PROGRAMMING METHODOLOGY
(PHƯƠNG PHÁP LẬP TRÌNH)

UNIT 10: Multidimensional Arrays

Unit 10: Multidimensional Arrays

Objective:

- Understand the concept and application of multi-dimensional arrays

Reference:

- Chapter 6: Numeric Arrays

Unit 10: Multidimensional Arrays (1/2)

1. One-dimensional Arrays (review)
   - 1.1 Print Array
   - 1.2 Find Maximum Value
   - 1.3 Sum Elements
   - 1.4 Sum Alternate Elements
   - 1.5 Sum Odd Elements
   - 1.6 Sum Last 3 Elements
   - 1.7 Minimum Pair Difference
   - 1.8 Accessing 1D Array Elements in Function

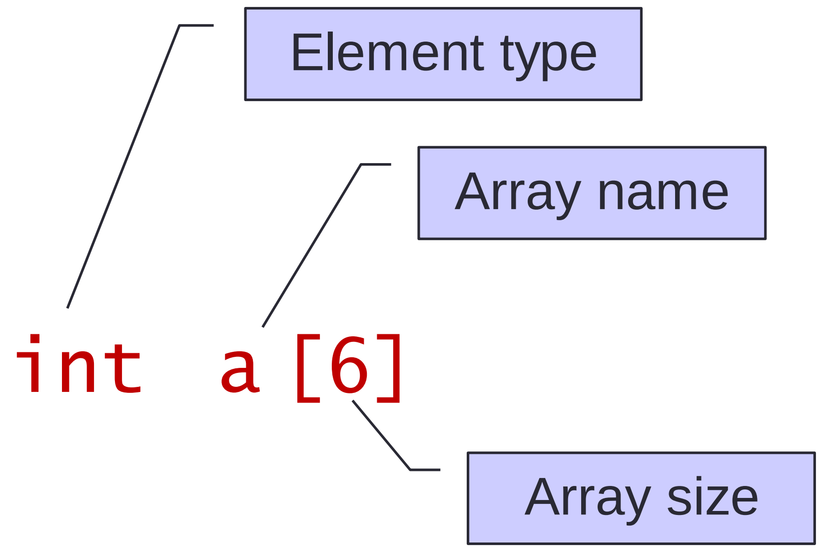
Unit 10: Multidimensional Arrays (2/2)

2. Multi-dimensional Arrays
   - 2.1 Initalizers
   - 2.2 Example
   - 2.3 Accessing 2D Array Elements in Function
   - 2.4 Class Enrolment
   - 2.5 Matrix Addition

1. One-dimensional Arrays (1/2)

Array

- A collection of data, called elements, of homogeneous type



a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
20	12	25	8	36	9

1. One-dimensional Arrays (2/2)

- Preparing an array prior to processing:

Initialization (if values are known beforehand):

```
int main(void) {
  int numbers[] = { 20, 12, 25, 8, 36, 9 };
  ...

  some_fn(numbers, 6);
}
```

Or, read data into array:

```
int main(void) {
  int numbers[6], i;
  for (i = 0; i < 6; i++)
    scanf("%d", &numbers[i]);
  ...

  some_fn(numbers, 6);
}
```

1.1 Print Array

```
void printArray(int arr[], int size) {
  int i;

  for (i = 0; i < size; i++)
    printf("%d ", arr[i]);
  printf("\n");
}
```

Calling:

```
int main(void) {
  int numbers[6];
  ...

  printArray(numbers, 6);
  printArray(numbers, 3);
}
```

Value must not exceed actual array size.

Print first 6 elements (all)

