Report Lab 2

Tatiana Lopez Guevara

2 de noviembre de 2013

1. Input/Output

In this section I played with the operators that handle the input and output.

One thing to notice here was that when I used `cin>>`*string_var* and the user input was composed of several words separated by space, only the first string token was read, so if I want to read a whole line, I have to use the `getline` function.

Also, the other characters stayed in the buffer and therefore the next command that reads from the standard input like `getline` would take this value. To solve this, the command `cin.ignore(INT_MAX, '\n')` was used. This clears max possible characters in the buffer or until it reaches an enter character [3].

It is also important to use `cin.clear()` to clear the error flag after reading, otherwise, if by some reason the user inputs a wrong type of data, then the error flag will activate and will ignore the next readings after it.

Listing 1: Input

```
/**
 * @brief exampleInputOutput illustrates the use of the
 * standard input and output commands and some tricks
 * like cin.ignore()
 */
void exampleInputOutput()
{
  string line;

  cout<<"Introducing the one and unque COUT function!"<<endl;
  cout<<"Please write a word and I'll read it with CIN: "<<endl;
  cin>>line;

```

```
  //This is needed because we want to flush non used characters
  cin.ignore(INT_MAX, '\n');

  cout<<"Ok, You entered "<<line<<" with CIN and I've displayed it
      using COUT. Magic!"<<endl;

  cout<<"Now, I want to test getline(). Please enter something else:
       "<<endl;
  getline(cin, line);
  cout<<"You entered: "<<line<<endl;
}
```

2. Parameters

Three diferent functions to swap were implemented to test the value or reference parameters and its effect on the affected variables. The first version of it (`swap_1`) receives the parameters by value, and therefore any change that is done inside the function does not prevail after the function ends. In this case this means that the function does not work correctly for swapping.

However, the second version of `swap_2` and `swap_3` receive the parameters by reference and pointers to the variables respectively and therefore the modifications are correctly preserved after the function finishes. The difference between these 2 calls is that in the last case you need to provide the address of the variable in the main by using the `&` operator.

Also, the declaration of the pointer version was changed a little by specifying that the parameter was a constant pointer to a variable integer [2]:

```
void swap_3(int* const, int* const)
```

Listing 2: Swap

```
/**
 * @brief swap_1 two values that are passed by value
 * This works fine inside the function, but the changes
 * do not persist after
 * @param a first value
 * @param b second value to swap
 */
void swap_1(int a,int b)
{
  int tmp;

  tmp = a;
  a = b;
  b = tmp;
  cout<<" --> Internal result swap_1 function: "<<a<<", "<<b<<endl;
```

```
}

/**
 * @brief swap_2 two values that are passed by reference
 * @param a first value
 * @param b second value to swap
 */
void swap_2(int &a,int &b)
{
  int tmp;

  tmp = a;
  a = b;
  b = tmp;

  cout<<" --> Internal result swap_2 function: "<<a<<", "<<b<<endl;
}

/**
 * @brief swap_3 two values that are pointers
 * @param a pointer to the first value
 * @param b pointer to the second value to swap
 */
void swap_3(int* const a, int* const b)
{
  int tmp;

  tmp = *a;
  *a = *b;
  *b = tmp;

  cout<<" --> Internal result swap_3 function: "<<a<<", "<<b<<endl;
}
```

3. Mutiple returns

C++ does not allow multiple return values. In order to implement the function `CartesianToPolar`, four parameters were needed: the first two provided the `a, b` input constant values and the last two were used to return the ρ and θ.

The difference between the functions for calculating the angle between `a, b` is that `atan2` takes into account the sign of both variables (since it receives 2 parameters) to determine the quadrant of the angle whereas `atan` only receives one parameter ($\frac{b}{a}$) and if this term is positive, the function doesn't know if the angle was in the first or the third quadrant [1].

