Software Engineering
Lab 3 Report

Emre Ozan Alkan
{emreozanalkan@gmail.com}
MSCV-5

29 October 2013

1 Dynamic allocation

Dynamically allocating an array of n integers.

Listing 1: Dynamic Allocation

```
1  /*
2   * Function: AllocateArray
3   * ----------------------------------
4   * Dynamically allocates and array of n integers
5   * and returning pointer to the first element of array
6   *
7   * n: Interger number as size of array
8   *
9   * returns: integer array pointer pointing to first element
10  * prints: nothing
11  */
12 int* AllocateArray(int n)
13 {
14     return new (std::nothrow) int[n];
15 }
```

2 Array deletion

Deleteing the array of n integers with given pointer pointing first element.

Listing 2: Array Deletion

```
1  /*
2   * Function: DeleteArray
3   * ----------------------------------
4   * Deleting the integer array given by user with pointer
5   *
6   * array: Interger pointer to array pointing first element
7   *
8   * returns: nothing
9   * prints: nothing
10  */
11 void DeleteArray(int *array)
12 {
13     delete [] array;
```

```
14      array = 0;
15      return;
16  }
```

3 Array initialisation using pointers

Initializating an array of n integers with the given pointer by user pointing to the frist element. All values assigned randomly between 0-99.

Listing 3: Initing Randomly

```
1  /*
2   * Function: InitArray
3   * ----------------------------
4   * Initialising an integer array with size n
5   * with random values between 0 and 99
6   *
7   * array: Interger pointer to array pointing first element
8   * n: Integer number size of the array
9   *
10  * returns: nothing
11  * prints: nothing
12  */
13 void InitArray(int *array, int n)
14 {
15      for(int i = 0; i < n; i++)
16      {
17          int rndNumber = rand() % 100;
18 #if _USE_POINTER_ARITHMETIC_
19          *(array + i) = rndNumber;
20 #else
21          array[i] = rndNumber;
22 #endif // _USE_POINTER_ARITHMETIC_
23      }
24
25      return;
26 }
```

4 On displaying an array using its pointer

Displaying the frist n values of an array using the given pointer pointing first element.

Listing 4: Displaying Array

```
1  /*
2   * Function: DisplayArray
3   * ----------------------------
4   * Displays the n values of an array using the pointer
5   * to its first element
6   *
7   * array: Interger pointer to array pointing first element
8   * n: Integer number size of the array
9   *
10  * returns: nothing
11  * prints: values of array size n
12  */
13 void DisplayArray(int *array, int n)
14 {
15      // http://www.cplusplus.com/reference/iterator/ostream_iterator/
16      // std::ostream_iterator<int> out_it (std::cout, "\n");
```

```
17      // std::copy ( array, array + n, out_it );
18
19      for(int i = 0; i < n; i++)
20      {
21  #if _USE_POINTER_ARITHMETIC_
22          std::cout<<(i+1)<<"th element: "<<*(array+i)<<std::endl;
23  #else
24          std::cout<<(i+1)<<"th element: "<<array[i]<<std::endl;
25  #endif // _USE_POINTER_ARITHMETIC_
26
27      }
28
29      return;
30  }
```

5 Reverse

Function to reverse the order of the array with given array pointer pointing to first element.

Listing 5: Reversing Array

```
1  /*
2   * Function: ReverseArray
3   * ----------------------------------
4   * Reverses the order of the given array by size n
5   *
6   * array: Interger pointer to array pointing first element
7   * n: Integer number size of the array
8   *
9   * returns: nothing
10  * prints: nothing
11  */
12 void ReverseArray(int *array, int n)
13 {
14      // http://www.cplusplus.com/reference/algorithm/reverse/
15      //std::reverse(array, array + n);
16
17      int temp = 0;
18
19      for(int* start = array, *end = array + (n - 1); start < end; start++, end--)
20      {
21          temp = *start;
22          *start = *end;
23          *end = temp;
24      }
25
26      return;
27 }
```

6 Swapping values of arrays represented by pointers

Swapping all the values of two same size arrays represented by two pointers given by user pointing first element/

Listing 6: Swapping Two Array

```
1  /*
2   * Function: SwapArrays
3   * ----------------------------------
4   * Swaps all the values of two arrays represented by pointers
```

```
 *
 * array1: Interger pointer to first array pointing to first element
 * array2: Interger pointer to second array pointing to first element
 * n: Integer number size of the both arrays
 *
 * returns: nothing
 * prints: nothing
 */
void SwapArrays(int *array1, int *array2, int n)
{
    //std::swap_ranges(array1, array1 + n, array2);

    int temp = 0;

    for(int i = 0; i < n; i++)
    {
#if _USE_POINTER_ARITHMETIC_
        temp = *(array1 + i);
        *(array1 + i) = *(array2 + i);
        *(array2 + i) = temp;
#else
        temp = array1[i];
        array1[i] = array2[i];
        array2[i] = temp;
#endif // _USE_POINTER_ARITHMETIC_
    }

    return;
}
```

7 On pointers used as function parameters

Sending pointers to function as value and as pointer reference. Honestly, I didn't know that difference. I needed to find many references and wrote some examples by my self. So function SpanArray2 is changing original pointer address to the end of the function. Before calling these functions, both arrays are printed. After function calls, the array sent to SpanArray2 was not able to print the values because its changed by the function.