Print first 3 elements

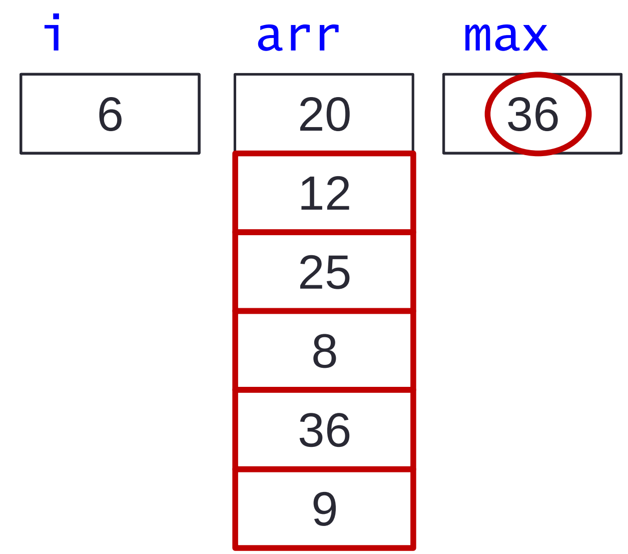
1.2 Find Maximum Value

- `findMax(int arr[], int size)` to return the maximum value in *arr* with *size* elements
- Precond: *size* > 0

```
int findMax(int arr[], int size) {
  int i, max;

  max = arr[0];
  for (i = 1; i < size; i++)
    if (arr[i] > max)
       max = arr[i];

  return max;
}
```



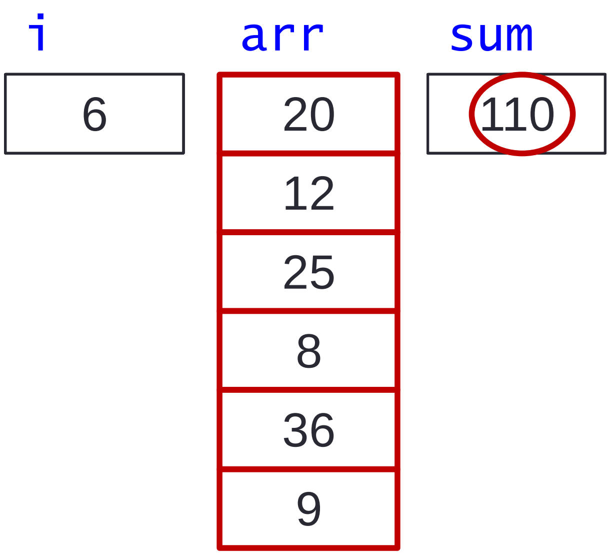
1.3 Sum Elements

- `sum(int arr[], int size)` to return the sum of elements in *arr* with *size* elements
- Precond: *size* > 0

```
int sum(int arr[], int size) {
  int i, sum = 0;

  for (i = 0; i < size; i++)
    sum += arr[i];

  return sum;
}
```



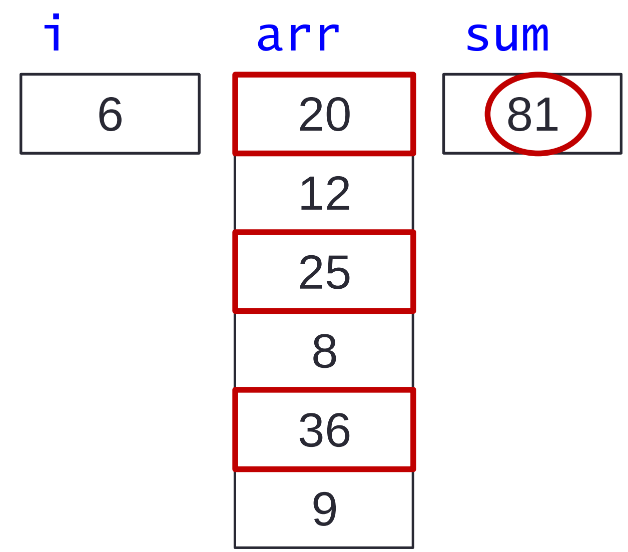
1.4 Sum Alternate Elements

- `sumAlt(int arr[], int size)` to return the sum of alternate elements (1st, 3rd, 5th, etc.)
- Precond: *size* > 0

```
int sumAlt(int arr[], int size) {
  int i, sum = 0;

  for (i = 0; i < size; i+=2)
    sum += arr[i];

  return sum;
}
```



