02/11/2024, Saturday

Cambridge O-Level Computer Science 2210

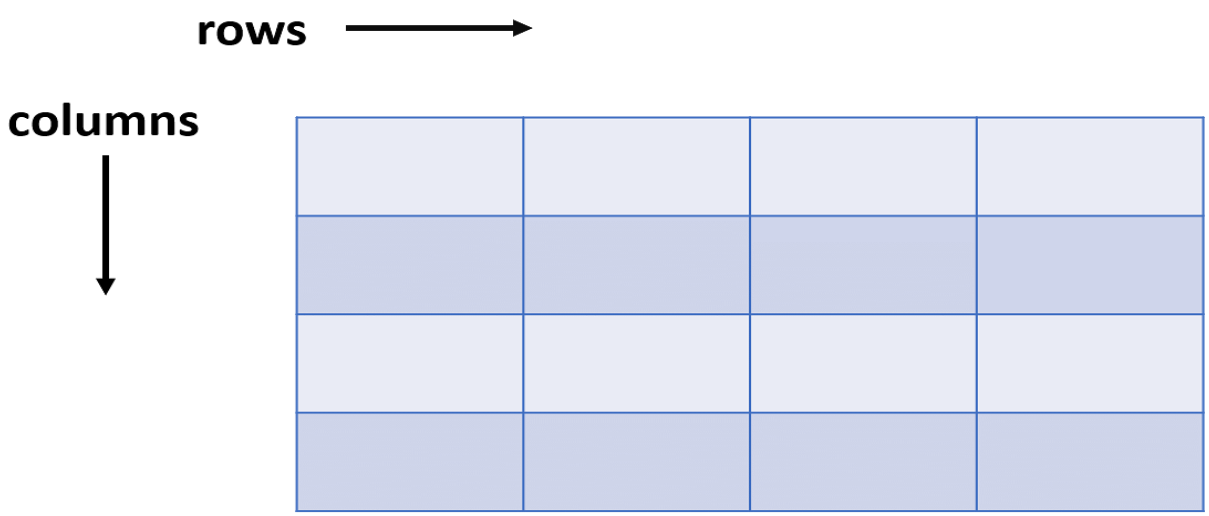
Chapter 8 Programming

Handout on Two-Dimensional Arrays

Two-dimensional arrays or multi-dimensional arrays are arrays where the data element's position is referred to, by two indices. The name specifies two dimensions, that is, row and column. Two-dimensional arrays or multi-dimensional arrays are arrays where the data element's position is referred to, by two indices. The name specifies two dimensions, that is, row and column.

What Are Two-Dimensional Arrays?

Two-dimensional arrays can be defined as arrays within an array. 2D arrays erected as metrics, which is a collection of rows and columns. It is common to design 2D arrays to accomplish a database that is similar to the data structure.



The Need for Two-Dimensional Arrays

Using 2d arrays, you can store so much data at one moment, which can be passed at any number of functions whenever required.

Picture this, a class consists of 5 students, and the class has to publish the result of all those students.

You need a table to store all those five students' names, subjects' names, and marks.

For that, it requires storing all information in a tabular form comprising rows and columns.

A row contains the name of subjects, and columns contain the name of the students.

That class consists of four subjects, namely English, Science, Mathematics, and Hindi, and the names of the students are first, second, third, fourth, and fifth.

	First	Second	Third	fourth	Fifth
English	**10**	**55**	**90**	**70**	**60**
Science	**50**	**60**	**75**	**65**	**95**
Mathematics	**95**	**75**	**55**	**85**	**45**
Hindi	**35**	**95**	**65**	**55**	**75**

