Informe del desarrollo del Desafío 1- Informática 2

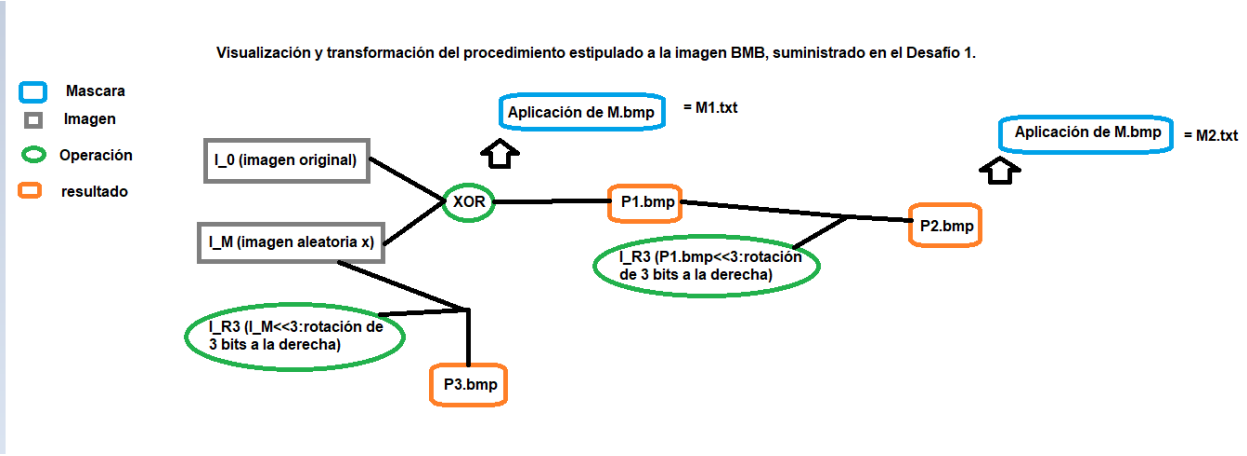
Introducción

En este documento se menciona conceptos fundamentales para iniciar el desarrollo del desafío 1 en el curso de Informática 2, son necesarios para la comprensión y la solución del problema propuesto por los profesores del curso, básicamente se hace una consulta en fuentes de información y/o bibliotecas acerca de las operaciones a nivel de bits(bitwise) en C++, con el fin de comprender su sintaxis y que operaciones se pueden hacer con ellas, estas son: AND(&), OR, XOR(^), NOT(~), desplazar a la derecha (<<), desplazar a la izquierda (>>). Todas ellas me permiten hacer operaciones a nivel de bits y están relacionadas con el desafío propuestos por los profesores, ya que una imagen BMP está compuesta por pixeles, cada pixel equivale a 3 bytes y 1 byte equivale a 8 bits, esta información es importante ya que es posible manipular imágenes a nivel de bits en C++ y en este caso según el problema planteado es necesario conocer cómo hacerlo para aprender a enmascarar y desenmascarar una imagen BMP, entonces como estudiante me surgen varias preguntas:

¿Cómo puedo desenmascarar una imagen BMP teniendo como recursos los archivos txt que me indican que transformación tuvo la imagen a nivel de bits?¿Cómo es posible revertir esos procesos para llegar a la imagen original?¿es posible hacerlo sin usar estructuras ni STL?¿Cómo sería la sintaxis en C++ para lograr ese objetivo?¿Es suficiente con las explicaciones que me dieron en clases o debo ir más allá para solucionar el problema?¿Cuál es el mapa de aprendizaje para cumplir con los objetivos y desafíos propuestos en el curso?

Pasos para la solución al Desafío 1:

1. Nuevamente se realiza una lectura minuciosa al documento del desafío 1 que nos compartió el profesor, si bien se había realizado una representación gráfica de la posible interpretación del problema planteado, se logra complementar una parte importante que no había podido inferir del documento, la cual era en que pasos de la transformación de la imagen "I_d" se generaban los archivos txt, entonces nuevamente comparto la imagen complementada con esa parte, que es la que se había compartido en el primer commit.