1.5 Sum Odd Elements

- `sumOdd(int arr[], int size)` to return the sum of elements that are odd numbers
- Precond: *size* > 0

```
int sumOdd(int arr[], int size) {
  int i, sum = 0;

  for (i = 0; i < size; i++)
    if (arr[i]%2 == 1)
      sum += arr[i];

  return sum;
}
```

i	arr	sum
6	20	34
	12	
	25	
	8	
	36	
	9	

1.6 Sum Last 3 Elements (1/3)

- `sumLast3(int arr[], int size)` to return the sum of the last 3 elements among *size* elements
- Precond: *size* ≥ 0
- Examples:

numbers	sumLast3(numbers, size)
{ }	0
{ 5 }	5
{ 12, -3 }	9
{ 20, 12, 25, 8, 36, 9 }	53
{ -1, 2, -3, 4, -5, 6, -7, 8, 9, 10 }	27

1.6 Sum Last 3 Elements (2/3)

Thinking…

- Last 3 elements of an array arr
  - *arr*[*size* – 1]
  - *arr*[*size* – 2]
  - *arr*[*size* – 3]
- A loop to iterate 3 times (hence, need a counter) with index starting at *size* – 1 and decrementing it in each iteration

```
int i, count = 0;

for (i = size - 1; count<3; i--) {
   . . .
   count++;
}
```

- But what if there are fewer than 3 elements in *arr*?

```
int i, count = 0;
for (i = size - 1; (i >= 0) && (count<3) ; i--) {
   . . .
   count++;
}
```

1.6 Sum Last 3 Elements (3/3)

- Complete function:

```
int sumLast3(int arr[], int size) {
  int i, count = 0, sum = 0;

  for (i = size - 1; (i>=0) && (count<3); i--) {
    sum += arr[i];
    count++;
  }

  return sum;
}
```

1.7 Minimum Pair Difference (1/3)

- Is it true that all problems on 1D arrays can be solved by single loop? Of course not!
- Write a function `minPairDiff(int arr[], int size)` that computes the minimum possible difference of any pair of elements in *arr*.
- For simplicity, assume *size* > 1 (i.e. there are at least 2 elements in array).

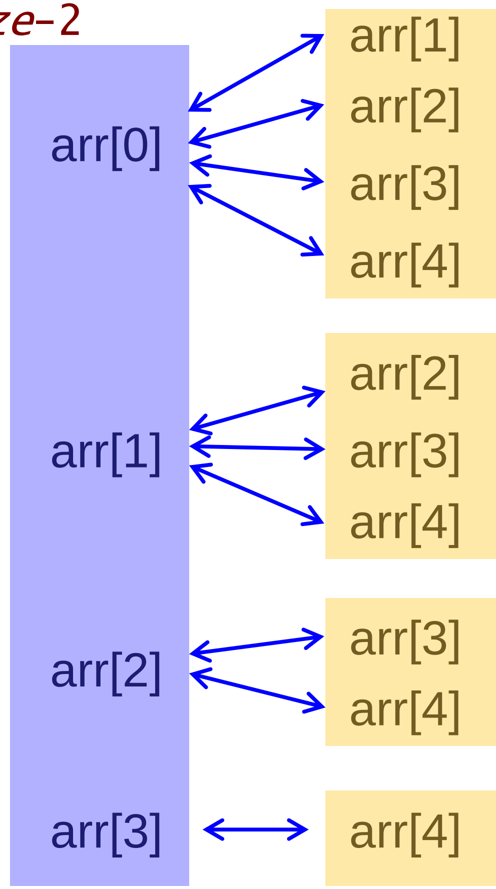
numbers	minPairDiff(numbers, size)
{ 20, 12, 25, 8, 36, 9 }	1
{ 431, 945, 64, 841, 783, 107, 598 }	43

1.7 Minimum Pair Difference (2/3)

Thinking… Eg: *size* = 5. Need to compute difference of

Outer loop index
i from 0 to *size*-2

Inner loop index
j from *i*+1 to *size*-1



~~Inner loop index~~
~~*j* from 1 to *size*-1~~

~~Inner loop index~~
~~*j* from 2 to *size*-1~~

~~Inner loop index~~
~~*j* from 3 to *size*-1~~

~~Inner loop index~~
~~*j* from 4 to *size*-1~~

1.7 Minimum Pair Difference (3/3)

The code…

Outer loop index
i from 0 to *size*-2

Inner loop index
j from *i*+1 to *size*-1

```
int minPairDiff(int arr[], int size) {
  int i, j, diff, minDiff;

  minDiff = abs(arr[0] - arr[1]); // init min diff.

  for (i = 0; i < size-1; i++)
    for (j = i+1; j < size; j++) {
      diff = abs(arr[i] - arr[j]);
      if (diff < minDiff)
        minDiff = diff;
    }

  return minDiff;
}
```

- This kind of nested loop is found in many applications involving 1D array, for example, sorting (to be covered later).
- In fact, this problem can be solved by first sorting the array, then scan through the array once more to pick the pair of neighbours with the smallest difference.