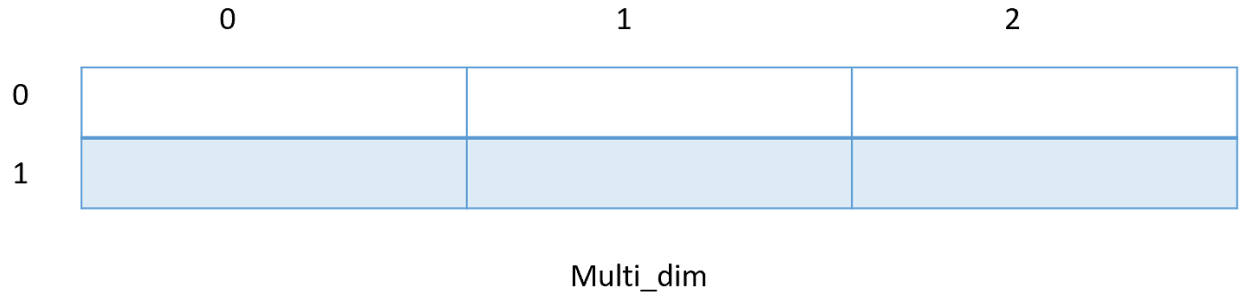
Declaration of Two-Dimensional Arrays

The syntax of two-dimensional arrays is:

```
Data_type name_of the array[rows][index];
```

Here is one example:

```
int multi_dim[2][3];
```



In the above example, the name of the 2d array is multi_dim consisting of 2 rows and three columns of integer data types.

Initialization of Two-Dimensional Arrays

There are two methods to initialize two-dimensional arrays.

Method 1

```
int multi_dim[4][3]={10,20,30,40,50,60,20,80,90,100,110,120};
```

Method 2

```
int multi_dim[4][3]={{10,20,30},{40.50,60},{70,80,90},{100,110,120}};
```

Here are two methods of initialization of an element during declaration. Here, the second method is preferred because the second method is more readable and understandable so that you can clearly visualize that multi_dim 2D arrays comprise four rows and three columns.

Accessing Two-Dimensional Arrays

Accessing two-dimensional arrays can be done using row index value and column index value.

```
Name_of_the arrays[row_index][column_index];
```

int multi_dim[4][3]={{10,20,30},{40,50,60},{70,80,90},{100,110,120}};

Suppose, in this example, you want to access element 80.

Multi_dim[2][1];

	0	1	2
0	**10**	**20**	**30**
1	**40**	**50**	**60**
2	**70**	**80**	**90**
3	**100**	**110**	**120**

Note: indexing always starts with zero.

Output of the Elements in a Two-Dimensional Array

Printing elements of a two-dimensional array can be done using two for loops.

```c

#include <stdio.h>

int main()
{
    int i,j;
    int multi_dim[4][3]={10,20,30,40,50,60,70,80,90,100,110,120};
    for(i=0;i<4;i++)
    {
        for(j=0;j<3;j++)
        {
            printf(" %d", multi_dim[i][j]);
        }
    }

    return 0;
}

```

```
input
 10 20 30 40 50 60 70 80 90 100 110 120

...Program finished with exit code 0
Press ENTER to exit console.
```

To define a two-dimensional array in pseudocode, you would first declare the array and then specify its dimensions. For instance, you might write something like "Declare an array A[5][5]" to create a 5x5 array.

How do you define a two-dimensional array in pseudocode?

A two-dimensional array in pseudocode is defined as a set of items categorized in rows and columns.

In more detail, a two-dimensional array is essentially an array of arrays. It is a data structure where data is stored in a tabular form, i.e., in rows and columns. This

structure is particularly useful when you need to organise data in a way that makes it easier to visualise or work with, such as in a matrix or a grid.
To define a two-dimensional array in pseudocode, you would first declare the array and then specify its dimensions. For instance, you might write something like "Declare an array A[5][5]" to create a 5x5 array. This statement tells us that the array A has 5 rows and 5 columns, meaning it can hold a total of 25 elements.
Each element in the array can be accessed by its row index and column index. For example, A[2][3] refers to the element in the third row and fourth column of the array. It's important to note that in most programming languages, array indices start at 0, not 1. So the first element in the array would be A[0][0], not A[1][1]..
When working with two-dimensional arrays in pseudocode, you can use nested loops to traverse the array and perform operations on each element. The outer loop typically iterates over the rows, while the inner loop iterates over the columns. For example, you might write a loop like "For i = 0 to 4" and then within that loop, write another loop like "For j = 0 to 4" to access each element in a 5x5 array.
A two-dimensional array is a collection of items organized in rows and columns, like a table. You define it in pseudocode by declaring its size, for example, "Declare an array A[5][5]" for a 5x5 grid. Use row and column indexes to access specific elements, with counting starting at 0. To work through the entire array, you can use nested loops.

