COMPILERS COURSE

MEMORY LAYOUT FOR MULTI-DIMENTIONAL ARRAYS

Each programming language has its own convention to store multi-dimentional arrays[1].

Suppose the following array declaration in the C programming language:

int A[3][2];

The way the elements of A are stored in memory is according to the following table[2]:

Array element	Address position relative to the first array element
…	
A[0][0]	0
A[0][1]	4
A[1][0]	8
A[1][1]	12
A[2][0]	16
A[2][1]	20
…	…

The expression to access a particular element of array A, A[i][j] is given by A+(i*2+j)*4 (multiplied by 4 considering a representation of 'int' with 4 bytes)

1) What is the error in the following function (consider the body of the function without errors) and what is the rule a C compiler must verify? Justify why in the C programming language this function as an error.

int f(int A[][]) {…}

2) Give an example of a programming language with another memory layout convention for 2D arrays.
3) Why the following equivalent Java code does not have the same error?

int f(int[][] A) {…}

4) What is the expression to calculate the address of an array element B[i][j][t] in case of the following array declaration in the C programming language?

int B[3][2][4];

5) Considering C code, what is the generic expression to calculate the address of the element position for an array A (type int) with N dimentions, with sizes S_1, S_2,…, S_N,

[1] Known in context of 2D arrays (matrices) as *column-major* (elements of a column in contiguous memory positions) or *row-major* (elements of a row in contiguous memory positions).

[2] *Row-major* layout.

considering a declaration based on the following one, and an array access like $A[i_1][i_2][i_3]...[i_N]$?

int $A[S_1][S_2][S_3]...[S_N]$;

6) The code in the textbox below multiplies each element of matrices A and B and the result is output to matrix C. One possible code optimization that can improve performance and save energy consumption is *loop interchange*[3]. By interchaning the two loops, Calvin found that the execution time reduced about 3× when using the following declarations for arrays A, B, and C, compiling the code with gcc –O3 and executing it in a Windows 7, 64-bit, Intel i5-2467M @1.60 GHz, 4 GB of RAM. What is the main reason for this execution time reduction?

```
#define M1 5000
#define N1 5000
float A3[M1][N1];
float A4[M1][N1];
float C1[M1][N1];
```

```
void DotProd(float A[M1][N1], float B[M1][N1],
             float C[M1][N1], int m, int n) {
   int i, j;

   for (j = 0; j < n; j++) {
      for (i = 0; i < m; i++) {
         C[i][j] = A[i][j]*B[i][j];
      }
   }
}
```

[3] In loop interchange the order of two nested loops are exchanged (see, e.g., https://en.wikipedia.org/wiki/Loop_interchange).