Listing 3: Cartesian To Polar

```
/**
 * @brief cartesianToPolar returns the norm and angle from the
     elements
 * z = a + ib of a complex number
 * @param a real part
 * @param b imaginary part
 * @param norm returned value with the norm
 * @param angle returned value with the angle
 */
void cartesianToPolar(const double &a, const double &b, double &norm
    , double &angle)
{
  norm = sqrt(a*a + b*b);
  angle = atan2(b,a);
}
```

4. Default parameter

The most important things to remark with the `isMultipleOf` function was that the default value was especified in the prototype of the header and not on the source code or cpp file. Also, that there was no need to have the name of the default variable written, so in the end I had this declaration:

```
bool isMultipleOf(const int &, const int=2);
```

Listing 4: Multiple Of

```
/**
 * @brief isMultipleOf evaluates if a number p is multiple of
     another (q).
 * If no q is provided it returns if p is even.
 * @param p number we want to know if it is multiple of another
 * @param q number we want to evaluate p to see if it is multiple of
      it
 * @return true if it is multiple, false otherwise
 */
bool isMultipleOf(const int &p, const int q)
{
  return p%q==0;
}
```

5. Recursivity

Here I created a function that determines if a given number `p` is prime or not by validating recursively that it is only divisible by 1 and by itself.

A wrapper function was also created in order to initialize the call to the validation number n. Notice that we only need to check the divisibility up to the square root of the number, because this is the maximum divisor that a number can have.

Listing 5: isPrime

```
/**
 * @brief isPrime Determines if a number is prime or not recursively
 * @param p number
 * @param n iterator
 * @return
 */
bool isPrime(const unsigned int &p, const unsigned int &n)
{
  if(n<=1)            //Base case: number is always divisible by 1
    return true;
  else if (p%n==0)    //Number divisible by another different than 1 and itself (NOT PRIME)
    return false;
  else
    return isPrime(p, n-1);
}

/**
 * @brief isPrime Determines if a number is prime or not recursively
 * @param p number
 * @return true if the number is prime, false otherwise
 */
bool isPrime(const unsigned int &p)
{
  //We only need to check if the number is divisible by others
  //lower than the square root
  int psqr = sqrt(p);

  return isPrime(p, psqr);
}
```

6. Arrays

6.1. Monodimentional

In this example, the use of static arrays is very straightforward. For the dynamic array, I created 2 functions: one for allocating and other for destroying. Also, a `displayArray` function was created, which receives a pointer to an monodimentional array either static or dynamic. This was nice because I could reuse the function for both.

Listing 6: ArrayExample

```cpp
/**
 * @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 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(int *arr, const short &N)
{
  //Iterate through the array and display in stdout
  for(int i=0; i<N; i++)
  {
    cout<<arr[i]<<" ";
  }
  cout<<endl;
}

/**
 * @brief arraysExample Shows the use of the statically and dynamically
 * allocated arrays
 */
void arraysExample()
{
  //Static array creation
  int statarr[NSIZE];

  //Dyanmic array declaration and allocation [Welcome to life!]
  int *dynarr;
  dynarr = allocateArray(NSIZE);

  //Initialize the arrays with the corresponding index
  for(int i=0; i<NSIZE; i++)
  {
    statarr[i] = i;
    dynarr[i] = i;
  }
```

```
    //Display the contents of the array
    cout<<"Contents of static array: "<<endl;
    displayArray(statarr, NSIZE);

    cout<<"Contents of dynamic array: "<<endl;
    displayArray(dynarr, NSIZE);

    //Release the memory [I dont need you anymore!]
    deleteArray(dynarr);
}
```

6.2. Pascal's Triangle

In this exercice I implemented the same Pascal's triangle done in the previous lab but taking into account the previous states that are stored in the positions of the matrix.