Two-dimensional arrays are a fundamental concept in computer science, serving as a crucial data structure for organizing and managing data efficiently. In this exploration, we focus on understanding the detailed properties, structure, and storage patterns of two-dimensional arrays, examining their utilization in tabular data organization and diverse computational scenarios.

Properties of Two-Dimensional Arrays

Two-dimensional arrays, conceptualized as tables comprising rows and columns, offer a systematic approach to storing collections of elements. Let's look into their inherent properties:

- **Dimensionality:** This refers to the array's two-dimensional nature, contrasted with a one-dimensional array's linear form. In practice, this equates to arrays having a height (number of rows) and a width (number of columns).
- **Homogeneity:** All elements in a two-dimensional array must be of the same data type, whether primitive (like '**int**' or '**char**') or complex (like objects of a class).
- **Fixed Size:** The size (rows x columns) of a two-dimensional array is set when it's declared and can't be dynamically altered. This necessitates careful pre-planning of the array's size before its use.

- **Contiguous Memory Allocation:** Memory for these arrays is allocated in contiguous blocks. Understanding this is essential for appreciating how memory access patterns affect performance, particularly in large arrays.

Structure and Storage Patterns

Memory Allocation Patterns

- **Row-Major Order:** Predominantly, programming languages use row-major order for storing two-dimensional arrays. Here, consecutive elements of a row are stored in adjacent memory locations, which impacts how we iterate over arrays for efficient memory access and performance.
- **Column-Major Order:** In some languages and specific applications, a column-major order is used, where elements in a column are stored contiguously. This affects memory access patterns, especially in operations that predominantly access data column-wise.

Indexing and Access

- **Indexing Elements:** Access to elements is through '**[row][column]**' indices. The first index typically selects the row, and the second index selects the column.
- **Base Address Calculation:** The memory address of elements can be calculated differently based on the storage order. For example, in row-major order, the address

is derived using **'Base_Address + ((row_index * number_of_columns) + column_index) * size_of_element'**.

Organising Data in Tabular Format

Two-dimensional arrays efficiently store and manage data in a tabular format. Here's how they are used:

Data Representation

- **Matrices:** In mathematics and physics, two-dimensional arrays are pivotal in representing matrices for various calculations and operations.
- **Database Tables:** Their resemblance to database tables with rows (records) and columns (attributes) makes them useful in data handling and processing.
- **Image Processing:** Used in storing images for processing, where each cell represents a pixel value with its coordinates corresponding to its position in the image.

Practical Computational Uses

The practical applications of two-dimensional arrays are extensive and varied:

- **Scientific Computing:** From modelling chemical interactions to simulating astrophysical phenomena, two-dimensional arrays find extensive use in these fields.

- **Game Development:** Essential in game programming for board games like chess or in representing terrain on maps.
- **Machine Learning and Data Analysis:** They help in structuring data sets for various algorithms, facilitating operations like feature extraction, classification, and clustering.

Challenges and Considerations

While two-dimensional arrays are invaluable, they come with their set of challenges:

- **Memory Consumption:** Given that space is allocated for the entire array, memory usage can be substantial, particularly with larger arrays.
- **Manipulation Complexity:** Operations like adding or removing rows/columns involve complex shifting of multiple elements, which can be computationally expensive.
- **Handling Sparse Data:** In scenarios with sparse data (where most elements are zeros), using a regular two-dimensional array leads to memory wastage. Alternative structures, such as sparse matrices or hash maps, might be more memory-efficient.