Code Provided

- Unit10_FindMax.c:
  - Section 1.2 Find Maximum Element
- Unit10_SumElements.c:
  - Section 1.3 Sum Elements
  - Section 1.4 Sum Alternate Elements
  - Section 1.5 Sum Odd Elements
  - Section 1.6 Sum Last 3 Elements
- Unit10_MinPairDiff.c:
  - Section 1.7 Minimum Pair Difference

1.8 Accessing 1D Array Elements in Function (1/2)

A function header with array parameter,

```
int sum(int a[ ], int size)
```

Why is it not necessary to have a value in here to indicate the "real" size?

- A value is not necessary (and is ignored by compiler if provided) as accessing a particular array element requires only the following information
  - The address of the first element of the array
  - The size of each element
- Both information are known
  - For example, when the above function is called with
    ```
    ans = sum(numbers, 6);
    ```
    in the main(), the address of the first element, &numbers[0], is copied into the parameter a
  - The size of each element is determined since the element type (int) is given (in sunfire, an integer takes up 4 bytes)

1.8 Accessing 1D Array Elements in Function (2/2)

A function header with array parameter,

```
int sum(int a[ ], int size)
```

Why is it not necessary to have a value in here to indicate the “real” size?

- With this, the system is able to calculate the effective address of the required element, say a[2], by the following formula:

  Address of a[2] = base address + (2 × size of each element)

  where base address is the address of the first element
- Hence, suppose the base address is 2400, then address of a[2] is 2400 + (2 × 4), or 2408.

a[0]	a[1]	a[2]	a[3]	
5	19	12	7	...

2. Multi-dimensional Arrays (1/2)

- In general, an array can have any number of dimensions
- Example of a 2-dimensional (2D) array:

```
// array with 3 rows, 5 columns
int a[3][5];
a[0][0] = 2;
a[2][4] = 9;
a[1][0] = a[2][4] + 7;
```

	0	1	2	3	4
0	2				
1	16				
2					9

- Arrays are stored in row-major order
  - That is, elements in row 0 comes before row 1, etc.

a[0][0]	…	a[0][4]	a[1][0]	…	a[1][4]	a[2][0]	…	a[2][4]
	row 0			row 1			row 2	

2. Multi-dimensional Arrays (2/2)

- Examples of applications:

$$\begin{pmatrix} 3 & 8 & 2 \\ -5 & 2 & 0 \\ 1 & -4 & 9 \end{pmatrix}$$

matrix[3][3]

	1	2	3	…	30	31
Jan	32.1	31.8	31.9		32.3	32.4
Feb	32.6	32.6	33.0	…	0	0
:						
Dec	31.8	32.3	30.9		31.6	32.2

Daily temperatures: temperatures[12][31]

Emily

	Ex1	Ex2	Ex3
Lab1	76	80	62
Lab2	60	72	48
Lab3	76	80	62
Lab4	60	72	48
Lab5	58	79	73

Zass

	Ex1	Ex2	Ex3
Lab1	59	68	60
Lab2	0	0	0
Lab3	67	71	75
Lab4	38	52	35
Lab5	78	86	82

Jerna

	Ex1	Ex2	Ex3
Lab1	79	75	66
Lab2	90	83	77
Lab3	81	73	79
Lab4	58	64	52
Lab5	93	80	85

Suise

	Ex1	Ex2	Ex3
Lab1	52	50	45
Lab2	57	60	63
Lab3	52	59	66
Lab4	33	42	37
Lab5	68	68	72

Students' lab marks: marks[4][5][3]

2.1 Multi-dimensional Array Initializers

- Examples:

```
// nesting one-dimensional initializers
int a[3][5] = { {4, 2, 1, 0, 0},
                {8, 3, 3, 1, 6},
                {0, 0 ,0, 0, 0} };

// the first dimension can be unspecified
int b[][5] = { {4, 2, 1, 0, 0},
               {8, 3, 3, 1, 6},
               {0, 0, 0, 0, 0} };

// initializer with implicit zero values
int d[3][5] = { {4, 2, 1},
                {8, 3, 3, 1, 6} };
```

What happens to the uninitialized elements?

