Report Lab 3

Tatiana Lopez Guevara

2 de noviembre de 2013

1. Array pointer basics

In this section we implemented the 4 basic operations performed on a dynamic monodimentional array:

- Allocation
- Initialization
- Display
- Deletion

These functions were used also in the two following sections where this kind of data type was used.

To initialize the array with random values, the `rand` function was called. This function returns a pseudo-random integer value between 0 and `RAND_MAX` which is a constant defined in `<cstdlib>` [1]. Since we want values between 0 and 99, the modulus operator must be used:

```
arr[i] = rand()%MAX_NUM;
```

Also, we want to obtain a different random value every time we execute the program. Therefore, we must make use of the `srand` function. Like [2] specifies, to generate random-like numbers the seed is chosen to be some distinctive runtime value. The most common one is the return value of the `time` function which is declared in the `<ctime>` header.

```
srand(time(NULL));
```

In this exercise, two arrays were created and initialized. After, the first array was reversed using the function `ReverseArray`, which iteraters up to the half of the array and interchanges each pair of opposite elements in it.

$$element[i] \Leftrightarrow element[SIZE - i]$$

Finally, the swap function was called for both of the arrays created. Two implementations of this function were made. The first one interchanges the values element by element and its completely safe. The last one interchanges the physical address to which each array is pointing. However, as discussed in class, this operation is not safe as will be discussed in the next section.

Listing 1: Dynamic Array Basics

```
/**
 * @brief AllocateArray reserves memory for n integers
 * @param n number of elements
 * @return the allocated array of integers
 */
int* AllocateArray(unsigned const int &n)
{
  int* arr = new int[n];
  return arr;
}

/**
 * @brief DeleteArray deletes an array dynamically assignated before
 * @param arr the pointer to the array
 */
void DeleteArray(int* arr)
{
  if(arr != NULL)
    delete [] arr;
}

/**
 * @brief InitializeArray initializes the array with random values
 * between 0 and MAX_NUM
 * @param arr the pointer to the array
 * @param n number of elements of the array
 */
void InitializeArray(int* const arr, unsigned const int &n)
{
  for(unsigned int i=0; i<n; i++)
  {
    arr[i] = rand()%MAX_NUM;
  }
}

/**
 * @brief DisplayArray is a helper function that allows
 * a pointer to an array to be displayed.
 * @param arr is the pointer to the first element of the array
 * @param N is the number of elements in the array
 */
void DisplayArray(const int* const arr, unsigned const int &n)
{
  for(unsigned int i=0; i<n; i++)
  {
    cout<<arr[i]<<endl;
  }
}

/**
 * @brief ReverseArray reverses the order of the elements of the
     array
 * @param arr pointer to the array
 * @param n number of elements of the array
 */
void ReverseArray(int* const arr, unsigned const int &n)
{
  int tmp;
  for(unsigned int i=0; i<n/2; i++)
  {
    tmp = arr[i];
    arr[i] = arr[n-1-i];
    arr[n-1-i] = tmp;
  }
}

/**
 * @brief SwapArraysDangerous swaps the values of two arrays by
 * changing the physical address of each one. This is dangerous!
 * @param arr1 pointer to the first array
 * @param arr2 pointer to the second array
 */
void SwapArraysDangerous(int* &arr1, int* &arr2)
{
  int* tmp;
  tmp = arr1;
  arr1 = arr2;
  arr2 = tmp;
}

/**
 * @brief SwapArrays swaps the elements of two given arrays.
 * The size of both arrays must be the same and equal to n
 * @param arr1 pointer to the first array
 * @param arr2 pointer to the second array
 * @param n number of elements in both arrays
 */
void SwapArrays(int* arr1, int* arr2, unsigned const int &n)
{
  int tmp;
  for(unsigned int i = 0; i<n; i++)
  {
    tmp = arr1[i];
    arr1[i] = arr2[i];
    arr2[i] = tmp;
  }
}
```

2. Spanned Arrays

Here we iterated through the elements of the array `p` making use of pointer arithmetic

```
p++
```

The difference between the 2 functions is the declaration or prototype of each. The first one receives a pointer to an array by value:

```
void SpanArray1(int *, const int &)
```

whereas the other receives it by reference:

```
void SpanArray2(int* &, const int &)
```

We therefore get the same behaviour as in the first lab with primitive variables. If the parameter is passed by value and it is modified within the function, then the modifications will only be reflected in the function's scope. However, in the second call, the increment performed on the pointer inside the function is actually affecting the address of the first element in the invoker function's pointer. This leaves it in an unwanted state and the `DeleteArray` for this variable fails:

```
lab3(7496) malloc: *** error for object 0x1001038ec:
                   pointer being freed was not allocated
```