Iteration Strategies

- **Row-wise vs. Column-wise Iteration:** The iteration strategy should ideally align with the memory layout. For instance, row-wise iteration in row-major arrays leads to better cache performance due to the locality of reference principle.
- **Nested Loops:** Typical access patterns in two-dimensional arrays involve nested loops. The outer loop typically iterates over rows, and the inner loop over columns (or vice versa), depending on the operation and memory layout.

Best Practices

- **Memory Efficiency:** Careful consideration of array size and avoiding oversized arrays can lead to better memory management.
- **Data Structure Choice:** For specific scenarios like sparse data or when dynamic resizing is frequently required, other data structures or collections might be more suitable than two-dimensional arrays.

In summary, two-dimensional arrays are more than just a data structure; they're a conceptual framework that underpins many advanced computing and data organization scenarios. Their proper understanding and application are pivotal in harnessing the full potential of algorithmic solutions in computer science. Their role

in structuring data, coupled with the implications of their memory layout and iteration patterns, makes them a versatile tool in the programmer's toolkit.

When working with two-dimensional arrays in programming, some common errors to be mindful of include:

- **Index Out of Bounds:** Attempting to access or modify elements outside the array's defined rows and columns can cause an 'index out of bounds' error. Ensuring indices are within the array's size limits is crucial.
- **Incorrect Initialization:** Not correctly initializing the array or its elements can lead to unexpected behavior or errors. For instance, failing to initialize a two-dimensional array in languages that don't automatically initialize can result in accessing garbage values.
- **Memory Allocation Errors:** Particularly in languages like C++, failing to correctly allocate and deallocate memory for a two-dimensional array can lead to memory leaks or segmentation faults.
- **Looping Mistakes:** Common mistakes in looping over two-dimensional arrays include incorrect loop boundaries, incrementing the wrong loop variable, or using the wrong index order (row index vs. column index), which can lead to logic errors or inefficient code execution.

Careful attention to array indices, proper initialization, memory management, and logical structure of looping constructs are key to avoiding these errors.

The dimensions of a two-dimensional array directly impact its memory usage and computational efficiency. A larger array size, determined by the product of its rows and columns, translates to more memory consumption. For instance, an array with dimensions 1000x1000 will consume significantly more memory than one with dimensions 10x10, assuming each element occupies the same amount of memory. From a computational perspective, the time complexity of operations like traversing or modifying an array scales with its size. Larger arrays require more time to iterate through each element, making operations more computationally expensive. Efficient use of two-dimensional arrays thus depends on balancing the requirements for data storage against memory and computational constraints. Allocating an oversized array unnecessarily can lead to wastage of memory and slower processing, whereas too small an array might not be sufficient for the intended data storage, necessitating resizing or use of another data structure.

Two-dimensional arrays are pivotal in the representation and manipulation of tabular data in computer science. They extend the concept of a linear array into multiple dimensions, allowing for the representation of grids or matrices. For Higher Level

(HL) students, an in-depth understanding of these structures is necessary, particularly in algorithm design focusing on data manipulation and retrieval.

Understanding Two-Dimensional Arrays

Two-dimensional arrays, also known as 2D arrays, can be visualized as a grid consisting of rows and columns. Each element in this array is accessed using two indices – one for the row and another for the column.

- **Conceptual Representation:** Think of a 2D array as a table, where each cell of the table is indexed with a row number and a column number.
- **Memory Storage:** Although represented as a grid, 2D arrays are stored in linear memory. In row-major order, which is commonly used, complete rows of the array are stored contiguously in memory.

Basic Operations on Two-Dimensional Arrays

Initialising, accessing, iterating over, and updating elements are fundamental operations.

- **Initialisation:** Define both the number of rows and the number of columns.

```pseudocode
var matrix : array[1..5, 1..5] of integer
```

- **Accessing Elements:** Use the indices to access an element. In a matrix **'[i, j]'**, **'i'** is the row index and **'j'** is the column index.

```pseudocode
var value = matrix[2, 3]
```

- **Iteration:** Typically done using nested loops – an outer loop for rows and an inner loop for columns.

```pseudocode
for i = 1 to numberOfRows do
    for j = 1 to numberOfColumns do
        process(matrix[i, j])
    end for
end for
```

- **Updating Elements:** Modification of elements is straightforward once the indices are known.

```pseudocode
matrix[3, 4] = newValue
```

Algorithmic Techniques in 2D Arrays

Searching in a Two-Dimensional Array

Sequentially scan each element. This method, while not efficient for large arrays, is straightforward.

