\begin{align*}
\text{lea} & \quad $4100, a_0 \quad \text{"Start of array"} \\
\text{move.w} & \quad #16, d_1 \quad \text{"Index=1" Size} \\
\text{move.w} & \quad 8(a_0, d_1.w), d_2 \quad \text{"Get 1ba3Mark"}
\end{align*}
\]

As with arrays of simple types, incrementing through the array requires the size of the array elements to be added to the index register. In this case the array element size is 16 bytes.

Example:

This program scans through an array of records, each of which is a triplet of bytes.

The largest of the 3 bytes in the record is stored in a 2nd array of bytes.
### Pseudo-Code

```java
class Rec {
    byte a, b, c;
};
Rec Trip[20];
byte Max3[20];
i = 0;
while (i < 20) {
    x = Trip[i].a;
    y = Trip[i].b;
    z = Trip[i].c;
    r = Big3(x, y, z); → returns largest of 3 nums.
    Max3[i] = r;
    i++;
}
```

### Assembly Language

Because of the difference in size of the elements of the 2 arrays, we need two separate index registers.

**When implementing arrays traversal, a different variable should be used for each array and loop control variable.**

### Assembly Language Program

```assembly
* A Program which scans an array of records, * each of which is a byte triplet, determines the * largest of the 3 bytes and stores it in a 2nd * array of bytes.
* The function Big3(x, y, z) which returns the * largest of the 3 numbers is assumed to exist.

```
```
org $4000
lea Trip, a3
lea Max3, a1
clr.l d0 *i=0; (for Trip)
cr.l d4 *j=0; (for Max3)
cr.l d5 *k=0; (for loop)

```
```
cmpl.w #20, d5 *while(k<20)
beg E_WHILE *
move.b 0(a3, d0.w), d1 * x=Trip[i].a;
move.b 1(a3, d0.w), d2 * y=Trip[i].b;
move.b 2(a3, d0.w), d3 * z=Trip[i].c;
bsr Big3 *
r:=Big3(x, y, z);
move.b d7, 0(a1, d4.w) * Max3[i]:=r;
addq.w #3, d0 *
addq.w #1, d4 *
addq.w #1, d5 *
bra WHILE *

```
```
E_WHILE
rts
Trip db 1, 2, ....... *Rec Trip[20]
Max3 ds b 20 *byte Max3[20]
END```
```
Static & Dynamic Storage

Static storage allocation
- Variables, arrays, records and strings.
- Predefined fixed length.
- Size and structure is fixed at assembly time.
- Allows quick access.
- No need to search for data.
- But:
  - Size cannot be increased or decreased.
  - Poor utilization of memory

Dynamic Storage

Dynamic storage allocation
- Memory is assigned on request
- At run time:
  - The program request X bytes of contiguous memory from the operating system (take from a storage pool).
  - The program may deallocate this memory.
  - Allocation and deallocation must be provided by the OS/Monitor through system call.

Dynamic Data Structures

- Linked List
- Binary Trees
- Stacks
- Queues

Linked list

- Collection of data elements (nodes).
- Dynamically created.
  - Size is not fixed at assembly time.
  - N bytes of contiguous memory is allocated each time a node is added to the linked list.
**Nodes**

- A node has at least two items:
  - First item: Data structure
  - Second item: Pointer
    - Points to the next node in the linked list.
- First node is pointed to by just a pointer.
- Nodes are linked via pointers.
- The last node in the list contains a **null pointer**.

**A Four Node Linked List**

- Node 1: 'A3'
- Node 2: 'i'
- Node 3: 's'
- Node 4: 'fu'

**System Service**

- We assume a subroutine called "GETNODE" implements:
  - Allocation of memory through:
    - System calls
  - Returns the first address of contiguous block of addresses in $a5$
  - Size of block is determined by node size.
  - Linked list do not occupy a contiguous area in memory.
    - Only the addresses associated with a node must be sequential.

**Four Node Example**

- Pointer = $0040 0000$
- Node 1: 'A3' $008f 5670$
- Node 2: 'i' $0034 90f0$
- Node 3: 's' $01ff 0b20$
- Node 4: 'fu' $0000 0000$

- Pointer field
- Null Pointer
Example: Add Node (one)

- The pointer that holds the address of the first node is implemented in a4.
- GETNODE is invoked to allocate memory for the new node.
- Subroutine returns start address for node node in a5.
- Pointer field of new node must contain the address of the previous first node.
- Copy a5 in the pointer field 2(a5)
- The first two bytes are used for data field.

Example: Add Node (two)

- Then, make the new node first in list by:
  - Moving a5 (start address of new node) into a4 (pointer to first node in linked list).
  - Copy the two new ASCII character into the data field of the new node.

Program

```
* Subroutine that inserts a new node at the beginning of a linked list.
* GETNODE returns the start address of a 6 bytes block of Memory in a5.
* a4 points to the first node in the linked list
* a5 points to the first node in the linked list
* M. Manzke, 5th March 2001

INSERT bsr GETNODE  * allocate memory for node
move.l a4,2(a5)     * copy a4 into pointer field
movea.l a5,a4      * set start of list to new node
move.w d0,(a4)     * copy '1B' into data field
```

New Node

```
Node 1: '1B' $0040 0000
Node 1 -> Node 2: 'A3' $008f 5670
Node 2 -> Node 3: 'j' $0034 90f0

6 bytes allocated by GETNODE
```

Pointer = $0020 0000
Node 1: '1B' $0040 0000
Node 1 -> Node 2: 'A3' $008f 5670
Node 2 -> Node 3: 'j' $0034 90f0

more nodes...