Listing 2: Span Array

```
/**
 * @brief SpanArray1 iterates through an dynamic array pointer
 * using pointer arithmetic. The variable is received by value.
 * @param p array pointer (by value)
 * @param n number of elements
 */
void SpanArray1(int *p, const int &n)
{
  for(int i=0; i<n; p++, i++) {}
}

/**
 * @brief SpanArray2 iterates through an dynamic array pointer
 * using pointer arithmetic. The variable is received by reference.
 * @param p array pointer (by reference)
 * @param n number of elements
 */
void SpanArray2(int* &p, const int &n)
{
  for(int i=0; i<n; p++, i++) {}
}
```

3. A little bit of Geometry

In this section, we implmented the inner product between two n-Dimentional vectors and the cross product between two 3D vectors.

To test the obtained results, we first create two random vectors and calculate their inner product, most likely obtaining a result different from zero. After that, we obtained a new vector `v3` from the cross product between `v1` and `v2`. Since the obtained vector must be orthogonal to the other two, we then check this fact by calculating the inner product between $< v1, v3 >$ and $< v2, v3 >$.

```
/**
 * @brief InnerProduct Calculates the n-D inner product between
 * the vectors v1 and v2 of size n1 and n2 respectively
 * @param v1 Vector 1
 * @param v2 Vector 2
 * @param n1 Size vector 1
 * @param n2 Size vector 2
 * @return The inner product between v1 and v2
 */
long long InnerProduct(const int * const v1, const int * const v2,
    const unsigned int &n1, const unsigned int &n2)
{
  if(n1 != n2)
  {
    cerr<<"Matrix dimmensions must agree"<<endl;
    return 0;
  }

  long long res = 0;
  for(unsigned int i=0; i<n1; i++)
  {
    res += v1[i] * v2[i];
  }

  return res;
}

/**
 * @brief CrossProduct Obtains the cross product between the
 * 3D vector v1 and v2. The obtained vector is orthogonal
 * to both v1 and v2.
 * @param v1 Vector 1
 * @param v2 Vector 2
 * @return Cross product between vector v1 and v2
 */
int *CrossProduct(const int * const v1, const int * const v2)
{
  int *vout;
  vout = AllocateArray(3);

  vout[0] = v1[1] * v2[2] - v1[2] * v2[1];
  vout[1] = v1[2] * v2[0] - v1[0] * v2[2];
  vout[2] = v1[0] * v2[1] - v1[1] * v2[0];

  return vout;
}

void someGeometryCaller()
{
  int n = 3;
  int *arr1, *arr2, *arr3;
  long long inner, inner0_1, inner0_2;

  arr1 = AllocateArray(n);
  arr2 = AllocateArray(n);

  InitializeArray(arr1, n);
  InitializeArray(arr2, n);

  cout<<"Array 1"<<endl;
  DisplayArray(arr1,n);

  cout<<"Array 2"<<endl;
  DisplayArray(arr2,n);


  inner = InnerProduct(arr1, arr2, n, n);
  cout<<"Inner product between Array 1 and Array 2 = "<<inner<<endl;


  arr3 = CrossProduct(arr1, arr2);
  cout<<"Cross product between Array 1 and Array 2: "<<endl;
  DisplayArray(arr3, n);


  cout<<"** BONUS: Checking the cross product ;)"<<endl;
  cout<<"   Since the cross product between two 3D-vectors gives us"<<endl<<
    "   an orthogonal vector to both of them, then the inner "<<endl<<
    "   product bewteen the result and any of them must be 0!!"<<endl<<endl;


  inner0_1 = InnerProduct(arr1, arr3, n, n);
  inner0_2 = InnerProduct(arr2, arr3, n, n);
  cout<<"   -> Inner product between Array 1 and Array 3 = "<<inner0_1<<endl;
  cout<<"   -> Inner product between Array 2 and Array 3 = "<<
      inner0_2<<endl<<endl;

  DeleteArray(arr1);
  DeleteArray(arr2);
  DeleteArray(arr3);
}
```

4. Bidimensional Dynamic Arrays

Just like in the first section, we implemented the 4 basic functions, but to manipulate bidimentional arrays.

When we display the matrix just after allocating it, but before initializing it, we get sometimes zeros and sometimes garbage. Dynamic allocation does not initialize values as opposed to static allocation and that is why it is mandatory to call the inialization function.