2.2 Multi-dimensional Array: Example

Unit10_2DArray.c

```
#include <stdio.h>
#define N 5     // number of columns in array
int sumArray(int [][N], int); // function prototype

int main(void) {
    int foo[][N] = { {3,7,1}, {2,1}, {4,6,2} };
    printf("Sum is %d\n", sumArray(foo, 3));
    printf("Sum is %d\n", sumArray(foo, 2));
    return 0;
}

// To sum all elements in arr
int sumArray(int arr[][N], int rows) {
    int i, j, total = 0;
    for (i = 0; i < rows; i++) {
        for (j = 0; j < N; j++) {
            total += arr[i][j];
        }
    }
    return total;
}
```

Second dimension must be specified; first dimension is not required.

Sum is 26
Sum is 14

2.3 Accessing 2D Array Elements in Function

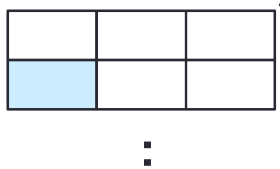
A function header with 2D array parameter,

```
function(int a[][5], ...)
```

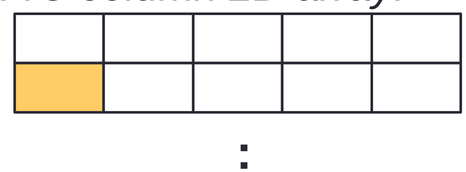
Why second dimension must be specified, but not the first dimension?

- To access an element in a 2D array, it <u>must know the number of columns</u>. It needs not know the number of rows.
- For example, given the following two 2D-arrays:

A 3-column 2D array:



A 5-column 2D array:



- As elements are stored linearly in memory in row-major order, element a[1][0] would be the 4th element in the 3-column array, whereas it would be the 6th element in the 5-column array.
- Hence, to access a[1][0] correctly, we need to provide the number of columns in the array.
- For multi-dimensional arrays, all but the first dimension must be specified in the array parameter.

2.4 Class Enrolment (1/5)

- A class enrolment system can be represented by a 2D array enrol, where the rows represent the classes, and columns the students. For simplicity, classes and students are identified by non-negative integers.
- A '1' in enrol[c][s] indicates student s is enrolled in class c; a '0' means s is not enrolled in c.
- Assume at most 10 classes and 30 students.
- Example of an enrolment system with 3 classes and 8 students:

	0	1	2	3	4	5	6	7
0	1	0	1	0	1	1	1	0
1	1	0	1	1	1	0	1	1
2	0	1	1	1	0	0	1	0

- Queries:
  - Name any class with the most number of students
  - Name all students who are enrolled in all the classes

2.4 Class Enrolment (2/5)

- Inputs:
  - Number of classes and students
  - Number of data entries
  - Each data entry consists of 2 integers s and c indicating that student s is enrolled in class c.
- Sample input:

```
Number of classes and students: 3 8
Number of data entries: 15
Enter 15 entries (student class):
3 1
0 0
0 1
1 2
2 0
2 1
2 2
3 2
7 1
6 0
5 0
4 1
4 0
6 2
6 1
```

	0	1	2	3	4	5	6	7
0	1	0	1	0	1	1	1	0
1	1	0	1	1	1	0	1	1
2	0	1	1	1	0	0	1	0

2.4 Class Enrolment (3/5)

```c
#define MAX_CLASSES 10
#define MAX_STUDENTS 30
int main(void) {
   int enrol[MAX_CLASSES][MAX_STUDENTS] = { {0} }, numClasses, numStudents;

   printf("Number of classes and students: ");
   scanf("%d %d", &numClasses, &numStudents);
   readInputs(enrol, numClasses, numStudents);

   return 0;
}
```

```
3 8
15
3 1
0 0
0 1
1 2
2 0
2 1
2 2
3 2
7 1
6 0
5 0
4 1
4 0
6 2
6 1
```

```c
// Read data into array enrol
void readInputs(int enrol[][MAX_STUDENTS],
                int numClasses, int numStudents) {
   int entries;   // number of data entries
   int i, class, student;

   printf("Number of data entries: ");
   scanf("%d", &entries);

   printf("Enter %d data entries (student class): \n", entries);
   // Read data into array enrol
   for (i = 0; i < entries; i++) {
      scanf("%d %d", &student, &class);
      enrol[class][student] = 1;
   }
}
```