- **Example:** Finding a specific value in a 2D array.

```pseudocode
function findValue(matrix, target)
    for i = 1 to numberOfRows(matrix) do
        for j = 1 to numberOfColumns(matrix) do
            if matrix[i, j] = target then
                return (i, j)
            end if
        end for
    end for
    return not found
end function
```

Sorting in Two-Dimensional Arrays

While sorting a 2D array, you can sort rows or columns based on specific criteria, or even sort the entire array as if it were one long 1D array.

- **Row Sorting:** Applying a conventional sorting algorithm (like quicksort or mergesort) to each row.

- **Column Sorting:** Similar to row sorting, but the sorting algorithm is applied to each column.

Grasping two-dimensional arrays enhances your problem-solving skills, enabling efficient data organization and manipulation for real-world applications like game development and image processing.

Matrix Operations

Common operations include matrix addition, subtraction, multiplication, and finding the transpose.

- **Transpose of a Matrix:** A transpose is obtained by flipping a matrix over its diagonal, switching the row and column indices of each element.

Pseudocode

```
function transpose(matrix)

var transpose: array [1..numberOfColumns(matrix), 1..numberOfRows(matrix)] of integer

for i = 1 to numberOfRows(matrix) do

for j = 1 to numberOfColumns(matrix) do
```

```
transpose[j, i] = matrix[i, j]

end for

end for

return transpose

end function
```

Dynamic Programming in 2D Arrays

Two-dimensional arrays are often used in dynamic programming to store intermediate results of subproblems.

- **Example:** The classic dynamic programming solution to the knapsack problem can be implemented using a 2D array.

Space Optimisation in 2D Arrays

For large 2D arrays with many default or empty values, consider using sparse arrays or hash tables to optimise space.

Application in Complex Problems

Game Board Implementations

From chess to tic-tac-toe, game boards are easily represented using 2D arrays, where each cell represents a state (empty, X, O, etc.).

Image Processing

Images, particularly those in pixelated formats, can be represented as 2D arrays, where each cell corresponds to a pixel's value. Operations on images, like rotation and inversion, often involve manipulation of these arrays.

Common Challenges and Best Practices

Boundary Checking

Always ensure that your indices are within the bounds of the array. Failing to do so can lead to errors or unexpected behaviour.

Iteration Efficiency

In situations where the array is large or the operation complex, consider the efficiency of your iteration strategy. Reducing the number of nested loops or terminating early when possible can significantly impact performance.

Memory Management

Large 2D arrays can consume significant amounts of memory.

2D Arrays

2D arrays are data structures capable of storing more than one item of data (value) at a time.

Unlike a 1D array, which is best visualised as a single list of values, a 2D array looks like a table of values.

To achieve this, a 2D array is technically an array which contains further arrays.

The individual values are still called elements, and all elements must be of the same data type.

Example:

The pseudocode to declare a 2D array might look like this:

DECLARE ExamMark : ARRAY[1:10, 1:3] OF INTEGER

Our example array called **ExamMark[]** has ten elements where we can store three values of the same data type (in this case integers).

Arrays usually start with index numbers of 0 unless intentionally declared otherwise (as above).

Notice how we now need two indexes (**i** and **j**) to read or assign values into the array.

ExamMark[]

		j		
		1	**2**	**3**
i	**1**			
	2			
	3			
	4			
	5			
	6		**99**	
	7			
	8			
	9			
	10			

To assign or read from a 2D array, we must refer to the array name and both index positions, e.g. ExamMark[6,2] contains 99.

It is common to use variables to represent the index values of a 2D array, e.g. ExamMark[**i**,**j**].

When a new value is assigned to an array element, it automatically replaces whatever was stored in that position previously.