The Matrix multiplication function receives two bidimentional matrices `mat1` and `mat2` whith their respective sizes. A validation for correct dimentionality is performed before doing any operation. That is, the columns of the first matrix must be equal to the rows of the second matrix.

$$mat3_{n,p} = mat1_{n,k} \times mat2_{k,p}$$

Since we are now dealing with dynamic matrices, we can use the function with any NxM matrix, not like the static declaration in which we had to fix the number of columns.

Listing 3: Bidimentional Dynamic Arrays

```
/**
 * @brief AllocateMatrix reservers memory for a bidimentional
 * array of n rows and m columns
 * @param n number of rows
 * @param m number of columns
 * @return allocated matrix of nxm
 */
int **AllocateMatrix(const unsigned int &n, const unsigned int &m)
{
  int **mat = new int*[n];

  for(unsigned int i=0; i<n; i++)
    mat[i] = new int[m];

  return mat;
}

/**
 * @brief InitializeMatrix initializes the nxm matrix with random
 * values between 0 and MAX_NUM
 * @param mat matrix to initialize
 * @param n number of rows
 * @param m number of columns
 */
void InitializeMatrix(int ** const mat, const unsigned int &n, const
     unsigned int &m)
{
  for(unsigned int i=0; i<n; i++)
  {
    for(unsigned int j=0; j<m; j++)
    {
      mat[i][j] = rand() %MAX_NUM;
    }
  }
}

/**
 * @brief InitializeMatrixOnes initializes the nxm matrix with ones
 * @param mat matrix to initialize
 * @param n number of rows
 * @param m number of columns
 */
void InitializeMatrixOnes(int ** const mat, const unsigned int &n,
    const unsigned int &m)
{
  for(unsigned int i=0; i<n; i++)
  {
    for(unsigned int j=0; j<m; j++)
    {
      mat[i][j] = 0;
    }
  }
}

/**
 * @brief DisplayMatrix outputs to the standard output
 * the values of the matrix
 * @param mat matrix to display
 * @param n number of rows
 * @param m number of columns
 */
void DisplayMatrix(int const* const* const mat, const unsigned int &
    n, const unsigned int &m)
{
  cout<<" Matrix "<<mat<<" Contents: "<<endl<<"  ";
  for(unsigned int i=0; i<n; i++)
  {
    for(unsigned int j=0; j<m; j++)
    {
      cout<<mat[i][j]<<" ";
    }
    cout<<endl<<"  ";
  }
}

/**
 * @brief DeleteMatrix releases the space of a matrix with
 * n rows
 * @param mat matrix to be deleted
 * @param n number of rows
 */
void DeleteMatrix(int ** mat, const unsigned int &n)
{
  for(unsigned int i=0; i<n; i++)
  {
    delete [] mat[i];
  }
}

/**
 * @brief MatrixMultiplication is a generic function to multiply
 * two dynamic bidimentional arrays mat1 of size n1xm1 and
 * mat2 of size n2xm2. Recall that m1 and n2 must be equal for
 * the operation to be valid.
 * @param mat1 Matrix 1
 * @param mat2 Matrix 2
 * @param n1 Number of rows of matrix 1
 * @param m1 Number of columns of matrix 1
 * @param n2 Number of rows of matrix 2
 * @param m2 Number of columns of matrix 2
 * @return mat1 x mat2
 */
int **MatrixMultiplication(int const* const* const mat1, int const*
     const* const mat2,
    const unsigned int &n1, const unsigned int &m1,
    const unsigned int &n2, const unsigned int &m2)
{
  if(m1!=n2)
  {
    cerr<<"Matrix dimmensions must match ["<<n1<<","<<m1<<"] - ["<<
        n2<<","<<m2<<"]"<<endl;
    return NULL;
  }

  int **mmult;
  mmult = AllocateMatrix(n1, m2);
  InitializeMatrixOnes(mmult, n1, m2);

  for(unsigned int i=0; i<n1; i++)
  {
    for(unsigned int j=0; j<m2; j++)
    {
      for(unsigned int k=0; k<m1; k++)
      {
        mmult[i][j] += mat1[i][k] * mat2[k][j];
      }
    }
  }

  return mmult;
}
```

5. Pointers Arithmetic

A new file called `ptrarithmetic` (header and source code) was created in order to replicate all the functions but without using the offset operator. Basically, what we did was to change the access of the monodimentional arrays from `arr[idx]` to `*(arr + idx)`. In the case of matrices, the modification was from `mat[i][j]` to `*(*(mat + i) + j)`.

Referencias

[1] cplusplus. rand. http://www.cplusplus.com/reference/cstdlib/rand/, October 2013.

[2] cplusplus. srand. http://www.cplusplus.com/reference/cstdlib/srand/, November 2013.