* **Merging** – The term merging describes creation of an array by combining the values of two existing array. If the source array are unsorted then the target array will before just by merging the source arrays end to end. But if the source arrays are sorted then, we need to put the values in the target array also in order, for this we need to compare the values of source arrays before compying to the target array. According to the result of comparison we put available to the target array that will target array sorted.

⇒ Merging unsorting

* **Algorithm:**

Step 1:- Start merge(A[], n, B[], n_2, C[], n)

Step 2:- Declare counter x, y = 0

Step 3:- If $n_1 + n_2 > n_3$ Then
    3.A "Print" target Array has

```
not is enough space"
3.B Return

Step 4 :- for x = 0 to n - 1
Step 5 :- let c[~~x~~ x] = A[x]
Step 6 :- end for
Step 7 :- for y = 0 to ~~m~~ x + n2 - 1
Step 8 :- let c[x] = B[y]
Step 9 :- let x = x + 1
Step 10 :- End for
Step 11 :- End of function
```

When one or two array are sorted then we merge

```
void sortmerge(int A[], int n, int n1,
               int B[], int n2, int c[], int n3)
{
    int i, j, k = 0;
    if(n1 + n2) > n3
```

2·D Array :- Matrices are very importa in mathematics. A Matric is a collectio of rows and columns. Each intersection of these rows and columns holds a single value. Since computers and mathematic has a great relationship, matrices has same importance in the computers. Particularly ~~to~~ to work with graphics we need to handle matrixces because any computer display is formed as a matrix of dots. These matrices are stored in the computers memory in the form of a two-dimension -al array. Every programming languag support syntaxes to work with two-D array. A 2-D array is created by providing two subscribtion that is the number of rows in the table and number of columns per row. However we think the matric in two-dimensions but the value stored in the memory just in a sequence of wordlengths. Therefo to represent the 2-D arrays a language must have to use some order to stored the values of a matric. This stor can be done in two different manners -

A. Column ~~Ma~~ Measure order - In this order of storage the first column of all the rows is stored first then the second column and then the third one and so on.
for example - FORTRAN and PASCAL use this storage order.
For the column measure order the location of cell can be calculated by ~~following~~ evaluating the given formula -

$\underset{row}{m} \times \underset{col}{n}$

$loc(A[i][j]) = base + w[m(j-lbc)+(i-lbr)]$

or

$base + w(m \times j + i)$

if lbc and lbr are fixed to zero.

w - word length
lbc - lower bound column
lbr - " " row

B. Row Measure Order: -

In this order of storage all the columns for the first row are stored first then the ~~firs~~ ~~secon~~ columns for second row are stored and so on. Most of the languages in modern era use this order of storage including C and its decendents. In the row measure order the location address of a cell can be accessed by evaluating the following formula:-

$$loc(A[i][j]) = base + w[n(i - lbr) + (j - lbc)]$$

or

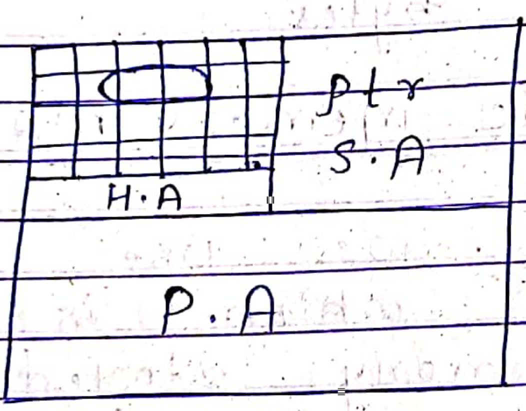
$$base + w(n \times i + j)$$

Refrencing a cell in 2D array:-
Since the values are stored row by row or column by column to access a cell we need to subscribe both the row index or column index for example in C it will be $A[r][c] \rightarrow A[3][5]$

Traversing the 2D array :-

If we want to release the memory then we use free() after this memory of the stack is reached at heap.

* Pointer is always stored in stack area and points the value in heap area.



Dynamic memory allocation

All the values we use in our program are stored in the memory location. If these memory locations are identifier they are used as variables otherwise they would be constant The memory binding for a value could be done at two levels of execution. They could be bound at the compile time called as static binding or they could be bound at the execution time called as dynamic binding. The memory location blind at the compile time could not be released untill the control goes outsize their scape. But the memory locations bind at runtime can be ~~rele~~ released whenever be want even the control is still in their scope. The static binding of memory is done by the compiler itself according to the declaration of variables and use of the constants. The static allocated global variables are stored in the prime area of the progra-m. While such local variables are stored in the stack of the function.

The dynamic binding of the memory is not by the compiler itself. The programm-er has to manage the dynamic allocation and deallocation of the memory themself by writting codes. Most of the languages provides some way for such memory management.

The dynamic memory management in 'C' language is based on four functions — malloc (), calloc realloc () and free (). These functions are prototyped in the header files. stdlib.h and malloc. The dynamic binding of memory is done in the heap area of the program, after then the base address of the allocated memory is return to access that location.

⇒ <u>malloc ()</u> :- This function accepts an integer argument and allocates the given no. of bytes in heap area and returns the base address as void pointer to manipulate that allocated location as a particular data types, we can convert this void pointer and stored them in that type of pointer.

Syntax —

```
    ptr = malloc (size);
    ex:-   int * ptr;
    ptr = (int *) malloc (size of ( [illegible] ) * 10)
```

⇒ calloc () :- Unlike the malloc() which allocate just a block of memory of certain bytes, the calloc function allocates no. of elements for an array. Therefore, we need to provide two argument to this function. first to specify the no. of elements and second to specify wordlength of each elements. It also returns a void pointer that can ~~co~~ be converted in a particular type.

Syntax -

Ptr = calloc (no. of element, word Length)

float * F ptr

float = calloc (7, 4);

⇒ Realloc () :- If after the first allocation the requirement is changed then we can use the realloc function to change the size of allocated memory, either to increase it or decrease it. It accepts two arguments. First the pointer holding address of allocated memory and second the new size.

ptr = realloc (ptr, new size);

Ex - fptr = realloc (fptr, 15);

⇒ free() :- If the allocated memory is no longer in use we can release that so, it can be used in some other work. To do this deallocation of the memory we can use the free function, which accepts the pointer holding the address of memory to be released.

Syntax -

free (pointer)

Ex - free (fptr);