2. Teniendo en cuenta los requisitos para la entrega de la solución, cabe resaltar que el profesor indica que no se debe usar estructuras ni STL. Partiendo de esto, es importante recordar que se va a manipular archivos bmp(resultado de las transformaciones), archivos txt(contiene información de la transformación y la máscara aplicada) en los procesos. La lógica que se definió para recuperar la imagen original es la siguiente según el mapa de procesos construido a partir de la lectura del desafío 1:

A. **I_R3**(rotación de bits a la derechas) aplicado **a I_M**(imagen aleatoria) = **P3** entonces,
P3 = XOR(I_R3, I_M)

Nota: Como no se puede usar estructuras ni STL, entonces se toma como referencia las librerías usadas por el profesor en código ejemplo, también se empieza a desarrollar el código en el mismo main y al investigar sobre las librerías se obtiene la siguiente información:

- **#include <fstream> :** En este desafío me sirve para leer y escribir datos binarios o de texto en archivos, en este caso para usarla en los archivos bmp.
- **#include <QImage>:** Me sirve para cargar, manipular y guardar imágenes en diversos formatos, incluyendo BMP. Permite acceder fácilmente a los canales RGB de cada píxel. Puedo modificar píxeles individualmente con funciones como setPixel(), entre otras.
- **#include <QString>:** Me sirve para manejar cadenas de caracteres de forma segura, también para manejar rutas de archivo, me ayuda a interactuar con QImage.

En el mismo main del ejemplo del profesor se empieza trabajar, inicialmente se usan las librerías antes mencionadas, también las otras líneas de código básicas e indispensables para empezar a utilizar C++, las cuales son #include <iostream> y using namespace std; entonces primero que todo necesito cargar los insumos con los cuales voy a aplicar procesos inversos los cuales son:

- P3, I_M, M, : creo funciones para extraer la información de los bits de las imágenes bmp y las guardo en variables apuntadas, con el fin de manipular y/o comparar su contenido.

```
#include <iostream>
#include <fstream>
#include <QImage>
#include <QString>

using namespace std;

unsigned char *cargar_pixels(QString input, int &width, int &height);


int main()
{
    //Cargar la imagen transformada final (P3)
    int Width, Height;
    unsigned char *I0 = cargar_pixels("P3.bmp", Width, Height);
    if (!I0) return 1;

    //Cargar la imagen aleatoria (IM)
    int Width_IM, Height_IM;
    unsigned char *IM = cargar_pixels("I_M.bmp", Width_IM, Height_IM);

    if (!IM) {
        delete[] I0;
        return 1;
    }
```

```
    //Cargar la mascara (M)
    int Width_M, Height_M;
    unsigned char *M = cargar_pixels("M.bmp", Width_M, Height_M);
    if (!M) {
        delete[] I0;
        delete[] IM;
        return 1;
    }
```

- Posteriormente a la validación de la carga de los archivos que se van a utilizar para aplicar ingeniería inversa, se pasa a el desarrollo de la lógica para recuperar I_0 a partir I_d que según la información del documento no es más que el último paso aplicado P3.bmp. Si bien sabemos que las transformaciones aplicadas a I_0 se aplicaron en un orden desconocido, por medio del análisis y la ilustración de los pasos que se escudriñaron en el documento, se pudo definir la ecuación. Quiero resaltar algo que me parece importante, a pesar de que se tiene mucho información de los procesos aplicados a I_0, no siempre se necesita de toda para llegar a la solución; encontré un punto clave en la descripción de los pasos y es que la máscara “M” se aplica después de la operación XOR, entonces para pasar de P3.bmp a I_0 se planteó la siguiente lógica:

P1 = XOR (I_0, I_M) y **P3 = I_R3 (I_M)** entonces **I_R3_izq(P3) = I_M** y **I_0 = XOR (P1, I_M)**

Esta lógica se aplica gracias a la investigación sobre operaciones XOR y desplazamiento de bits, entonces se entiende que las operaciones XOR son reversibles y el desplazamiento de los bits también, por ejemplo si aplico x desplazamiento a la derecha lo puedo revertir con el mismo número de desplazamiento (x) a la izquierda, con esta lógica se obtuvo I_0 nuevamente.

```
//Validación de la carga de lo contrario se elimina I_d para evitar ocupar espacio de memoria innecesario
    if (!IM) {
        delete[] I_d;
        return 1;
    }


    //Revertir último XOR (P3.bmp = XOR(IR3, IM))
    Aplicar_XOR(I_d, IM, Width * Height * 3);

    //Revertir la rotación de 3 bits a la derecha
    Rotar_bits_izq(I_d, Width * Height * 3, 3);


    //Revertir primer XOR (P1.bmp = XOR(Io_original, IM))
    Aplicar_XOR(I_d, IM, Width * Height * 3);

```

- Se utiliza una función para guardar la imagen recuperada (I0_recuperada.bmp).

```
    // Guardar imagen recuperada
    exportImage(I_d, Width, Height, "I0_recuperada.bmp");

    int WidthRef, HeightRef;
    unsigned char *ImagenRef = Cargar_pixels("I_0.bmp", WidthRef, HeightRef);
    if (ImagenRef) {
        bool coinciden = Comparar_imagenes(I_d, Width, Height, ImagenRef, WidthRef, HeightRef);
        delete[] ImagenRef;
    }

    // Liberar memoria
    delete[] I_d;
    delete[] IM;

    return 0;
}
```