Listing 7: Pointers as Parameters

```
/*
 * Function: SpanArray1
 * ----------------------------
 * Spans the array p with size n
 *
 * p: Interger pointer to array pointing to first element
 * n: Integer number size of the array p
 *
 * returns: nothing
 * prints: nothing
 */
void SpanArray1(int *p, const int& n)
{
    for(int i = 0; i < n; p++, i++);

    return;
}

/*
 * Function: SpanArray2
 * ----------------------------
 * Spans the array p with size n
 * Function getting pointer p as reference where
 * function is modifying the original pointer to
```

```
25  * point end of the array even after function returns
26  *
27  * p: Interger reference to pointer to array pointing to first element
28  * n: Integer number size of the array p
29  *
30  * returns: nothing
31  * prints: nothing.
32  *
33  * References:
34  * http://www.cplusplus.com/forum/beginner/96862/
35  * http://stackoverflow.com/questions/1257507/what-does-this-mean-const-int-var
36  * http://stackoverflow.com/questions/4424793/can-someone-help-me-understand-this-int-pr
37  * http://stackoverflow.com/questions/9340674/does-int-has-any-real-sense
38  */
39 void SpanArray2(int* &p, const int& n) // Reference to a pointer
40 {
41     for(int i = 0; i < n; p++, i++);
42
43     return;
44 }
```

8 Allocation and deallocation of monodimensional and bidimensional arrays represented by pointers

Here is the functions to play with matrices; matrix allocation, matrix initializing, displaying matrix and deleting the matrix. Remark: displaying the uninitialized matrix shows random values from memory, because created pointer looks some random place in memory, we see the old memory values.

8.1 Allocate Matrix

Allocating 2D integer array and returns pointer to it.

Listing 8: 2D Matrix Allocation

```
1  /*
2   * Function: AllocateMatrix
3   * ------------------------
4   * Allocating nxm size matrix and returning pointer of it
5   *
6   * Matrix n x m, n = row, m = cols
7   *
8   * n: Interger number representing size n of a matrix
9   * m: Integer number representing size m of a matrix
10  *
11  * returns: 2D integer pointer array of matrix
12  * prints: nothing
13  */
14 int** AllocateMatrix(int n, int m)
15 {
16     int **matrix = new int*[n];
17     for (int i = 0; i < n; i++)
18     {
19 #if _USE_POINTER_ARITHMETIC_
20         *(matrix + i) = new int[m];
21 #else
22         matrix[i] = new int[m];
23 #endif // _USE_POINTER_ARITHMETIC_
24
25     }
26
```

```
27      return matrix;
28  }
```

8.2 Initialize Matrix

Initializing 2D integer array with random values between 0-99

Listing 9: 2D Matrix Initializing

```
1  /*
2   * Function: InitializeMatrix
3   * ----------------------------
4   * Initializing given matrix with size nxm with random values between 0-99
5   *
6   * Matrix n x m, n = row, m = cols
7   *
8   * matrix: 2D integer pointer to matrix
9   * n: Interger number representing size n of a matrix
10  * m: Integer number representing size m of a matrix
11  *
12  * returns: nothing
13  * prints: nothing
14  */
15 void InitializeMatrix(int **matrix, int n, int m)
16 {
17      for(int i = 0; i < n; i++)
18          for(int j = 0; j < m; j++)
19          {
20 #if _USE_POINTER_ARITHMETIC_
21              *(*(matrix + i) + j) = rand() % 100;
22 #else
23              matrix[i][j] = rand() % 100;
24 #endif // _USE_POINTER_ARITHMETIC_
25
26          }
27
28      return;
29 }
```

8.3 Delete Matrix

Deleting 2D integer array

Listing 10: 2D Matrix Deletion

```
1  /*
2   * Function: DeleteMatrix
3   * ----------------------------
4   * Deleting the given matrix
5   *
6   * Matrix n x m, n = row, m = cols
7   *
8   * matrix: 2D integer pointer to matrix
9   * n: Interger number representing size n of a matrix
10  *
11  * returns: nothing
12  * prints: nothing
13  */
14 void DeleteMatrix(int **matrix, int n)
15 {
16      for (int i = 0; i < n; i++)
17      {
```

```
18 #if _USE_POINTER_ARITHMETIC_
19         delete [] *(matrix + i);
20 #else
21         delete [] matrix[i];
22 #endif // _USE_POINTER_ARITHMETIC_
23
24     }
25
26     delete [] matrix;
27     matrix = 0;
28     return;
29 }
```

