DYNAMICALLY GENERATING AND ACCESSING MULTIDIMENSIONAL ARRAYS

Massi Mapani (783767) Thabiso Magwaza (836403)

School of Electrical & Information Engineering, University of the Witwatersrand, ELEN4020A Electrical Engineering Data Intensive Computing

Abstract: This report presents the solution that a group of students produced for Laboratory one. The Laboratory required the implementation of three procedures and a main program that could dynamically generate multidimensional arrays. Assumptions and trade offs were made to come up with a solution, noting that other solutions are present for the Laboratory but are not discussed in this report. The program achieves all the functionality required of it in the specifications given by the Laboratory brief.

Key words: Dynamic memory, Multidimensional arrays, malloc(), realloc()

1. INTRODUCTION

One of the concepts in Data Intensive computing is the adjustment of memory distribution and the optimization of in-memory storage and run time of an executable file. C programming language provides functionality of being able to dynamically change a data type and the size of the variable and/or array of variables[1]. Being able to do this prevents the wastage of large sections of memory and gives the same memory multiple purposes of use in a program[1].

A group of two students was required to design and implement a C program which dynamically generates arrays and allocates memory. Then proceeds to carry out three procedures on the elements of the array. Firstly, to initialize the elements to zero, then randomly sets 10 % of the elements to 1. Lastly, the program randomly takes 5% of the total elements in the array and prints out the coordinate indices of the elements and the value at those indices.

This paper presents the design and thought process used to design the program and its, implementation. Section 2 presents the design parameters and success criteria and assumptions and trade-offs made for the program. Section 3 presents the Design methodology, mainly the approaches taken to implement the three procedures specified in the Laboratory brief. Section 4 discusses the testing of the code on different types of arrays and evaluates the performance of the code. Section 5 presents alternative ways to approach the Laboratory Exercise.

2. BACKGROUND

2.1 Design Parameters and Success Criteria

The Laboratory problem was handled by separating the procedures into individual functions that handled one task at a time. This allows for easy debugging and step by step problem solving. Modeling a multidimensional array required the group to simplify it into a single dimensional list similar to the way it is stored in memory, and using formulas with indices' of the elements to access an individual elements in the array.

- The program allows for arrays to be dynamically stored in memory. This results in the user being able to change the numerical data type of array as the program runs giving the program multiple write functionality.
- The initialization of all elements to zero removes uncertainty of what the elements are after this procedure and hence tracking the proceeding algorithms easy.
- The program needs to be able to generate a random process to select a tenth of the total elements in the array and change their value to one. Being random, a time based random function is best for the purpose.
- Then a fifth of the total elements are picked and their indices are printed out to the user and their value, being either 1 or 0 as well. .

The program would prove success full if all the above specifications are met.

2.2 Assumptions and Trade-Offs

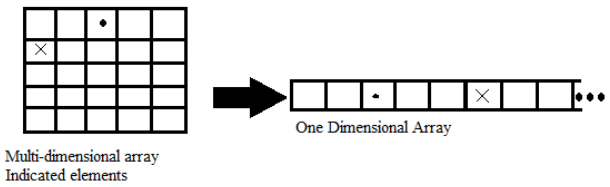
The program assumes a number scenarios which were used to design the program and the trade offs that were made. One of the assumptions made is that an an array of more than one dimension can not have an intermediate zero dimension and this was used to store the bounds of the array. Another point to note is that the algorithm for computing the indices of the array is computationally complex and will thus translate to a computationally expensive process

3. DESIGN METHODOLOGY

3.1 First Procedure

The first procedure required the initializing of elements to zero. This procedure's functionality provides insight into the array indexing of multidimensional arrays into a one dimensional list of elements. This is done by mapping all other indices by one indice by using a formula that makes use of the size of the array, the number of rows and columns and the bounds of the array. Figure 1 illustrates how the mapping is done in memory, as the elements are stored in contiguous memory from a 2 dimensional array to the one dimensional list. The formula for the mapping is given in equation (1) where zero products (empty prod-

Figure 1 : Illustrating how elements in a 2D *array* are mapped back to a 1D list



ucts) are equated to one.Each element in this procedure is initialized to zero.[2].

$$IndexforMappingto1D = \sum_{k=1}^{N} (\prod_{l=1}^{k-1} B_l) n_k \qquad (1)$$

Where
N is the number of dimensions of the array
k is a defined variable beginning from 0 and ending at N-1
B is the index bound value
n is the indix value of the array

3.2 Second and Third Procedure

The second procedure requires the identification of 10 % of the total elements, then randomly changing that number of elements to one from zero. The implementation of this is presented in Appendix A as pseudo-code.

The third procedure takes 5% of the total elements in the array, randomly selects these elements, keeps track of their indices and values and prints these out. This procedure makes use of the indexing techniques employed in the first procedure, however it uses of nested for loops to keep track of the coordinates and prints out the coordinates and the element at that index (either a zero or one). The implementation is presented in Appendix A.

3.3 Dynamically Generating Arrays

C programming language has four functions that allow a program to dynamically change, **malloc()** allocates byte storage without initialization, **calloc()** gives items or structures any number of bytes long, **free** removes a previously allocated item from the heap, and **realloc()** changes the size of previously allocated storage. For the program, changing a data type and size of an array is necessary to meet the specifications of the Laboratory problem.

The use of the **malloc()** function in C reserves contiguous memory for the generated array but does not initialize the elements in the array. This means that the array can change size and while the program runs. This is not possible for statically generated arrays as a set amount and type of memory is made for them. The use of **realloc()** allows the program to change the data type of the array during the program which provides the multipurpose functionality that a dynamically generated array has.

4. TESTING CODE

The code was tested iteratively by hard coding small arrays and using the terminal output to verify that the procedures were carried out. However, when arrays of larger dimensions and bounds were used, the run-time of the first procedure was significantly long and would have made the development of the rest of the code slow. Other testing methods will be used in sequential labs as these are low level methods for testing functionality.

5. CONCLUSION

The report has given a presentation of the design process that the group used to approach the problem presented for the lab. The code present on the group's Github repository meets the specifications of the Laboratory One. The code achieves multipurpose memory allocation and performs the mathematical procedures to access the dynamically generated elements. Alternatives to approaching this problem are presented in this paper and can be implemented in C.

REFERENCES

[1] B. W. Kerninghan and D. M. Ritchie, *The C Programming Language*, ser. C(Computer program language. Bell Telephone Laboratories Inc., Murray Hill, 1978.
[2] R. Ierusalimsch, *LUA 5.0, Programming in LUA*, 2003.

Appendix

Appendix A

Algorithm 1 Method 2: Setting 10% of the element to ones

$size \leftarrow$ total number of elements
$tenPercent \leftarrow size * 0.1$
FOR $i = 0 \leftarrow tenPercent$
$index \leftarrow$ Random number from 0 to $size$
$array[index] \leftarrow 1$
ENDFOR

Algorithm 2 Method 3: Printing out values at 5% random co-ordinates

$size \leftarrow$ numebr of elements
$fivePercent \leftarrow size * 0.05$
FOR $i = 0 \leftarrow fivePercent$
$coordinate \leftarrow$ random coordinate
Print $coordinate$
$index \leftarrow coordinateToIndex(coordinate)$
PRINT array$[index]$
ENDFOR

Algorithm 3 Generate Random Co-ordinate

$dim \leftarrow$ number of dimensions
$coordinate \leftarrow$ Dynamic array of size dim
FOR $i = 0 \leftarrow dim$
$coordinate[i] =$ random index within bound of dimension
ENDFOR