- En esta parte voy a colocar el contenido de las funciones que se invocaron en el main, para no hacer el informe tan extenso, su debida explicación la doy en la socialización de la solución del desafío en el siguiente link

```
//Función para aplicar el XOR entre dos imagenes haciendo la opercaion reversible
void Aplicar_XOR(unsigned char* imagen, unsigned char* IM, int size) {
    for (int i = 0; i < size; ++i) {
        imagen[i] ^= IM[i];     Assigned value is garbage or undefined [clang-analyzer-core.uninit
    }
}

//Funcion para revertir la rotación a la derecha
void Rotar_bits_izq(unsigned char *Datos, int Size, int Bits) {
    Bits %= 8;
    for (int i = 0; i < Size; i++) {
        //Realiza la rotación de bits a la Izquierda en cada uno de los Byte
        Datos[i] = (Datos[i] << Bits) | (Datos[i] >> (8 - Bits));
    }
}

```

```cpp
//Función para cargar los pixeles que se utiliza para (P3,I_M,I_0)
unsigned char *Cargar_pixels(QString input, int &width, int &height){
    // Cargar la imagen BMP desde el archivo especificado
    QImage imagen;

    // Cargar la imagen con conversión explícita a formato RGB888
    if (!imagen.load(input)) {
        cout << "Error: No se pudo cargar la imagen: " << input.toStdString() << endl;
        return nullptr;
    }

    // Convertir al formato correcto
    imagen = imagen.convertToFormat(QImage::Format_RGB888);

    // Verifica si la imagen fue cargada correctamente
    if (imagen.isNull()) {
        cout << "Error: No se pudo cargar la imagen BMP." << endl;
        return nullptr; // Retorna un puntero nulo si la carga falló
    }

    // Convierte la imagen al formato RGB888 (3 canales de 8 bits sin transparencia)
    imagen = imagen.convertToFormat(QImage::Format_RGB888);

    // Obtiene el ancho y el alto de la imagen cargada
    width = imagen.width();
    height = imagen.height();

    // Calcula el tamaño total de datos (3 bytes por píxel: R, G, B)
    int dataSize = width * height * 3;

    // Copia cada línea de píxeles de la imagen Qt a nuestro arreglo lineal
    for (int y = 0; y < height; ++y) {
        const uchar* srcLine = imagen.scanLine(y);              // Línea original de la imagen con posible padding
        unsigned char* dstLine = pixelData + y * width * 3;     // Línea destino en el arreglo lineal sin padding
        memcpy(dstLine, srcLine, width * 3);                    // Copia los píxeles RGB de esa línea (sin padding)
    }

    // Retorna el puntero al arreglo de datos de píxeles cargado en memoria
    return pixelData;
}

bool exportImage(unsigned char* pixelData, int width,int height, QString archivoSalida){

    // Crear una nueva imagen de salida con el mismo tamaño que la original
    // usando el formato RGB888 (3 bytes por píxel, sin canal alfa)
    QImage outputImage(width, height, QImage::Format_RGB888);

    // Copiar los datos de píxeles desde el buffer al objeto QImage
    for (int y = 0; y < height; ++y) {
        // outputImage.scanLine(y) devuelve un puntero a la línea y-ésima de píxeles en la imagen
        // pixelData + y * width * 3 apunta al inicio de la línea y-ésima en el buffer (sin padding)
        // width * 3 son los bytes a copiar (3 por píxel)
        memcpy(outputImage.scanLine(y), pixelData + y * width * 3, width * 3);
    }

    // Guardar la imagen en disco como archivo BMP
    if (!outputImage.save(archivoSalida, "BMP")) {
        // Si hubo un error al guardar, mostrar mensaje de error
        cout << "Error: No se pudo guardar la imagen BMP modificada.";
        return false; // Indica que la operación falló
    } else {
        // Si la imagen fue guardada correctamente, mostrar mensaje de éxito
        cout << "Imagen BMP modificada guardada como " << archivoSalida.toStdString() << endl;
        return true; // Indica éxito
    }

}

bool Comparar_imagenes(unsigned char* pixelData1, int width1, int height1, unsigned char* pixelData2, int width2, int height2) {
    // Verificar si las dimensiones coinciden
    if (width1 != width2 || height1 != height2) {
        cout << "Las imagenes tienen diferentes dimensiones." << endl;
        return false;
    }

    // Verificar cada pixel
    int dataSize = width1 * height1 * 3;
    for (int i = 0; i < dataSize; ++i) {
        if (pixelData1[i] != pixelData2[i]) {
            cout << "Las imagenes no coinciden." << endl;
            return false;
        }
    }

    cout << "Las imagenes coinciden." << endl;
    return true;
}
```