8.4 Display Matrix

Displaying 2D integer array

Listing 11: 2D Matrix Display

```
1  /*
2   * Function: DisplayMatrix
3   * ------------------------------
4   * Displaying the given matrix with sizes n x m
5   *
6   * Matrix n x m, n = row, m = cols
7   *
8   * matrix: 2D integer pointer to matrix
9   * n: Interger number representing size n of a matrix
10  * m: Integer number representing size m of a matrix
11  *
12  * returns: nothing
13  * prints: the values of the matrix
14  */
15 void DisplayMatrix(int **matrix, int n, int m)
16 {
17     for(int i = 0; i < n; i++) {
18         for(int j = 0; j < m; j++) {
19 #if _USE_POINTER_ARITHMETIC_
20             std::cout<<*(*(matrix + i) + j)<<" ";
21 #else
22             std::cout<<matrix[i][j]<<" ";
23 #endif // _USE_POINTER_ARITHMETIC_
24
25         }
26         std::cout<<std::endl;
27     }
28     return;
29 }
```

9 A little bit of geometry

Computing the dot/inner product of the given vectors of any dimension/size.

Listing 12: Dot Product

```
1  /*
2   * Function: DotProduct
3   * ------------------------------
4   * Computing the Dot/Inner product of two nDimension vectors
5   *
6   * array1: First nDimensional vector with given type T
```

```
 * array2: Second nDimensional vector with given type T
 * dimension: integer number representing dimension of the vectors
 *
 * returns: Dot Product of the 2 vectors wtih specified type T
 * prints: nothing
 */
template<class T>
T DotProduct(T *array1, T *array2, int dimension)
{
    T dotProduct = 0;

    for(int i = 0; i < dimension; i++)
    {
#if _USE_POINTER_ARITHMETIC_
        dotProduct += *(array1+i) * *(array2 + i);
#else
        dotProduct += array1[i] * array2[i];
#endif // _USE_POINTER_ARITHMETIC_
    }

    return dotProduct;
}
```

10 Matrix multiplication in the general case

Multiplying arbitrary size 2 matrix if their size is compatible otherwise printing error.

Listing 13: Matrix Product

```
/*
 * Function: MatrixProduct
 * ---------------------------
 * Given matrix A and B, calculates multiplication of them and assigns result to C
 *
 * Matrix n x m, n = row, m = cols
 *
 * matrix1: 2 dimensional integer array representing matrix A
 * n: integer row size of matrix1
 * m: integer col size of matrix1
 * matrix2: 2 dimensional integer array representing matrix B
 * a: integer row size of matrix2
 * b: integer col size of matrix2
 *
 * returns: 2D integer pointer of matrix of result of the matrix multiplication
 * prints: Prints error message if matrix sizes are not compatible
 *
 * Method Used:
 * http://en.wikipedia.org/wiki/Loop_tiling
 * http://msdn.microsoft.com/en-us/library/hh873134.aspx
 * Original matrix multiplication:
 *   DO I = 1, M
 *     DO K = 1, M
 *       DO J = 1, M
 *         Z(J, I) = Z(J, I) + X(K, I) * Y(J, K)
 *
 */
int** MatrixProduct(int **matrix1, int n, int m, int **matrix2, int a, int b)
{
    if(m != a)
    {
        std::cerr<<"First array's column size should be equal with second array's row
            size"<<std::endl;
        return 0;
    }

```

```
36      int **product = AllocateMatrix(n, b);
37
38      for(int i = 0; i < n; i++){
39          for(int j = 0; j < b; j++){
40              for(int k = 0; k < a; k++){
41 #if _USE_POINTER_ARITHMETIC_
42                  *(*(product + i) + j) += *(*(matrix1 + i) + k) * *(*(matrix2 + k) + j);
43 #else
44                  product[i][j] += matrix1[i][k] * matrix2[k][j];
45 #endif // _USE_POINTER_ARITHMETIC_
46
47              }
48          }
49      }
50
51      return product;
52 }
```

11 Pointers arithmetic

Redoing everything above without using indexes and offset operator [] was the task. I achieved this by using pre-processors to switch between offsets and pointer arithmetic operations, otherwise implementing all above in new functions etc. would be tough task.

In "Lab3.h", I defined my pre-processor to switch between 2 implementation.

Listing 14: Pointer Arithmetic

```
1  /*
2   * Question 11: Pointers Arithmetic
3   * ------------------------------
4   * Redo functions with pointer arithmetic without
5   * using indexes and offset operator []
6   *
7   * true: using pointer arithmetic
8   * false: using indexes and offset operator:[]
9   */
10 #define _USE_POINTER_ARITHMETIC_ true
11 //#define _USE_POINTER_ARITHMETIC_ false
```