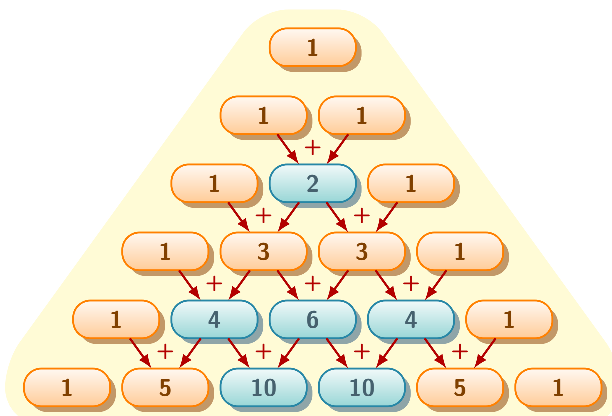

Figura 1: Pascal Triangle. Source [4]

This functionality can be divided into 3 parts:

- Allocation
- Initialization
- Calculation
- Display

Unlike the previous exercice, a generic function that receives the double pointer to the start of the matrix can not be used for both, static and dynamic implementations. Since the matrix can only be passed as a parameter of a function by explicitly specifying the size of the columns, it can only be used with static matrices of that size.

Listing 7: Pascal Static

```
/**
 * @brief initialize the matrix mat with NSIZE columns with
 * ones in the first column and zeros in all the other elements
 * @param mat is the matrix to initialize
 */
void initialize(int mat[][NSIZE])
{
  //Initialize the matrix
  for(int i=0; i<NSIZE; i++)
  {
    for(int j=0; j<NSIZE; j++)
    {
      //The first column is all ones
      if(j==0)
        mat[i][j] = 1;
      //The others are initialized to zero
      else
        mat[i][j] = 0;
    }
  }
}

/**
 * @brief calcPascalStatic Calculate the each position as the sum of
       the previous 2 elements
 * @param mat is the matrix to initialize
 */
void calcPascalStatic(int mat[][NSIZE])
{
  //Calculate the each position as the sum of the previous 2
      elements
  //from the row before
  for(int i=1; i<NSIZE; i++)
  {
    for(int j=1; j<NSIZE; j++)
    {
      mat[i][j] = mat[i-1][j] + mat[i-1][j-1];
    }
  }
}

/**
 * @brief displayPascalStatic Displays the values of the given
      matrix of NSIZE columns
 * @param mat is the matrix to initialize
 */
void displayPascalStatic(int mat[][NSIZE])
{
  //Display the values calculated!
```

```
  for(int i=0; i<NSIZE; i++)
  {
    for(int j=0; j<NSIZE; j++)
    {
      printf(" %3d ",mat[i][j]);
    }
    printf("\n");
  }
}

/**
 * @brief pascalTriangle1 calculates the elements of the Pascal Triangle
 * obtaining each value from the previous calculated values of the row before.
 */
void pascalTriangleStatic()
{
  int pascal[NSIZE][NSIZE];

  initialize(pascal);
  calcPascalStatic(pascal);
  displayPascalStatic(pascal);
}
```

The dynamic version of the calculation is much more flexible because it not only allows the user to input the desired last level of the triangle but also the functions can be invoked for any dynamically allocated integer matrix.

The code generated by using pointer arithmetic, although it achieves the same goal as by using he bracket operator `[]` is much less readable.