	0	1	2	3	4	5	6	7
0	1	0	1	0	1	1	1	0
1	1	0	1	1	1	0	1	1
2	0	1	1	1	0	0	1	0

2.4 Class Enrolment (4/5)

- Query 1: Name any class with the most number of students

```
int classWithMostStudents
      (int enrol[][MAX_STUDENTS],
       int numClasses, int numStudents) {
  int classSizes[MAX_CLASSES];
  int r, c; // row and column indices
  int maxClass, i;

  for (r = 0; r < numClasses; r++)
    classSizes[r] = 0;
    for (c = 0; c < numStudents; c++) {
      classSizes[r] += enrol[r][c];
  }

  // find the one with most students
  maxClass = 0;  // assume class 0 has most students
  for (i = 1; i < numClasses; i++)
    if (classSizes[i] > classSizes[maxClass])
      maxClass = i;

  return maxClass;
}
```

	0	1	2	3	4	5	6	7	*Row sums*
0	1	0	1	0	1	1	1	0	5
1	1	0	1	1	1	0	1	1	6
2	0	1	1	1	0	0	1	0	4

2.4 Class Enrolment (5/5)

- Query 2: Name all students who are enrolled in all classes

```
// Find students who are enrolled in all classes
void busiestStudents(int enrol[][MAX_STUDENTS],
                     int numClasses, int numStudents) {
    int sum;
    int r, c;

    printf("Students who take all classes: ");
    for (c = 0; c < numStudents; c++) {
        sum = 0;
        for (r = 0; r < numClasses; r++) {
            sum += enrol[r][c];
        }
        if (sum == numClasses)
            printf("%d ", c);
    }
    printf("\n");
}
```

	0	1	2	3	4	5	6	7
0	1	0	1	0	1	1	1	0
1	1	0	1	1	1	0	1	1
2	0	1	1	1	0	0	1	0
	2	1	3	2	2	1	3	1

Column sums

Refer to Unit10_ClassEnrolment.c for complete program.

2.5 Matrix Addition (1/2)

- To add two matrices, both must have the same size (same number of rows and columns).
- To compute C = A + B, where A, B, C are matrices

  $c_{i,j} = a_{i,j} + b_{i,j}$

- Examples:

$$\begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} + \begin{pmatrix} -1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 2 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 2 & 0 \\ 2 & 2 & 1 \\ 1 & 2 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 10 & 21 & 7 & 9 \\ 4 & 6 & 14 & 5 \end{pmatrix} + \begin{pmatrix} 3 & 7 & 18 & 20 \\ 6 & 5 & 8 & 15 \end{pmatrix} = \begin{pmatrix} 13 & 28 & 25 & 29 \\ 10 & 11 & 22 & 20 \end{pmatrix}$$

2.5 Matrix Addition (2/2)

Unit10_MatrixOps.c

```
// To sum mtxA and mtxB to obtain mtxC
void sumMatrix(float mtxA[][MAX_COL], float mtxB[][MAX_COL],
               float mtxC[][MAX_COL], int row_size, int col_size) {
  int row, col;

  for (row=0; row<row_size; row++)
    for (col=0; col<col_size; col++)
      mtxC[row][col] = mtxA[row][col] + mtxB[row][col];
}
```

$$\begin{bmatrix} 10 & 21 & 7 & 9 \\ 4 & 6 & 14 & 5 \end{bmatrix} + \begin{bmatrix} 3 & 7 & 18 & 20 \\ 6 & 5 & 8 & 15 \end{bmatrix} = \begin{bmatrix} 13 & 28 & 25 & 29 \\ 10 & 11 & 22 & 20 \end{bmatrix}$$

Summary

- In this unit, you have learned about
  - Declaring 2D arrays
  - Using 2D arrays in problem solving