Listing 8: Pascal Dynamic

```
/**
 * @brief allocateMatrix reserves space for an array of n positions were
 * each position points to an array of n integers, thus giving nxn (squared matrix)
 * of reserved space
 * @param n number of rows and columns
 * @return the double pointer to the allocated space
 */
int **allocateMatrix(const unsigned int &n)
{
  int **mat = new int*[n];

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

  return mat;
}

/**
 * @brief deleteMatrix releases the allocated space for a matrix mat
```

```
 * of n columns
 * @param mat matrix to be released
 * @param n number of rows
 */
void deleteMatrix(int ** mat, const unsigned int &n)
{
  //For each row of the matrix delete the pointer array
  for(unsigned int i=0; i<n; i++)
  {
    delete [] *(mat+i);
  }
}

/**
 * @brief initializeMatrix initializes the matrix mat with n rows and n columns.
 * Assigns ones in the first column and zeros in all the other elements
 * @param mat is the matrix to initialize
 * @param n number of rows
 */
void initializeMatrix(int ** mat, const unsigned int &n)
{
  //Initialize the matrix
  for(unsigned int i=0; i<n; i++)
  {
    for(unsigned int j=0; j<n; j++)
    {
      //The first column is all ones
      if(j==0)
        *(*(mat + i) + j) = 1;
      //The others are initialized to zero
      else
        *(*(mat + i) + j) = 0;
    }
  }
}

/**
 * @brief calcPascalStatic Calculate the each position as the sum of the previous 2 elements
 * @param mat is the matrix to initialize
 */
void calcPascal(int ** mat, const unsigned int &n)
{
  //Calculate the each position as the sum of the previous 2 elements
  //from the row before
  for(unsigned int i=1; i<n; i++)
  {
    for(unsigned int j=1; j<n; j++)
    {
      *(*(mat + i) + j) = *(*(mat + i - 1) + j) + *(*(mat + i - 1) + j - 1);
    }
  }
}
```

```
/**
 * @brief displayPascalStatic Displays the values of the given
     matrix of NSIZE columns
 * @param mat is the matrix to initialize
 */
void displayPascal(int ** mat, const unsigned int &n)
{
  //Display the values calculated!
  for(unsigned int i=0; i<n; i++)
  {
    for(unsigned int j=0; j<n; j++)
    {
      printf(" %3d ",*(*(mat + i) + j));
    }
    printf("\n");
  }
}

/**
 * @brief pascalTriangleDynamic calculates the elements of the
     Pascal Triangle
 * obtaining each value from the previous calculated values of the
     row before.
 */
void pascalTriangleDynamic(unsigned const int &n)
{
  int **mat;

  mat = allocateMatrix(n);
  initializeMatrix(mat, n);
  calcPascal(mat, n);
  displayPascal(mat, n);
  deleteMatrix(mat, n);
}
```

7. Multidimensional arrays as function parameters

The use of static matrices as parameters was already used and explained in the previous section.

Listing 9: Matrix Parameters

```
/**
 * @brief multMatrix multiplies a 3x3 static matrix C=AxB
 * @param A Matrix 1
 * @param B Matrix 2
 * @param C Result matrix
 */
void multMatrix(int A[][SQRMAT], int B[][SQRMAT], int C[][SQRMAT])
```

```
{
  for(unsigned short i=0; i<SQRMAT; i++)
  {
    for(unsigned short j=0; j<SQRMAT; j++)
    {
      for(unsigned short k=0; k<SQRMAT; k++)
      {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}

/**
 * @brief displayMatrix Displays a 3x3 matrix
 * @param C matrix to display
 */
void displayMatrix(int C[][SQRMAT])
{
  for(unsigned short i=0; i<SQRMAT; i++)
  {
    for(unsigned short j=0; j<SQRMAT; j++)
    {
      cout<<C[i][j]<<" ";
    }
    cout<<endl;
  }
}
/**
 * @brief matMultCaller controlls the call to the matMult function
 */
void matMultCaller()
{
  int A[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
  int B[][3] = {{1, 2, -1}, {1, 2, -1}, {1, 2, -1}};
  int C[3][3] = {{}};

  multMatrix(A,B,C);

  displayMatrix(A);
  cout<<"TIMES "<<endl;
  displayMatrix(B);
  cout<<"EQUALS "<<endl;
  displayMatrix(C);
}
```

Referencias

[1] cplusplus. atan2. `http://www.cplusplus.com/reference/cmath/atan2/`, October 2013.

[2] Andrew Hardwick. The c++ 'const' declaration: Why & how. `http://`

duramecho.com/ComputerInformation/WhyHowCppConst.html, October 2013.

[3] Stack Overflow. Why would we call cin.clear() and cin.ignore() after reading input? http://stackoverflow.com/questions/5131647/why-would-we-call-cin-clear-and-cin-ignore-after-reading-input, October 2013.

[4] Tikz. Example: Pascal's triangle and sierpinski triangle. http://www.texample.net/tikz/examples/pascals-triangle-and-sierpinski-triangle/, October 2013.